TEXAS INSTRUMENTS

SOFTWARE

TI-74 LEARN PASCAL USER'S GUIDE

TI–74 Learn Pascal User's Guide

This book was developed and written by:

Nancy Bain Barnett

With contributions by:

Robert G. Harr
Glen Thornton
Scott Thomson
Bill Petersen
Gary Von Berg
Chris Alley
Rosemary DeYoung
Robert E. Whitsitt, II

Chapter 1—Pascal on the TI-74 7
Introduction 7
Using this Manual 7

Chapter 2—Getting Started 9
Installing or Replacing a Cartridge 9
Initializing the Pascal System 10
The Key-reference Card 10
Leaving the Pascal System 10
Writing, Running, and Listing a Pascal Program 11
Editing Program Lines 12
Program Storage and Execution 15
    Saving a Program 15
    Executing a Stored Program 15
Review 16

Chapter 3—Computer Programming 17
Introduction 17
Pascal Language 17
    Program Format 18
        Program Heading 18
        Program Block 19
Pascal Syntax 20
    Identifiers 20
    Constants 21
    Reserved Symbols 21
Program Lines 22
    Line Numbering 22
    Renumbering Program Lines 23
    Indentation 23
    Line Length 24
    Punctuation 24
    Multiple-Statement Lines 24
    Statements on Multiple Lines 24
Comments 25
Interpreter Options 25
Output Statements 27
    The WRITELN Statement 28
    The WRITE Statement 28
Terminating Program Execution 29
    The Reserved Word END 29
    The HALT Statement 30
    The EXIT Statement 30
Using Statements without Line Numbers 30
Error Handling 30
Debugging a Program 32
Review 33

Chapter 4—Expressions . . . 36
Introduction . . . 36
Constant Declarations . . . 36
Variable Declarations . . . 37
  INTEGER Type . . . 38
  REAL Type . . . 39
  CHAR Type . . . 40
  STRING Type . . . 40
  BOOLEAN Type . . . 42
Setting the Values of Variables . . . 42
  Assignment Statements . . . 42
  Input Statements . . . 44
    The READLN Statement . . . 47
    The READ Statement . . . 47
    Using Prompts for Input . . . 49
Operators . . . 50
  INTEGER Data Operators . . . 50
    Arithmetic Operators . . . 50
    Relational Operators . . . 52
    Logical Operators . . . 53
  REAL Data Operators . . . 53
    Arithmetic Operators . . . 53
    Relational Operators . . . 54
    Logical Operators . . . 55
  Character Data Operators . . . 55
    Relational Operators . . . 55
    Logical Operators . . . 56
  STRING Data Operators . . . 56
    Relational Operators . . . 56
    Logical Operators . . . 57
  BOOLEAN Data Operators . . . 57
    Relational Operators . . . 57
    Logical Operators . . . 58
Operator Precedence . . . 60
Forming Expressions . . . 62
Functions . . . 63
  Integer Functions . . . 64
    Numeric . . . 64
    Memory . . . 65
    String . . . 68
    Ranking . . . 69
  Real Functions . . . 69
  Character Function . . . 71
  String Functions . . . 72
  Boolean Functions . . . 73
  Multi-Type Functions . . . 73
TYPE Declarations . . . 74

Data Type Formats 74
Unformatted Data 74
Formatted Data 76
Positioning the Cursor 78
Review 79

Chapter 5—Flow of Control 82
Introduction 82
Repetition Statements 82
The FOR Statement 83
The REPEAT Statement 86
The WHILE Statement 87
Nested Loops 89
Conditional Branch Statements 91
The IF Statement 91
Nested IF Statements 93
The CASE Statement 97
Unconditional Branch Statements 100
The GOTO Statement 100
LABEL Declarations 100
Review 101

Chapter 6—Arrays 103
Introduction 103
Declaring an Array Type 107
Random Access to a One-Dimensional Array 108
Two-Dimensional Arrays 109
Accessing the Elements in a Two-Dimensional Array . . . 110
Three-Dimensional Arrays 112
Arrays of Characters 113
Packed Arrays 115
Review 120

Chapter 7—Procedures and Functions 121
Introduction 121
Procedure Declarations 122
Function Declarations 125
Ending a Procedure or Function 126
Parameters 129
Global and Local Identifiers 131
Passing Information 132
One-Way Transfer 133
Two-Way Transfer 133
User-Defined Functions 133
User-Defined Procedures 135
Array Parameters 138
The FORWARD Declaration 143

Intrinsic Procedures 144
String Procedures 144
Array Procedures 146
Recursion 147
Review 150

Chapter 8—File Handling 152
Introduction 152
Data Format 153
Data Records 153
File Organization 153
Initializing a Mass-Storage Medium 153
Deleting a File 154
File-Processing Keywords 154
File Declaration 154
Opening and Closing a File 154
The RESET Procedure 154
The REWRITE Procedure 155
The CLOSE Procedure 156
File Input and Output 156
The EOLN and EOF Functions 156
File Input with READ and READLN 157
The PAGE Procedure 160
I/O Status 160
Review 161

Answer Key 163

Index 175

Chapter 1—Pascal on the TI-74

Introduction

The *TI-74 Learn Pascal User's Guide* was written to help you learn to write Pascal programs on the TI-74. The *TI-74 Learn Pascal Reference Guide* contains the features of TI-74 Pascal in alphabetical order followed by appendices that contain reserved words, error messages, and other reference material for use once you are familiar with Pascal.

Pascal is a programming language that was defined by Niklaus Wirth in the late sixties. In comparison to other programming languages such as BASIC, FORTRAN, or COBOL, Pascal is characterized by a highly disciplined, relatively formal syntax and structure. Thus, Pascal offers many advantages over other less-structured programming languages, such as an easily understood syntax, implicit error-checking, and program modularity. As a rule, programs written in Pascal can be easily moved from one computer to another.

The original definition of Pascal by Nicklaus Wirth is now known as standard Pascal. Newer versions of Pascal have been released that contain additions to the original definition. These versions often incorporate advanced features into standard Pascal. One of the most widely-used versions, UCSD Pascal, was developed for use on time-sharing systems and small computers. UCSD Pascal is a trademark of the Board of Regents of the University of California at San Diego. The version of Pascal implemented for the TI-74 is a subset of UCSD Pascal.

The Pascal *Solid State Software*™ cartridge is a learning aid that was designed to help you learn the Pascal programming language and to write Pascal programs in a very short time. This learning aid provides a fast, easy, and economical method of learning Pascal.

Using this Manual

This chapter is an introduction to the Pascal programming language for use with the Texas Instruments TI-74.

The lexical standards and syntactic conventions of the version of Pascal used with the TI-74 are discussed in the remaining chapters. At the end of each chapter are review questions. You can check your answers in the Answer Key, located after chapter 8.

Chapter 2 provides information on installing the *Solid State Software*™ cartridge, using the Pascal key-reference card provided with the cartridge, writing and executing a program, editing lines, and saving programs.

Chapter 3 provides an overview of a Pascal program and the rules for writing program lines.

Chapter 4 provides information on constructing expressions for use as program statements.

Chapter 5 is a discussion of statements that control the flow of a program.

Chapter 6 contains information on arrays.

Chapter 7 describes procedures and functions.

Chapter 8 describes the process of using files.

This manual was designed to enable you to begin writing Pascal programs immediately, even if you have never programmed or used Pascal before. You should, however, be familiar with the *TI-74 User's Guide*.

The most effective way to learn a programming language is t use it. You can learn Pascal on the TI-74 more quickly if you try the examples in this manual, complete the review questions at the ends of chapters 2 through 8, and then experiment with any programs you write. You cannot damage your computer by entering instructions. Any operation can be cancelled by pressing the **BREAK** and **CLR** keys or the **RESET** key.

Caring for the Cartridge

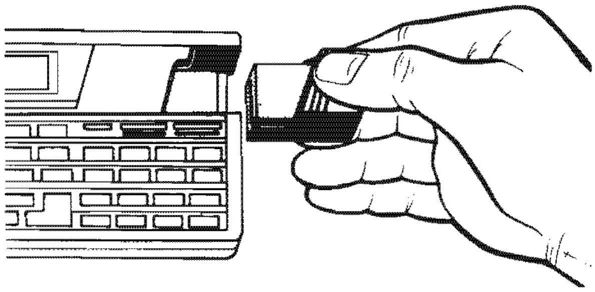
Handle the cartridge with the same care you would give any other piece of electronic equipment. You should:

- Avoid static electricity. Prior to handling the cartridge, touch a metal object to discharge any static electricity.
- Store the cartridge in its original container or in the cartridge port, on the upper right side of the TI-74.

Installing a Cartridge

The TI-74 is shipped with a port protector in the cartridge port. The port protector resembles a cartridge and is installed and removed in the same way.

1. Make sure the TI-74 is turned off. Installing a cartridge while the TI-74 is on may result in memory loss.
2. If the port protector or a cartridge is currently in the port, remove it by placing your thumb on the ridged area on top of the cartridge and sliding the cartridge to the right. Store the removed cartridge in its container.
3. Turn the Pascal cartridge so that the ridges are facing upward.
4. Insert the cartridge into the port, small end first.



5. Slide the cartridge to the left until it snaps into place.

You should keep a cartridge or the port protector in the port at all times to prevent the accumulation of dust.

Initializing the Pascal System

After the Pascal cartridge is installed, turn the console on by pressing the **ON** key. If a message is displayed, press the **CLR** or **ENTER** key to clear the display. When the flashing cursor appears, make sure the computer is in BASIC mode, then typ **run "pascal"** and press the **ENTER** key.

The computer then determines if a non-Pascal program is in memory. If there is such a program, the message `Erase program (y/n)?` is displayed. If you press **n**, the computer leaves the program in memory and returns to the BASIC command level. If you press **y**, the program in memory is erased and the message `Pascal System Initialized` is then displayed, informing you that the Pascal system is in command. Press the **CLR** or **ENTER** key to clear the message from the display and the cursor appears in column 1.

If no program is in memory or if a Pascal program is there, the message `Pascal System Initialized` is displayed after **run "pascal"** is entered.

The Overlay

The overlay provided with your Pascal cartridge fits over the keyboard to show the Pascal keywords that can be entered into the display with the **FN** key. To access a keyword, press the **FN** key and then the key under the keyword. Using the **FN** key can save you many keystrokes.

In addition to the symbols marked on the keyboard and overlay, the TI-74 has characters you access after pressing **CTL**. The *TI-74 User's Guide* describes the control keyboard.

Leaving the Pascal System

You can leave the Pascal system and return to the BASIC command level by entering the reserved word **BYE**. The BASIC command level is then automatically initialized and you can begin entering instructions in BASIC. Any Pascal program is erased from memory.

Note also that when the computer is reset or turned off by either the **OFF** key or the *Automatic Power Down*™ feature, the Pascal system is exited. When the computer is turned back on, the BASIC system is in command. You must enter **run "pascal"** to return to the Pascal system. Any Pascal program lines that were stored in memory remain there unless you have erased the computer memory with a NEW or NEW ALL command. Thus, after you have cleared the message `Pascal System Initialized` by pressing either the **CLR** or **ENTER** key, you can use the Pascal program in memory.

To enter a new program, you must first erase the memory by entering either **NEW** or **NEW ALL**.

If the Pascal system was running before the computer was turned off and you turn the computer on to program in BASIC, you must first initialize the BASIC system by entering the **NEW ALL** command.

Writing, Running, and Listing a Pascal Program

Enter the following program in your TI-74 exactly as it appears below. (Don't forget the period after END.)

```
100 PROGRAM example;
110 BEGIN
120 WRITELN('writeln is an output statement');
130 END.
```

To execute or run the program, press the **RUN** key (or type the word **run**) and then press the **ENTER** key. The message `writeln is an output statement` is displayed. Press the **CLR** key to clear the display.

You can see that the program lines have been stored in memory by typing the word **LIST** and pressing the **ENTER** key. The single line

```
100 PROGRAM example;
```

is displayed. The number 100 is the line number of the first line of the program. Each line of a TI-74 Pascal program must have a line number from 1 through 32766 followed by a space and at least one nonblank character.

Press the **ENTER** key to see each successive program line. When no more lines are displayed, LIST has displayed all of the lines in memory.

You can also use the ↑ and ↓ keys to view the stored lines. Pressing ↑ displays each program line in descending order; pressing ↓ displays the lines in ascending order.

Program lines are stored in numerical order, regardless of the order in which they are entered. For example, enter the following lines in your TI-74.

```
119 WRITELN('first statement displayed');
125 WRITELN('third statement displayed');
```

You can list the program (or use the ↑ and ↓ keys) to see that the additional program lines are stored in memory in numerical order.

To run the program, clear the display and enter the **RUN** command. The line

```
first statement displayed
```

is displayed. Press the **ENTER** key to see the next line.

```
writeln is an output statement
```

Press the **ENTER** key to see the next line.

```
third statement displayed
```

After this statement is displayed, press the **CLR** or **ENTER** key to clear the display. Press the ↓ key until the line

```
120 WRITELN('writeln is an output statement');
```

is displayed. Note that the characters `writeln is an output statement` are enclosed in apostrophes and parentheses. Characters enclosed in apostrophes are called a character string and are displayed exactly as they appear between the apostrophes.

The parentheses are used to enclose all of the items that the WRITELN is to display. For example, another character string can be displayed by this WRITELN by using the edit keys to insert another character string into line 120 as described in the next section.

Editing Program Lines

After you have used the ↓ key to display line 120, press the → key until the cursor is positioned over the closing parenthesis. Press the **SHIFT** → keys and then enter the following. (Don't forget the comma.)

```
,' 2nd character string'
```

When line 120 contains the following

```
120 WRITELN('writeln is an output statement',
' 2nd character string');
```

press the **ENTER** key to enter the line. The WRITELN in line 120 now has two character strings to display. To run the program, press the **RUN** and **ENTER** keys.

After the first line is displayed, press the **ENTER** key to view the next line. Note that the → indicator in the display is turned on as a signal that characters are in some columns to the right of column 31. Press **CTL** → to shift the characters so that column 25 is positioned in the first column of the display. You can then view the second string, `2nd character string`.

Press **CTL** ← or **CTL** ↑ to shift the displayed characters so that the first character of the line is in column 1 of the display. Press the **ENTER** key to see the last displayed line and then **CLR** to clear the display.

You can also display numbers in addition to character strings. Numbers do not have to be enclosed in apostrophes. To display the numbers 10, 345, and 867.5309 in the program, change the program lines by using the edit keys as shown below.

Press the ↑ or ↓ key until line 119 is displayed. Press → until the cursor is over the first apostrophe. Then type **10**. Press the **SHIFT** ← keys until the characters through the next apostrophe are deleted. When line 119 contains the following

```
119 WRITELN(10);
```

press the ↓ key to enter the line and to display line 120. Press → until the cursor is over the first apostrophe and type **345);**. Press **CTL** ↓ to clear all characters to the right of the cursor and the following line is then displayed.

```
120 WRITELN(345);
```

Press ↓ to enter the line and to display the next line. Then change line 125 to the following.

```
125 WRITELN(867.5309);
```

List the program (or use the ↑ or ↓ keys) to see that your program contains the following lines.

```
100 PROGRAM example;
110 BEGIN
119 WRITELN(10);
120 WRITELN(345);
125 WRITELN(867.5309);
130 END.
```

After you enter the RUN command, the number 10 is displayed. Press the **ENTER** key to view the next number, 345. Then press the **ENTER** key to view the last number, 867.5309.

Suppose you now want to delete lines 119 and 125 from the program. Press **CLR** to clear the display and type **DEL** (or press **FN** ←) and the line numbers as shown below.

```
DEL 119,125
```

Press **ENTER** and then LIST the program to see that lines 119 and 125 have been deleted. Your program should now contain the lines shown below.

```
100 PROGRAM example;
110 BEGIN
120 WRITELN(345);
130 END.
```

Note that you can use the DEL command with any of the following specifications.

Command	**Result**
DEL 100	Deletes line 100.
DEL 100,110,130	Deletes lines 100, 110, and 130.
DEL 100 –	Deletes line 100 and all following lines.
DEL – 100	Deletes line 100 and all preceding lines.
DEL 100 – 110, 120 – 130	Deletes lines 100 through 110 and lines 120 through 130.

If you want to delete all of the lines in memory, type **NEW** or **NEW ALL** and press **ENTER**. When you list the program (or use the ↑ or ↓ keys), no program lines are displayed.

Program Storage and Execution

You can save a program that you want to keep by using the SAVE command. To execute a program that has been stored, use the OLD command and the RUN command.

Saving a Program

The SAVE command is used to copy a program in memory to an external storage device. To store a program on a new medium, you must first format the medium. Note that if you format a medium that has data on it, you lose the data. For information on formatting, refer to the manual supplied with the peripheral device you are using.

The command

```
SAVE '1.myprog'
```

writes the program in memory to the medium on device 1. The program is saved under the filename "myprog." **Warning:** When you save a program on a medium that contains other programs, be sure to give the program in memory a name that does not already exist for a program on the medium. Otherwise, the program on the medium is deleted before the program in memory is written to the medium.

You can also protect a program when you save it, by using PROTECTED in the SAVE command. The program in memory remains unprotected, but the saved copy cannot be listed, edited, or stored. For example, the following SAVE command places a protected copy of the program in memory on an external device.

```
SAVE '1.myprog',PROTECTED
```

Note: Because a protected program can never be listed, edited, or stored, be sure to save an unprotected copy.

Executing a Stored Program

To execute a program stored on a peripheral device, the program must be loaded into memory by using the OLD or the RUN command. The OLD command is used when you want to load the program into memory. You can then verify that the program was loaded correctly, edit the program, or list the program before you run it. The statements shown on the next page illustrate loading a program into memory and verifying that it was loaded correctly.

OLD '1.myprog'
VERIFY '1.myprog'

To execute the program, enter **RUN**.

The RUN command can be used to retrieve and execute a program stored on a peripheral device. The command below loads a program into memory from a peripheral device and then executes it.

RUN '1.myprog'

Review Chapter 2

1. After you install the Pascal cartridge, you must enter what command before you can begin to program in Pascal?

2. You leave the Pascal system when the computer is turned off or when you enter what command?

3. In the program line

   ```
   120 WRITELN('writeln is an output statement');
   ```

   the number 120 is called the __________ __________.

4. 'writeln is an output statement' is known as a __________ __________.

5. Character strings must be enclosed in ________________

6. What is missing in the following program?

   ```
   100 PROGRAM example;
   110 BEGIN
   120 WRITELN('writeln is an output statement');
   130 END
   ```

7. To delete a line from a program, you would use the ______ command.

8. To save a program, you can use the __________ command.

9. What is wrong with the following command?

   SAVE "1.myprog"

10. Write the command that loads a program called "myprog" located on the medium in device 7.

Chapter 3—Computer Programming

Introduction

Programming languages such as FORTRAN, PL/1, BASIC, and Pascal are called high-level languages. These languages must use a compiler or interpreter to translate the language into one the computer can use. The Pascal available with the TI-74 uses an interpreter rather than a compiler to translate instructions into the machine language. When you enter an instruction, the interpreter scans it for syntax errors. If no syntax errors are found, the instruction is translated into the internal machine form and stored in memory. If a syntax error is detected, a message is displayed to inform you of the error so that you can correct the line and reenter it.

A Pascal system that uses a compiler requires that the entire program be written before any line is translated. No errors are detected until the entire program is entered to be compiled. If any errors are detected by the compiler, they must be corrected and the entire program must be compiled again to determine if other errors are present. After the program is compiled, it is stored to a file. The program can be executed after it is loaded into memory.

Because it uses an interpreter, TI-74 Pascal is easy to learn and use. After all the instructions in a program are entered, translated, and stored in memory, you can have the computer perform or execute the stored program with a single command.

Pascal Language

To solve a problem effectively with a computer, you must be able to reduce the solution of the problem to a sequence of steps that is both definite (always produces the same results) and finite (must end eventually). Such a sequence of steps is called an algorithm. Once you have developed the algorithm for solving a problem, you must translate the algorithm into a language the computer understands.

Pascal facilitates the conversion of algorithms into computer programs. First, the algorithm or sequence of steps is written in a general outline form. Then, the outline is broken down into a number of simpler programming tasks that are independent of each other. This stepwise refinement, called top-down design, results in Pascal programs that are organized as blocks of programming tasks. Top-down design produces programs that are organized, or "structured," in an easily understood manner.

In this manual, the elements of Pascal are grouped into five classes.

• Statements	instructions for the computer to perform
• Declarations	definitions of names
• Input	the information the program processes
• Output	the results
• Commands	instructions to the computer that cannot be performed in a program

A Pascal program is made up of statements and declarations. A typical program processes the data (input) entered from th keyboard or a storage medium. The information produced by the program is known as output.

Program Format

Every Pascal program must contain two parts: a program heading and a program block. The illustration below shows the two major parts of a Pascal program.

program heading	**PROGRAM** identifier;
	declarations
program block	**BEGIN**
	program body
	END.

Major Parts of a Pascal Program

Program Heading

The first line in a program must be a program heading. In the example,

```
100 PROGRAM example;
110 BEGIN
120 WRITELN('writeln is an output statement');
130 END.
```

line 100 contains the program heading, **PROGRAM** example; A program heading assigns a program name to all the lines that follow it.

Program Block

The program block consists of statements and declarations. A program block must have a program body. Declarations are optional, but if they are used, they must precede the program body.

The **program body** consists of the reserved word BEGIN followed by the statements that are to be executed and the reserved word END.

The line

```
120 WRITELN('writeln is an output statement');
```

is the only statement in the previous program that is performed. Note that a program body can contain no statements (and thus do nothing). The last line in every Pascal program must be the word END followed by a period (.).

In the remainder of this manual, the reserved word PROGRAM and the reserved words BEGIN and END that enclose the program body are printed in uppercase letters in **BOLDFACE**.

An illustration of the various elements that comprise a Pascal program is shown below.

program heading	**PROGRAM** identifier;
declarations	LABEL declaration
	CONST declaration
	TYPE declaration
	VAR declaration
	PROCEDURE/FUNCTION declarations
program body	**BEGIN**
	statements
	END.

Detailed Structure of a Program

Declarations are used in Pascal to define the names being used in a program. There are LABEL, CONST, TYPE, VAR, PROCEDURE, and FUNCTION declarations. Declarations are not executed when the computer encounters them; they are used to define names. The computer begins execution with the first statement in a program body. Declarations are optional, but if used, they must be in the order shown below. Note that after the LABEL, CONST, TYPE, and VAR declarations, PROCEDURE and FUNCTION declarations can be in any order.

Pascal Syntax

The syntax of a programming language defines the arrangement of its elements and the construction of its vocabulary and signs. In Pascal, the vocabulary includes words (identifiers) and numbers (constants); the signs are called reserved symbols.

Identifiers

Each word in Pascal is an identifier, and can be entered in either uppercase or lowercase characters or a combination of the two. Identifiers are of two types.

Reserved words have predefined meanings in Pascal and include words such as PROGRAM, BEGIN, and END. Reserved words (also called keywords) are displayed or printed in uppercase letters, regardless of how they are entered from the keyboard. (**Note:** When identifiers are entered from the playback buffer or from a user-assigned string, the identifier is displayed as it was entered.)

Reserved words must always be followed by a delimiter such as a space, a semicolon, a parenthesis, or an end of line. To help you identify these words, reserved words are printed in uppercase letters in this manual.

User-defined identifiers are names that you define in a program. Identifiers must start with a letter, consist of only letters and digits, and not be a reserved word.

Although you can enter as many as 80 characters for an identifier, the computer accepts only the first eight characters. Any others are discarded (unlike UCSD Pascal, which retains all entered characters but uses only the first eight). A user-defined identifier is always displayed or printed in lowercase letters regardless of how it is entered from the keyboard. You can distinguish between user-defined identifiers and reserved words by how they are displayed.

In this manual, identifiers that illustrate where a user-defined identifier can be used are printed in *italics*.

Constants

A constant is a value that does not change during the execution of a program. In Pascal, there are four types of constants: numeric, character, string, and Boolean constants.

Numbers such as 50 and −34.3 are called numeric constants. Positive numbers may be written with an optional plus (+) sign. Negative numbers must be preceded by a minus (−) sign. Commas and spaces are not allowed in numbers.

Numeric constants written without decimal points are called integers. The maximum integer allowed is 32767 and is called MAXINT. If an integer greater than 32767 or less than −32767 is entered from the keyboard, an error occurs.

Numeric constants written with a decimal point are called real numbers. Real numbers may be entered with any number of digits, but they are rounded to 13 or 14 digits for storage in the computer. If the number is entered with an odd number of digits to the left of the decimal point, a maximum of 13 digits are stored. If the number is entered with an even number of digits to the left of the decimal point, a maximum of 14 digits are stored. Only 10 digits of a real constant are displayed when a program is running, but all 13 or 14 digits are used in calculations and are displayed when a program is listed.

A character constant is a single character enclosed within apostrophes such as 'a', 'N', '*', or 'b'.

A string constant is a sequence of characters enclosed in apostrophes such as '2301 N. Ash #39' and 'Angie''s age is 12'. An apostrophe within a string constant is represented by two apostrophes.

A Boolean constant is the word TRUE or the word FALSE.

Reserved Symbols

Some of the symbols reserved for use in Pascal denote operations such as +, −, * (multiply), and / (divide). Other symbols such as ; and ' are used for syntactical purposes. The symbol .. denotes all the intervening values. For example, 1..5 means the numbers 1, 2, 3, 4, and 5. All of the reserved symbols listed on the next page have a predefined meaning in Pascal and are discussed in later sections. Note that a two-character symbol cannot have a space between the two characters.

+	–	*	/	=	<>	<	<=
>	>=	:=	.	,	;	:	..
(	)	[	]	(*	*)	{	}

Reserved Symbols in Pascal

Program Lines

The lines in a program must meet certain requirements and restrictions. These rules are listed below along with some features of the TI-74 that can facilitate writing your programs.

Line Numbering

Each line in a TI-74 Pascal program must begin with a numb followed by a space. A line number can be any integer from through 32766. Line numbers are used only to order and edi the program lines.

You can have the computer supply line numbers for your program by typing **NUM** (or pressing the **FN** → keys) and the pressing the **ENTER** key. The TI-74 displays 100 followed b a space with the cursor positioned where the first character of the line starts. After you type the statement and press th **ENTER** key, the TI-74 displays the number 110 followed by space and waits for you to enter a statement. When you hav finished entering all of the program lines, press either **ENTE** or **BREAK** when the next line number appears.

You can optionally specify where the numbering is to start and what increment is to be used. For example, entering **NU 1000,20** starts the line numbers at 1000 and uses increment of 20.

Note that if you enter the NUM command when there are program lines in memory, NUM displays a program line if on already exists for a given line number. If NUM is entered wit no options, program line 100 is displayed if it exists.

Note also that if a program line exists but its line number is not in the sequence NUM is using, the line is not displayed. For example, if **NUM 115,10** is entered, the numbering begi at 115 and increments to 125. If program line 120 is stored in memory, it is not displayed.

Renumbering Program Lines

After you have added and deleted program lines, you may want to renumber the lines in the program. The TI-74 renumbers the lines in a program when you enter **REN** (or **RENUMBER**).

For example, if the program

```
100 PROGRAM example;
110 BEGIN
119 WRITELN('first statement displayed');
120 WRITELN('writeln is an output statement');
125 WRITELN('third statement displayed');
130 END.
```

is stored in the computer and the command **REN** is entered, the line numbers will begin at 100 and increment by 10 as shown below.

```
100 PROGRAM example;
110 BEGIN
120 WRITELN('first statement displayed');
130 WRITELN('writeln is an output statement');
140 WRITELN('third statement displayed');
150 END.
```

You can optionally specify the beginning line number and the increment for RENUMBER. If using the given (or default) specifications would cause any line number to be greater than 32766, no line numbers are changed.

Indentation

Pascal statements can be indented to make the program more readable. Statements are often indented a number of spaces to show the layout of a program. You can add any number of spaces between a program line number and the beginning of the Pascal statement. These spaces are retained when the line is printed or displayed. Any other superfluous spaces are deleted.

The reserved words BEGIN and END that enclose the statements of a program are usually aligned with the reserved word PROGRAM. The statements that make up the program body are usually indented two to three spaces to show how they fit together and to make the program easier to read.

For example, the program from chapter 2 could be indented as shown on the following page.

```
100 PROGRAM example;
110 BEGIN
120   WRITELN('writeln is an output statement');
130 END.
```

Line Length

A line can be up to 80 characters long, including the line number. Additional characters typed at the end of the line replace the 80th character. When a line is entered, the interpreter removes any extra spaces (other than indention spaces). Note, however, that when the interpreter lists a Pascal line, it may add some spaces to the line for clarity. If these added spaces cause the length of the line to exceed 80 characters, an error occurs during the listing.

Punctuation

A semicolon is used to end a statement or declaration and separate it from the next one. This use of the semicolon enables you to enter more than one statement or declaration on a line and also to continue one statement or declaration over several lines. The computer ignores spaces entered on a line and continues reading characters as part of a line until it encounters a semicolon or the reserved word END.

A semicolon is not required to end all statements. For example, a semicolon is not necessary after the reserved word BEGIN and is optional before the reserved word END.

Multiple-Statement Lines

You can enter several statements on a line by separating them with a semicolon. BEGIN must be separated from the statement following it and END must be separated from the statement preceding it by some type of delimiter, such as a space or an end of line.

You can enter as many statements on a line as will fit into 80 characters. For example, the following program is entered on one line.

```
100 PROGRAM example; BEGIN WRITELN('One line');
      WRITELN('END') END.
```

If you run the program, press **ENTER** or **CLR** after the message `One line` is displayed to see the next line.

Statements on Multiple Lines

A statement can be entered on multiple lines. For example, the statement

```
140 WRITELN('third statement displayed');
```

can be entered as shown on the next page.

```
140 WRITELN
150 (
160 'third statement displayed'
170 )
180 ;
```

The interpreter continues reading lines until it has read a complete statement. In this case, the semicolon at line 180 signals to the interpreter the end of one statement and the beginning of another.

Comments

Comments make a program easier to understand and can appear anywhere in a program except in the middle of an identifier, constant, reserved word, or two-character symbol. The text of a comment is ignored by the computer.

Comments are placed in a program by enclosing them in the symbols (* and *) or { and }. These symbols may not be mixed. You can, however, use one type of symbol to enclose a comment that contains another comment enclosed by the other type of symbol. Note that a comment cannot be extended to the next line. The following example illustrates the use of comments.

```
100 PROGRAM example;
110 BEGIN (* start of program body *)
120   WRITELN('writeln is an output statement');
130 {Only statement}
140 END. (* example *)
```

It is good programming practice to include the name of the program in a comment on the last line of a program.

Note: Comments can contain specific instructions for the interpreter. This use of a comment is discussed in the next section and is **not** ignored by the computer.

Interpreter Options

You can specify three options to the interpreter to implement as it executes a program. The options enable you to:

- Have the computer wait or not wait after characters are sent to the display.
- Have the computer check or not check input/output operations.

An option is specified in a comment anywhere in a program after the reserved word **BEGIN** of the program body. As the computer scans the program lines during execution, an interpreter option is turned on or off as specified only if the statement containing the comment is executed. Only one option can be included per comment.

To specify an option, place a $ immediately after the opening delimiter, (* or { in a comment, followed by the letter w (wait) or i (input/output check). A plus sign (+) written after the letter causes the computer to turn on the option; a negative sign (–) written after the letter causes the computer to turn off the option.

For example, the following program line includes a comment containing an interpreter option.

```
110 BEGIN {$w-}
```

This comment causes the interpreter to turn off the wait that occurs when characters are displayed and continue program execution. The characters may be displayed so quickly, however, that you may not have time to view them.

For example, when the WRITELN in line 120 in the following program is performed, the computer leaves the characters in the display until the **ENTER** or **CLR** key is pressed. When either key is pressed, the comment causes the computer to turn off the wait option. Without the wait, the output is displayed so quickly that you cannot read it. After the WRITELN is performed, the wait option is turned back on in line 130 and the characters printed by line 140 remain in the display. Program execution is stopped until you press the **ENTER** key.

```
100 PROGRAM example;
110 BEGIN
120   WRITELN('first statement displayed');{$w-}
130    {turn off wait}
140    WRITELN('writeln is an output statement');{$w+}
150    {turn on wait}
160    WRITELN('third statement displayed');
170 END.
```

The other option enables you to determine if input and output operations are checked by the interpreter during program execution. If an input/output error occurs, the program is aborted. You can perform your own checking within the program, however, by turning off the input/output check option as shown below.

```
190 {$i-}
200 WRITELN('No I/O check');
```

Checking input/output operations in a program is discussed later in chapter 8.

After the RUN command is entered and the interpreter encounters the reserved word BEGIN in the program body, the interpreter turns on the default options shown below.

Letter	Default	Option
w	+	The interpreter suspends execution of a program when the program writes characters to the display. This delay gives you time to view the display. When either the **CLR** or **ENTER** key is pressed, program execution is resumed.
i	+	The interpreter checks input/output operations. See IORESULT in chapter 8.

utput Statements

Output statements are used to display the results of a program. An output statement includes (in parentheses) a list of items to be printed. The items in the list are separated by commas. Any item enclosed in apostrophes is called a character string and is printed exactly as it appears between the apostrophes. Any item not enclosed in apostrophes has its value printed, with no blanks printed before or after it.

The following sections describe using output statements to display data. Refer to chapter 8 for information on using output statements with files.

The WRITELN Statement

This statement displays the data listed within the parenthese and then advances the cursor to the next line. Normally, the interpreter option wait (w) is turned on so that you have time to view the displayed data. The **ENTER** or **CLR** key must be pressed to continue program execution when the wait option is in effect.

The WRITE Statement

This statement displays the data listed within the parenthese and leaves the cursor at the end of the displayed data. The next input/output operation to the display begins at the location of the cursor. Normally, the interpreter option wait (w) is turned off before a WRITE statement because more data is going to be either displayed or requested on the same line.

When the following program is run, the wait option (turned on when the interpreter encounters the reserved word BEGIN in line 110) causes the data displayed by the first WRITE statement to remain in the display until the **ENTER** key is pressed. The next output is then displayed and also remains in the display until the **ENTER** key is pressed.

Program	**Displa**
`100 PROGRAM example1;`	
`110 BEGIN`	
`120    WRITE(2+5,' is the answer');`	`7 is the answer`
`130    WRITE(' for number 10');`	`7 is the answer for number 10`
`140    WRITELN;`	`7 is the answer for number 10`
`150 END. (*example1*)`	

Note that line 140 actually displays nothing, but the characters in the display remain there until **ENTER** is pressed. Line 140 advances the cursor to the next line, where the next input or output will begin.

In the following program, the wait option is turned off before the WRITE statements are executed. The wait option is turned back on before the WRITELN in line 140 moves the cursor to the first column of the next line.

Program	**Display**
`100 PROGRAM example2;`	
`110 BEGIN {$w-}`	
`120   WRITE(2+5,' is the answer');`	
`130   WRITE(' for number 10');`	
`140   WRITELN {$w+};`	`7 is the answer for number 10`
`150 END. (*example2*)`	

In the following program, each line of output is displayed on a separate line with the WRITELN statement. The **ENTER** key can be pressed to view each succeeding line.

Program	**Display**
`100 PROGRAM example3;`	
`110 BEGIN`	
`120   WRITELN(2+5,' is the answer');`	`7 is the answer`
`130   WRITELN(' for number 10');`	`for number 10`
`140   WRITELN;`	`(displays a blank line and advances the`
`150 END. (*example3*)`	`cursor to the next line)`

Terminating Program Execution

The three valid methods of terminating program execution include the reserved word END, the HALT statement, and the EXIT statement.

The Reserved Word END

The reserved word END, followed by a period (.), appears after the last statement in a program. When END followed by a period is encountered, program execution stops. Note that the period must immediately follow END.

The HALT Statement

The HALT statement is used in abnormal situations to terminate program execution before the reserved word END. When HALT is executed, the program aborts, displaying the message `Programmed Halt.`

The EXIT Statement

The EXIT statement can be used to terminate a program before the reserved word END. When EXIT terminates program execution, no message is displayed.

Using the HALT and EXIT statements to end program execution is described in chapter 7.

Using Statements without Line Numbers

Many Pascal statements can be performed immediately by entering them without line numbers. In TI-74 Pascal, this type of statement is called an imperative and is executed as soon as the **ENTER** key is pressed. For example, the line

```
WRITELN('writeln is an output statement');
```

displays the message `writeln is an output statement` immediately after the **ENTER** key is pressed.

Note that an imperative must fit on a single line and must end with a semicolon. Refer to appendix C in the *TI-74 Learn Pascal Reference Guide* for a list of the statements that can be used as imperatives.

Error Handling

As you begin writing programs, you will find that some types of errors produce an error message as soon as you enter the line. You can use the **SHIFT PB** feature to display the erroneous line and use the edit keys to correct it.

Other types of errors in a program are not detected by the interpreter until you run the program. Errors can be detected at two different times after the RUN command has been entered. The first time is when the computer scans the instructions to detect specific types of errors before the program actually begins execution. The second time is during program execution. Errors detected at either time cause program execution to terminate.

For example, the following program has two errors in it.

```
100 PROGRAM example;
110 BEGIN
120   WRITELN("writeln is an output statement');
130 END
```

Lines 100, 110, and 130 can be entered and stored in memory. However, when you try to enter line 120, the error indicator appears and the error message `Illegal character in text` is displayed. To correct the line, press **SHIFT 9** to display it, and then change the quotation mark to an apostrophe.

When you run the program, the message `Illegal nesting` is displayed. Press → to display the error code and the line number of the erroneous line. In this case, the error code and line number are E27 L130.

To display the line specified in the error message, press ↑ or ↓. Use the edit keys to place the period after the word END and enter the corrected line. The program will then run and terminate correctly.

Occasionally the line displayed as causing an error may not be the source of the problem. Values generated or actions taken elsewhere in the program may cause the error. The line number displayed is the line where the interpreter detected an error. For example, enter the following program.

```
100 PROGRAM example
110 BEGIN
120   WRITELN(' writeln is an output statement');
130 END.
```

When you run the program the error message `';' expected` is displayed. When you press the → key, the error code and line number are `E14 L110`. Press ↑ or ↓ to display line 110. Line 110 does not have an error in it, but line 100 does. After the interpreter scanned line 100, it moved to line 110, expecting to find a semicolon to separate the statements on lines 100 and 110. Therefore line 110 is displayed as the erroneous line because the interpreter detected a missing semicolon during its scan of line 110. Press ↑ to display line 100 and enter a semicolon after the word `example`.

Note that if an error code is preceded by a `W` rather than an `E`, the message displayed was a warning and not an error. Program execution continues after a warning when the **ENTER** key is pressed. Remember that the line number displayed in an error (or warning) message is an indication of where the interpreter detected the error (or warning).

Refer to appendix I in the *TI-74 Learn Pascal Reference Guide* for a list of the error codes and messages.

Debugging a Program

When a program does not work the way you intended, there are logical errors in it (called "bugs" in computer usage). Testing a program to find these bugs is called "debugging" a program. When a program does not work properly, think about what could be wrong, then devise tests such as displaying values throughout the program to aid you in finding the bugs.

The BREAK command can be used to stop a program at specific lines and allow you to determine what is happening in the program. When a program stops at a breakpoint, you can display values in the program.

For example, breakpoints can be set at lines 120 and 130 in the program

```
100 PROGRAM example2;
110 BEGIN
120   WRITE(2+5,' is the answer');
130   WRITE(' for number 10');
140   WRITELN;
150 END. (*example2*)
```

by entering the BREAK command before the RUN command as shown below.

BREAK 120,130
RUN

After the RUN command is entered, the breakpoint at line 120 causes the message `Break` to be displayed.

Press the **CLR** or **ENTER** keys to erase the message and you can then perform any imperative statement. Enter **CON** to resume program execution. The program then displays the message `7 is the answer`

from the WRITE at line 120. Press the **ENTER** key and the breakpoint at line 130 stops the program and displays the `Break` message. Press the **ENTER** or **CLR** keys to erase the message and enter **CON** to resume program execution. The message

```
 for number 10
```

is then displayed. Press the **ENTER** key to proceed to the WRITELN in line 140. The message remains in the display until the **ENTER** key is pressed again.

Note that for statements entered on multiple lines, the breakpoint occurs at the beginning of the first executed statement on or after the specified line. Breakpoints entered in a program continue to stop program execution until you use the UNBREAK command to delete the breakpoints.

Review Chapter 3

1. Every Pascal program must contain two parts. These parts are ________________ and ________________.

2. A Pascal program body is enclosed between the reserved words ________________ and ________________.

3. A program block consists of declarations and ________________.

4. Declarations are used to ________________ names.

5. Declarations that are used in a program must appear in what order?

   ____

   ____

   ____

   ____

   followed by

   ________________ ________________ .

6. Each word in Pascal is called an ________________.

7. Which of the following user-defined identifiers are valid?

   measure

   5percent

   printheader

   END

account1

sales-tx

8. Name the four types of constants.

____

____

____

____

9. In Pascal, a number written with a decimal point is called a __________ number.

10. Integers greater than ________________ or less than ______ cannot be entered from the keyboard

11. An apostrophe within a character string is represented as __________ ____________________

12. What is wrong with the following comment?

```
(   *Two-character symbols cannot have a
 space between them. *)
```

13. The computer supplies line numbers for you when you enter the __________ command and renumbers the program lines when you enter the __________ command.

14. The maximum length of a line is __________ characters.

15. What is the error in the following line?

```
100 PROGRAM example; BEGIN WRITELN
('One line') WRITELN('end'); END.
```

16. Comments are enclosed in the symbols __________ and __________ or __________ and __________ .

17. What does the following interpreter option do?

```
150 BEGIN {$w-}
```

18. Write the output produced by the following program segments.

```
150 WRITE('The answer is ');
160 WRITELN(10);
```

____

```
170 WRITELN('The answer is ');
180 WRITELN(10);
```

____

19. Write a program that displays the following.

```
5+5 is 10
```

20. Write a program that displays the following.

```
***The results are listed below***
    x=5
    y=10
```

21. The three valid methods of terminating program execution are

____

____

____

22. An imperative must fit on a single line and must end with a ______________________________.

23. Find two errors in the following program.

```
100 PROGRAM example;
110 WRITELN('writeln is an output statement');
120 END
```

Chapter 4—Expressions

Introduction

Expressions are the calculations that you assemble in a program for the computer to perform. Before you can write expressions, you must be familiar with the elements of expressions and the rules for combining them. These elements—constants, variables, operators, and functions—are described in the following sections.

Constant Declarations

A constant declaration is used to define the value of a user-defined identifier as a numeric, string, character, or Boolean constant. The value of a constant identifier cannot be altered during program execution.

A constant declaration in its simplest form is

CONST *identifier=value*;

where CONST informs the interpreter that the specified *identifier* has the *value* of the indicated numeric, character, string, or Boolean constant. For example, the identifier salestax can be defined as the number 0.05 by including it in a CONST declaration.

```
CONST salestax=0.05;
```

You can declare several constants in a program. Note, however, that the reserved word CONST can appear only once in a declaration section. In the lines

```
CONST salestax=0.05;
      heading='sales tax';
      age=21;
      grade='A';
      flag=TRUE;
```

five constants are defined for use in a program.

It is good programming to declare a number or a string of characters that is used more than once in a program as a constant. Then if the value of the constant has to be changed, you need to edit only the CONST declaration, thus reducing the chance for error.

In the following program, which prints the circumference of circles with diameters of 2 cm, 9 in., and 3 m, a constant declaration is used to define the value of the constant identifier pi with the value of π (3.14159265359).

```
100 PROGRAM circum;
110 CONST pi=3.14159265359;
120 BEGIN
130   WRITELN('Circum. of 2 cm: ',pi*2,' cm');
140   WRITELN('Circum. of 9 in: ',pi*9,' in');
150   WRITELN('Circum. of 3 m: ',pi*3,' m');
160 END.
```

If you run the program, the following output is displayed.

```
Circum. of 2 cm: 6.283185307 cm

Circum. of 9 in: 28.27433388 in

Circum. of 3 m: 9.424777961 m
```

Variable Declarations

When only constants are used in a program, a program can become very long if it has to perform many computations. If the values of items can change in a program, the program is much easier to write and much more useful.

Variables are used when the values of items in a program vary or change. A variable is a name given to a memory location in the computer. You can store a value in the location and then change it in the program as many times as needed.

Before you can use a variable in a Pascal program, you must define it in a VAR (for variable) declaration. VARiable declarations must appear after any CONSTant declarations. You can define as many variables as you need in a program; however, the reserved word VAR can appear only once in a declaration section.

A variable declaration in its simplest form is

VAR *identifier* : *type* ;

where VAR informs the interpreter that a variable with a name of *identifier* is being declared with a specified *type*. A variable's *type* determines how the variable can be used in a program. There are five fundamental types in Pascal that are used to form expressions. These five types are listed below.

- INTEGER
- REAL
- CHAR
- STRING
- BOOLEAN

The following VAR declaration defines some variables and their types.

```
VAR lenth,width,height:REAL;
    counter,index:INTEGER;
    payment:REAL;
    name:STRING;
    grade:CHAR;
    test:BOOLEAN;
```

A program with the above VAR declaration in it can use the following variables.

- lenth, width, height, and payment will contain REAL values
- counter and index will contain INTEGER values
- name will contain a character string
- grade will contain a single character
- test will contain a BOOLEAN value

A program cannot use a variable that has not been declared in a VAR declaration. Each time a variable is referenced, the computer verifies that the variable is used in the program as it was declared. If the variable is used improperly, an error message is displayed. For example, CHAR variables cannot be multiplied and INTEGER variables cannot have REAL values.

Remember that although the **value** of a variable may be changed at any time in a program, the **type** of a variable **cannot** be changed.

INTEGER Type

Integers are the natural counting numbers, their negatives, and the number zero. The maximum integer allowed in TI-74 Pascal, called MAXINT, is 32767; the smallest integer allowed is −32767. Note that −32768 is allowed in computations in the computer, but cannot be entered from the keyboard.

The following are valid integers in Pascal.

```
39        +40
0         MAXINT
-543
```

Some invalid integers are shown below.

5,280	no comma allowed
3.14	no decimal point allowed
40394	value too large for an integer

REAL Type

Real numbers in Pascal correspond to the decimal numbers or floating-point numbers. Real numbers in the TI-74 can have a magnitude as small as ±1.0E−128 or as large as ±9.9999999999999E+127. Only 10 digits of a real constant are displayed when a program is running, but all 13 or 14 digits are used in calculations and are displayed when a program is listed.

In Pascal, a real number **must** have the following:

- a decimal point
- at least one digit to the left of the decimal point
- at least one digit to the right of the decimal point

The following are examples of real values in Pascal.

+345.0	2.236456
−345.0	0.0
0.1	40394.0

Examples of invalid real values are shown below.

+345	no decimal point and no digit to the right of the decimal point
−345	no decimal point and no digit to the right of the decimal point
.1	digit missing to the left of the decimal point
2.236456−	minus sign must precede the number
0	no decimal point and no digit to the right of the decimal point
40,394.0	no commas allowed in numbers

Note that an integer value can be used in computations for a real value because the interpreter can convert an integer to its real equivalent. For example, if the integer 7 is used in computations with real values, the interpreter converts it to 7.0.

Real numbers can also be written in scientific notation. A real value is automatically displayed in scientific notation when its magnitude is 9999999999.49995 or greater. In scientific notation, a number is expressed in a format in which a number (the mantissa) is multiplied by 10 raised to a power (the exponent).

For example, the number 12345678 can be expressed in scientific notation as 1.2345678E+7, which represents 1.2345678×10^7. The number 0.00000075 is expressed in scientific notation as 7.5E – 7, which represents 7.5×10^{-7}. In Pascal, numbers represented in scientific notation cannot be written with spaces in them. Therefore, the number 7.5 E – 7 must be written without a space as 7.5E – 7.

When a number is displayed in scientific notation, the computer displays a maximum of nine digits. If the exponent has two digits, the mantissa is limited to seven digits. When the exponent has three digits, the mantissa is limited to six digits.

Some examples of real numbers in scientific notation are shown below.

1.717172E + 7   1.2E – 5

CHAR Type

A character is any symbol that is in the TI-74 character set. In Pascal, character constants are enclosed in apostrophes.

Some examples of characters are shown below.

'/'	'N'
'?'	'B'
']'	'#'
'%'	'4'

'''' (The character constant apostrophe must be represented as two apostrophes inside the enclosing apostrophes.)

STRING Type

A string is a sequence of characters enclosed in apostrophes.

The following are valid string constants.

'Pascal language'
'the cat''s meow' (embedded single quotes are typed twice)
(the empty string)

You can specify the maximum length of a string variable by following the reserved word STRING with an integer constant enclosed in brackets. This integer must be from 1 through 255. If you do not specify the length of a string variable, a default value of 80 characters is used. A string cannot contain more characters than its specified (or default) length. A string with no characters is called a null string and has a length of zero.

Examples of valid string variables are shown below.

```
VAR heading:STRING;
(* by default, maximum length is 80 *)
    grafline:STRING[200]; (* max length 200 *)
```

A string's maximum length specifies the maximum length that the string can be throughout the program. The dynamic length of a string is its actual length and is equal to the number of characters that are currently in the string. The dynamic length of a string may change during the execution of a program, but it may never be greater than the string's maximum length.

The characters of a string are numbered from left to right beginning with 1 and continuing to the last character currently in the string. This numbering system is called indexing. To access a character in a string, write the name of the string followed by the character's index enclosed in brackets. For example, if the string variables `proglang` and `version` contain the characters "PASCAL" and "TI", respectively,

`proglang[1]` contains "P"
`proglang[5]` contains "A"
`version[1]` contains "T"

Note that if a string is indexed past the last character currently in it, an error occurs. The empty string cannot be indexed.

A STRING data type contains zero or more characters. Note, however, that when an identifier is defined in a CONST declaration,

- an identifier defined with zero or more than one character is considered a STRING constant.

- an identifier defined with one character is considered a CHAR constant and must be used accordingly.

For example, suppose the following declarations are entered in a program.

```
CONST strname1='abc';
      strname2='';
      strname3='a';
VAR   strdata:STRING;
```

The value of strname1 and the value of strname2 can be stored in the variable strdata (which is a STRING type). However the value of strname3 cannot be stored in strdata because strname3 has exactly one character, making its type CHAR rather than STRING.

BOOLEAN Type

Boolean data types have a value of TRUE or FALSE. These values, TRUE and FALSE, are called predefined constants in Pascal. Note that when Boolean types are compared, FALSE is defined to be less than TRUE.

Setting the Values of Variables

Initially, all the variables declared in a Pascal program are undefined. To give a value to a variable, you must store a value in the memory location reserved for that variable by using an assignment statement or an input statement.

Assignment Statements

The assignment statement is used to store values in variables. An assignment statement uses the symbol := that is called the assignment operator. The := symbol should be read as ''becomes equal to'' and should not be interpreted as an ordinary equals sign. The equals sign by itself cannot be used as an assignment operator.

`lenth:=10.5;`	Stores the real value 10.5 in the location called `lenth`
`counter:=25;`	Stores the integer value 25 in the location called `counter`
`name:='Brian';`	Stores the string ''Brian'' in the location called `name`
`grade:='A';`	Stores the character ''A'' in the location called `grade`
`test:=TRUE;`	Stores the Boolean value TRUE in the location called `test`

The left side of the assignment statement is the name of the location in memory where the value on the right side is stored. The program

```
100 PROGRAM exvar;
110 VAR a,b:INTEGER;
120 BEGIN
130    a:=3;
140    b:=5;
150    WRITELN(' The values of the variables
       are' ,a,' and ',b);
160 END. (* exvar *)
```

stores the values 3 and 5 in a and b, respectively, as shown in the output below.

```
The values of the variables are 3 and 5
```

The right side of an assignment statement is always executed first, regardless of what variable appears on the left. In the statement,

```
x:=z;
```

the computer determines the value stored in the location named z and stores the same value in the location called x. Both x and z then have the same value. The previous contents of x are destroyed. The program

```
100 PROGRAM exvar;
110 VAR a,b:INTEGER;
120 BEGIN
130    a:=35;
140    b:=23;
150    a:=b;
160    b:=a;
170    WRITELN('The values of a and b are ',
       a,' and ',b);
180 END. (* exvar *)
```

produces the output shown below.

```
The values of a and b are 23 and 23
```

In an assignment statement, the value being assigned to a variable must be of the same data type as the variable. Note, however, that integer values can be assigned to real variables.

The following program defines a constant called pi that is a REAL value. The variables a and b are INTEGER; the variables c and d are REAL. When c is assigned a value, the computer finds the integer value stored in a, converts it to its real equivalent, multiplies it by pi, and stores the real result in c. A similar process is used to assign a value to d.

```
100 PROGRAM exvar;
110 CONST pi=3.14159265359;
120 VAR a,b:INTEGER;
130         c,d:REAL;
```

```
140 BEGIN
150    a:=3;
160    b:=5;
170    c:=a*pi;
180    d:=b*pi;
190    WRITELN(a,'   ',c);
200    WRITELN(b,'   ',d);
210 END. (* exvar *)
```

The output from the program is shown below.

```
3   9.424777961

5   15.70796327
```

Input Statements

In Pascal, the READ and READLN statements are used to input data. An input statement is used to store in a variable a value entered from the keyboard or read from a file. An input statement contains (in parentheses) one or more variables that are to be assigned a value or values. A variable included in an input statement must be an INTEGER, REAL, CHAR, or STRING type; the value entered must be a valid data type for that variable type.

For example, when the input statement

```
160 READ(x);
```

is performed, the computer waits until a value is entered from the keyboard. When a value is entered, it is stored in the variable x.

In the following program, the integer value entered from the keyboard is stored in the INTEGER variable a and then displayed. The next value entered can be either an integer or real value because it is stored in the REAL variable x.

```
100 PROGRAM exinput;
110 VAR a:INTEGER;
120     x:REAL;
130 BEGIN
140    READ(a);
150    WRITELN('   a= ',a);
160    READ(x);
170    WRITELN('   x= ',x);
180 END. (* exinput *)
```

If **12** is entered for a, the display contains the following after the first WRITELN is executed.

```
a= 12
```

If **7** is entered for x, the display contains the following after the second WRITELN is executed.

```
x= 7.0
```

If an input statement contains two or more variables, they must be separated by commas. If x, y, and z have been declared INTEGER variables and the statement

```
160 READ(x,y,z);
```

is executed, the computer waits until three integer values have been entered from the keyboard. Each value entered is stored in the specified variable as soon as it is entered. When more than one value is being entered, the values can be separated by one or more spaces or can be entered on different lines.

For example, the following program accepts an integer value for a, a real value for x, and an integer value for b. The values can be entered on one, two, or three lines. If more than one value is entered on a line, the values must be separated by at least one space.

```
100 PROGRAM exinput;
110 VAR a,b:INTEGER;
120         x:REAL;
130 BEGIN
140   READLN(a,x,b);
150   WRITELN(' a= ',a,'   x= ',x,'   b= ',b);
160 END. (* exinput *)
```

If the program is run and the values **71**, **7.12**, and **40** are entered, the program displays the output shown below.

```
a= 71   x= 7.12   b= 40
```

When the computer is reading data to assign to an INTEGER or a REAL type variable, all leading blanks and ends of lines are skipped until a nonblank character is reached. If the nonblank character is not a sign or a digit, an error occurs.

All characters after the nonblank character are read until a nonnumeric character is reached. For an INTEGER type, the nonnumeric character causes the computer to stop reading characters for that integer variable. For a REAL type, all characters after the nonblank are read until a nonnumeric character is reached that is not a valid character for a number expressed in either decimal or scientific notation.

The value 39.0 may be entered as

39.0 (decimal notation)

or

+3.9E+01 (scientific notation)

or

39 (an integer, which may be entered for a real variable)

For example, the following program accepts three REAL values from the keyboard and assigns the values to x, y, and z.

```
100 PROGRAM exread;
110 VAR x,y,z:REAL;
120 BEGIN
130   READLN(x,y,z);
140   WRITELN(' x= ',x,'    y= ',y,'    z= ',z);
150 END. (* exread *)
```

If you enter the input line shown below, the program displays the characters shown in the output line.

Input: **39.0** **+3.9E+01** **39**

Output:
```
x= 39.0   y= 39.0   z= 39.0
```

For a STRING type, all characters are read up to the end-of-line (**ENTER**) character. For a CHAR type, the character pointed to by the cursor is stored in the variable and the cursor advances one column. If the character just read is the last one on the line, the cursor then points to the end-of-line (**ENTER**) character. If the end-of-line character is read, a space is stored in the CHAR variable and the cursor moves to the first character on the next line.

The READLN Statement

The READLN statement stores values in its variables, ignores any other characters to the end of the line, and then moves past the end-of-line character to position the cursor to the first character in the next line. For example, the program

```
PROGRAM exreadln;
VAR x:INTEGER;
     y,z:REAL;
BEGIN
  READLN(x,y);
  READLN(z);
  WRITELN(x);
  WRITELN(y);
  WRITELN(z);
END. (* exreadln *)
```

reads and displays one integer and two real values. If the values shown in the input line are entered, the program displays the values shown in the output line.

```
Input:     12          7.1239e3    45.5
           12.739490

Output:    12          7123.9
           12.73949
```

Note that the number 45.5 is ignored by the READLN statement because there is no variable to assign it to as the READLN statement moves past the end of the line to the first character in the next line.

When you execute a READLN statement with no variables in parentheses, no data is read and the input cursor is moved to the first character in the next line. For example, the statement

```
READLN;
```

moves past the end-of-line marker and positions the cursor at the first character in the next line.

The READ Statement

The READ statement allows the next input statement (READ or READLN) to get values from the same line. A READ statement also reads to the end-of-line character, but it does not ignore characters as it moves to the end of the line. All of the characters are retained in an input buffer for the next input statement(s).

If the first READLN statement in the previous example is changed to a READ statement, the input line can be entered on one line, as shown below.

```
100 PROGRAM exread;
110 VAR x:INTEGER;
120       y,z:REAL;
130 BEGIN
140    READ(x,y);
150    READLN(z);
160    WRITELN(x);
170    WRITELN(y);
180    WRITELN(z);
190 END. (* exread *)
```

```
Input:    12          7.1239e3   12.739490  45.5
Output:   12          7123.9     12.73949
```

Note that the value 45.5 is again ignored by the READLN statement. If the READLN statement were a READ statement, however, this value would be retained in an input buffer for the next input statement. The output would begin where the READ statement left the cursor.

The extra values placed in an input buffer by READ are retained there for the next input statement. These values are assigned according to the following rules.

• If the next input statement is another READ statement, the variables in this READ statement are assigned values from the extra values. If any values still remain unassigned, they are retained until the next input statement is encountered.

• If the next input statement is a READLN statement, the variables in the READLN statement are assigned values from the extra values. If any values still remain unassigned, they are discarded.

Only one string can be read in an input statement because the characters in a string include every character from the beginning of the string up to the end-of-line character. Therefore, the statements

```
150 READLN(string1, string2);
```

and

```
160 READ(string1);
170 READ(string2);
```

result in string2 being a null string.

To read the two strings, string1 and string2, two READLN statements should be executed as shown below.

```
READLN(string1);
READLN(string2);
```

The following program is an illustration of how the READ and READLN statements read entered data. Two integers are read and then a character is read. Two real values are then read followed by a string. If the data shown in the input line is entered, the results (or output) displayed are those shown in the output line.

```
100 PROGRAM getdata;
110 VAR m,n:INTEGER;
120     x,y:REAL;
130     a:CHAR;
140     st:STRING;
150 BEGIN
160   READ(m,n);
170   READ(a);
180   READ(x,y);
190   READLN(st);
200   WRITELN(m);
210   WRITELN(n);
220   WRITELN(x);
230   WRITELN(y);
240   WRITELN(a);
250   WRITELN(st);
260 END. (* getdata *)
```

```
Input:    12   7A   40.5   39.4        'This is a test'
Output:   12   7    40.5   39.4
          A    'This is a test'
```

Using Prompts for Input
A program can use an output statement to display a message that prompts for input. For example, if the **y** or **n** key should be pressed to continue or stop program execution, a program could include a prompt for the character as shown below. Note that the wait interpreter option is turned off after the WRITE statement but before its semicolon. Otherwise, the

prompt would be displayed with the wait option implemented. Then the **ENTER** or **CLR** key would have to be pressed before the computer could accept data.

After the data is read, the wait option must be turned back on so that the results of the WRITELN can be read in the display.

```
100 PROGRAM exprompt;
110 VAR ch:STRING;
120 BEGIN
130   WRITE(' Continue? (y or n)') {$w-};
140   READ(ch) {$w+};
150   WRITELN(ch);
160 END. (* exprompt *)
```

Operators

With each data type, specific operations can be performed by using a special symbol, called an operator, with the data. Unary operators process one quantity (called an operand); binary operators process two operands.

There are three different kinds of operators.

arithmetic operators	perform arithmetic processes such as addition and subtraction on operands.
relational operators	compare two operands.
logical operators	perform logical tests on the true/false values of operands.

The arithmetic, relational, and logical operators that can be used with each data type are discussed in the following sections.

INTEGER Data Operators

The following operators can be used with INTEGER data.

Arithmetic Operators
There are two unary operators (+ and −) and six binary operators (+, −, *, /, DIV, and MOD).

Unary operators

+	keeps the sign of the operand following it.
−	changes the sign of the operand following it.

Binary operators

+	computes the sum of the left and right operands.
–	computes the difference between the left and right operands.
*	computes the product of the left and right operands.
/	computes the quotient of the left operand divided by the right operand. The result is a REAL value.
DIV	computes the quotient of the left operand divided by the right operand and truncates the result (drops any digits to the right of the decimal point). DIV returns an integer.
MOD	computes the quotient of the left operand divided by the right operand and returns only the remainder. MOD returns an integer.

Examples of using arithmetic operators with integer data are shown below.

Operation	**Result**	**Comments**
–(–3)	3	changes sign of the operand
40+7	47	
12–7	5	
5*6	30	
5/2	2.5	returns a REAL result
25 DIV 3	8	25/3 is 8.333; the integer portion of the quotient is 8
–25 DIV 3	–8	–25/3 is –8.333; the integer portion of the quotient is –8
25 MOD 3	1	25/3 is 8, with a remainder of 1
–25 MOD 3	–1	–25/3 is –8, with a remainder of –1
7 MOD 7	0	7/7 is 1, with a remainder of 0

Relational Operators
Seven relational operators can be used with integer data. A relational operator returns a value of TRUE or FALSE, based on the comparison.

>	(greater than)	returns a TRUE result if the left operand is greater than the right operand. Otherwise, FALSE is returned.
>=	(greater than or equal to)	returns a TRUE result if the left operand is greater than or equal to the right operand. Otherwise, FALSE is returned.
<	(less than)	returns a TRUE result if the left operand is less than the right operand. Otherwise, FALSE is returned.
<=	(less than or equal to)	returns a TRUE result if the left operand is less than or equal to the right operand. Otherwise, FALSE is returned.
=	(equal to)	returns a TRUE result if the left and right operands are equal. Otherwise, FALSE is returned.
<>	(not equal to)	returns a TRUE result if the left and right operands are not equal. Otherwise, FALSE is returned.
IN	(set membership)	returns a TRUE result if the left operand is an element of the right operand. The right operand must be a set of values enclosed in brackets. If the left operand is not a member of the right operand, FALSE is returned.

Examples of using the seven relational operators are shown on the next page.

Operation	Result	Comments
4>3	TRUE	
4>5	FALSE	
4>=4	TRUE	
4>=5	FALSE	
4<5	TRUE	
4<3	FALSE	
4<=4	TRUE	
4<=3	FALSE	
4=4	TRUE	
4=5	FALSE	
4<>5	TRUE	
4<>4	FALSE	
4 IN[4,5,6]	TRUE	The integer 4 is in the set shown in brackets.
4 IN[1,2,3]	FALSE	The integer 4 is not in the set.
4 IN[−5..5]	TRUE	The integer 4 is in the set of integers from −5 through 5.

Logical Operators
Logical operators cannot be used with INTEGER data.

REAL Data Operators

The following operators can be used with REAL data.

Arithmetic Operators
There are two unary and four binary operators that can be used with REAL data.

Unary operators

+	keeps the sign of the operand following it.
–	changes the sign of the operand following it.

Binary operators

+	computes the sum of the left and right operands.
–	computes the difference between the left and right operands.
*	computes the product of the left and right operands.
/	computes the quotient of the left operand divided by the right operand. DIV may not be used with REAL numbers.

Examples of arithmetic operations are shown below.

Operation	Result	Comments
1.2+0.7	1.9	
5.2–5.3	–0.1	
5.0 * 3.1	15.5	
2.2 / 2.0	1.1	At least one digit must be to the right of the decimal point in the number 2.0.
5.345 / 0.5	10.69	At least one digit must be to the left of the decimal point in the number 0.5.
7 / 2	3.5	Division of INTEGERS results in a REAL value.

Relational Operators
Seven relational operators can be used with CHAR data. The comparisons of the operands are performed using the ASCII codes of the characters. Refer to appendix G in the *TI-74 Learn Pascal Reference Guide* for a list of the ASCII codes.

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
=	equal to
<>	not equal to

The following are examples of relational operations on REAL data.

Operation	Results
5.5>5.1	TRUE
5.5>=5.5	TRUE
5.5<5.1	FALSE
5.5<=5.1	FALSE
5.5=5.1	FALSE
5.5<>5.5	FALSE

Logical Operators
Logical operators cannot be used with REAL data.

Character Data Operators

Arithmetic operators cannot be used with character data, but the following operators can be used.

Relational Operators
Seven relational operators can be used with CHAR data. The comparisons of the operands are performed using the ASCII codes of the characters. Refer to Appendix G in the *TI-74 Learn Pascal Reference Guide* for a list of the ASCII codes.

Some examples of using relational operators with CHAR data are shown on the next page.

Operation	Result	Comments
'a'<'b'	TRUE	The ASCII code of a (97) is less than the ASCII code of b (98).
'a'<='c'	TRUE	The ASCII code of a (97) is less than the ASCII code of c (99).
'A'>'d'	FALSE	The ASCII code of A (65) is not greater than the ASCII code of d (100).
'A'>='%'	TRUE	The ASCII code of A (65) is greater than the ASCII code of % (37).
'A' = 'G'	FALSE	The ASCII code of A (65) is not equal to the ASCII code of G (71).
'A'<>'a'	TRUE	The ASCII code of A (65) is not equal to the ASCII code of a (97).
'a' IN['a','b','c','d']	TRUE	The character a is in the set of specified values.

Logical Operators
Logical operators cannot be used with CHAR data.

STRING Data Operators

Arithmetic operators cannot be used with STRING data, but the following operators can be used.

Relational Operators
Six relational operators (<, <=, >, >=, =, and <>) can be used with STRING data to compare the ASCII values of the characters in the strings. The ordering of strings is alphabetical (lexicographical); uppercase precedes lowercase. A shorter string precedes a longer string if the characters in the shorter string are the same as the characters in the beginning of the longer string.

Some examples of string comparisons are shown on the next page.

Operation	Results	Comments
'Pascal - IV' = 'Pascal - TI'	FALSE	Strings are not the same
'Pascal - IV'<>'Pascal - iv'	TRUE	Uppercase and lowercase letters do not compare equal
'Pascal - IV'<'Pascal - TI'	TRUE	Lexicographically I comes before T
'Pascal - IV'<= 'Pascal'	FALSE	A longer string compares greater than a shorter string
'Pascal - IV'>'Pascal - 4.0'	TRUE	Letters have a higher ASCII code than numbers
'Pascal - TI'>= 'Pascal - TI'	TRUE	Strings are the same

Logical Operators
Logical operators cannot be used with STRING data.

BOOLEAN Data Operators

Arithmetic operators cannot be used with BOOLEAN data, but the following operators can be used.

Relational Operators
Seven relational operators (<, <=, >, >=, =, <>, and IN) can be used with BOOLEAN values. FALSE is defined to have the value 0, whereas TRUE is defined to have a value of 1. Therefore, by definition, FALSE<TRUE.

<	less than	returns a TRUE result if the left operand is FALSE and right operand is TRUE. Otherwise, a FALSE value is returned.
<=	less than or equal	returns a TRUE result if the right operand is TRUE or if the left operand is FALSE. Otherwise, a FALSE value is returned.

>	greater than	returns a TRUE result if the left operand is TRUE and the right operand is FALSE. Otherwise, a FALSE value is returned.
>=	greater than or equal	returns a TRUE result if the left operand is TRUE or if the right operand is FALSE. Otherwise, a FALSE value is returned.
=	equal to	returns a TRUE result if the left and right operands are both TRUE or are both FALSE. Otherwise, a value of FALSE is returned.
<>	not equal to	returns a TRUE result if the left and right operands are not the same. Otherwise, a value of FALSE is returned.
IN	member of	returns a TRUE result if the left operand is an element of the right operand (a set). Otherwise, a value of FALSE is returned.

Logical Operators
There are three logical operators that can be used with BOOLEAN data types, AND, OR, and NOT.

AND	returns a value of TRUE if the left operand and the right operand are TRUE. Otherwise, a value of FALSE is returned.
OR	returns a value of TRUE if either the left or right operand is TRUE or if both are TRUE. If both operands are FALSE, a value of FALSE is returned.
NOT	returns the negation of the operand following it. A value of TRUE is returned if the operand following it is FALSE; a value of FALSE is returned if the operand is TRUE.

The results of using the BOOLEAN operators for all cases are given on the next page.

Operation	Results
TRUE AND TRUE	TRUE
TRUE AND FALSE	FALSE
FALSE AND TRUE	FALSE
FALSE AND FALSE	FALSE
TRUE OR TRUE	TRUE
TRUE OR FALSE	TRUE
FALSE OR TRUE	TRUE
FALSE OR FALSE	FALSE
NOT TRUE	FALSE
NOT FALSE	TRUE

Examples of using the logical operators with BOOLEAN data are given below. Note that even though the operands themselves use relational operators with integer values, these operands have a BOOLEAN value of TRUE or FALSE.

Operation	Results	Comments
(3<4) AND (5<7)	TRUE	both operands are TRUE (3 is less than 4 and 5 is less than 7).
(3<4) AND (5>7)	FALSE	one of the operands (5>7) is not TRUE.
(3<4) OR (5<7)	TRUE	at least one of the operands is TRUE.
(3>4) OR (5>7)	FALSE	neither operand is TRUE.
NOT (3<4)	FALSE	the operand is TRUE and NOT TRUE is FALSE.
NOT (3>4)	TRUE	the operand is FALSE and NOT FALSE is TRUE.

Operator Precedence

When an expression is evaluated, an ambiguity may arise when there is a sequence of operators.

For example

```
temp:=20-4*3
```

The instruction could be interpreted as

```
temp:=20-4*3
    :=16 *3
    :=    48
```

Or as

```
temp:=20-4*3
    :=20- 12
    :=       8
```

For you to know what results you will always obtain in an expression, the order in which operations are performed has been defined in programming languages. In Pascal, the following order of precedence has been established.

()	any calculation within parentheses is computed first.
NOT	is performed next.
* / MOD DIV AND	are performed next.
+ − OR	are performed next.
= <> <= >= <> IN	are performed last.

If two operators of the same priority appear in an expression, they are evaluated in left-to-right order.

In the previous example, the * (multiplication) is always performed before + (addition). Therefore, `temp:=20-4*3` is evaluated as shown below.

```
20-4*3
20-12
8
```

Note that the order of precedence for operations in Pascal differs from some programming languages. Because the logical operator AND is performed before any relational operation, a relational expression on either side of AND must be enclosed in parentheses.

(5<6) AND (2<3)
TRUE AND TRUE which is TRUE.

To subtract the sum of two numbers from another number, you must use parentheses to override the left-to-right order of precedence. For example, to subtract the sum of 39 and 7 from 40, the expression must be written as

40 - (39 + 7)

that is evaluated as

40 - 46
-6

If the expression is written as

40 - 39 + 7

the expression is evaluated from left to right, 40 - 39 + 7 = 1 + 7 = 8, because - and + have the same level of precedence.

The following program accepts temperatures in degrees Fahrenheit and converts them to degrees Celsius. Note that the expression contains terms in parentheses that are operated on first. The division and multiplication are performed from left to right.

```
100 PROGRAM tcelsius;
110 VAR fahdeg:REAL;
120 BEGIN
130   WRITE('Enter deg: ') {$w-};
140   READLN(fahdeg) {$w+};
150   WRITELN(fahdeg,'deg F. = ',
      (fahdeg-32)*5/9,' deg C.')
160 END. (* tcelsius *)
```

Input: **98.6**
Output: `98.6 deg F.= 37.0 deg C.`

Forming Expressions

You can form expressions by combining constants, variables, and functions (described in the next section) with any operators that are valid for the type of constant, variable, and function you are using. The following conventions apply to expressions.

- An expression can be a single constant or variable, which may be preceded by a unary plus or minus.
- An expression can be a sequence of variables, constants, and/or functions separated by operators. The variables, constants, and functions are called terms of the expression.
- Two operators cannot be adjacent to each other. Parentheses must be used to separate operators.

  For example, to multiply 12 by a – 7, you must write 12 * (– 7). In Pascal, 12* – 7 is not allowed.
- A function may replace any variable or constant.

In Pascal, all of the constants, variables, and values of functions used in an expression must be of the same type. For example, all variables, constants, and results of functions in an integer expression must be integers. All variables, constants, and results of functions in a real expression should be real. Note, however, that if an integer value is used in a real expression, it is converted into a real type.

If the variable `count` is declared to be INTEGER, the examples

```
count + 1                    count DIV 5
count + 10                   count MOD 4
count * 10
```

are illustrations of valid integer expressions.

Some invalid integer expressions are shown below.

count + 1.0	an integer and a real added together produce a real result
count/5	/ is the symbol of division for real data and the result is a real value
3* – 4	two operators together

If the variables average and rate are declared to be REAL, the following examples are illustrations of valid real expressions.

average + 5.0	
average + 10	the integer 10 is converted to a real number
average/3.0	
average*rate	

Some examples of invalid real expressions are shown below.

average + -3	two operators together
average DIV 5	DIV can be used only with integer operands
average/.5	no digit before the decimal point

Because of the way numbers are stored internally in the TI-74, operations performed on real numbers may not yield an exact value. For example, the fraction 1/3 is represented by a finite number of decimal digits, 0.33333333333333. After many operations are performed on a real value, the error due to truncation or to the rounding off of the result can become large in some cases.

In most cases, the approximation of the result is insignificant. However, you should not test real values for equality; instead, test that the difference between two real values is less than a specified amount. For more information on numerical accuracy, refer to appendix H in the *TI-74 Learn Pascal Reference Guide*.

Functions

A function is a specialized routine that performs a computation and returns a value. In Pascal there are both standard functions and user-defined functions. User-defined functions are discussed later in chapter 7.

The standard functions available in Pascal are represented by a standard identifier usually followed by an operand (called an argument) enclosed in parentheses.

Some functions require that an argument be an expression of a specific data type; other functions use arguments that can be expressions of any data type. Some functions require an argument be an expression that is ordinal, a category that includes INTEGER, CHAR, and BOOLEAN types.

In this manual, the data type of an argument is represented as shown on the next page.

integer-expression	INTEGER expression
real-expression	REAL expression
iorr-expression	INTEGER or REAL expression
string-expression	STRING expression
char-expression	CHAR expression
bool-expression	BOOLEAN expression
multi-expression	multiple types of expressions

You use functions like variables in a program except that you cannot place a function name on the left side of an assignment operator (: =). When a statement containing a function name is executed, the value of the function is returned and used in place of the function name. The argument of a function does not always have to be the same data type as the value it returns.

Because each value in Pascal has a specified data type, a function must be used with the same data types as the value it returns. In this manual, the functions are grouped by the type of the value they return.

A function cannot be used alone as an imperative; it can, however, be part of an imperative (such as a WRITELN statement).

Integer Functions

The following standard functions, classified as numeric, memory, string, and ranking, return an INTEGER value.

Numeric
The numeric functions operate on numeric values.

ABS(*integer-expression*)	returns the absolute value of the *integer-expression*.
SQR(*integer-expression*)	returns the square of the *integer-expression*.
TRUNC(*real-expression*)	returns the integer portion of the *real-expression*.
ROUND(*real-expression*)	returns the integer that is nearest the *real-expression*. If the fractional part of the expression is exactly 0.5, the result is rounded up if *real-expression* is positive or down if *real-expression* is negative.

The examples below return the INTEGER value shown.

Operation	Result	Comment
ABS(- 7)	7	
ABS(12)	12	
SQR(7)	49	(7*7)
SQR(- 4)	16	(- 4)*(- 4)
TRUNC(12.2)	12	TRUNC discards the fractional part.
TRUNC(- 12.3)	- 12	
ROUND(7.2)	7	ROUND rounds to the nearest integer
ROUND(7.8)	8	
ROUND(7.49999)	7	
ROUND(7.5)	8	ROUND rounds up to the nearest integer if the fractional part is 0.5 and the number is positive.
ROUND(- 7.5)	- 8	ROUND rounds down to the nearest integer if the fractional part is 0.5 and the number is negative.
ROUND(- 7.499999)	- 7	

Memory
The memory functions are used for information about memory usage.

MEMAVAIL
returns the number of unallocated bytes in main memory.

SCAN(*integer-expression*,<> *char-expression*, *multi-expression*)
SCAN(*integer-expression*, = *char-expression*, *multi-expression*)
scans memory comparing each byte with the character specified by *char-expression*. If an equal sign (=) precedes *char-expression*, the search is made for the first character that is the same as *char-expression*. If the unequal symbol (<>) precedes *char-expression*, the search is made for the first character that is different from *char-expression*. *Integer-expression* specifies the maximum number of bytes that can be searched. The value returned is the number of bytes searched minus 1. If *integer-expression* is negative, a backwards search is made and the value returned is the negative of the number of bytes searched. If the first byte satisfies the search, the value returned is 0; if the second byte satisfies the search, the value returned is 1, an so on. If no match or mismatch is found, the value returned is *integer-expression*. The search through memory begins at the location specified by *multi-expression*.

SIZEOF(*multi-expression*)
returns the number of bytes the variable specified by *multi-expression* takes up in memory. *Multi-expression* can be an identifier or one of the predefined data types.

For example, the number of bytes used for each data type ca be determined by using the SIZEOF function in imperatives a shown below.

Imperative	**Result Displayed**
WRITELN(SIZEOF(INTEGER));	2
WRITELN(SIZEOF(REAL));	8

Imperative	Result Displayed
WRITELN(SIZEOF(CHAR));	1
WRITELN(SIZEOF(STRING));	81
WRITELN(SIZEOF(BOOLEAN));	1

The program

```
100 PROGRAM exscan;
110 VAR bytes,storage,location:INTEGER;
120       str1:STRING;
130 BEGIN
140    bytes:=MEMAVAIL;
150    str1:='TI-74 Pascal is a subset of UCSD

Pascal';

160    storage:=SIZEOF(str1);
170    location:=SCAN(39,='U',str1[1]);
180  WRITELN('location = ',location);
190 END. (* exscan *)
```

displays the following.

```
location = 28
```

The following information is stored in the variables.

bytes	the number of bytes of memory that are available.
storage	the number of bytes of memory that str1 uses in memory.
location	the number of bytes (minus 1) that the interpreter searched until it found a character equal to (or the same as) the character U. The search starts at the first character in str1 and can continue for up to 39 bytes. In this case, 29 bytes are searched until the character U is found. Thus the variable location is assigned a value of 28.

You can display the contents of the variables `bytes`, `storage`, and `location` by using the following WRITELN imperatives.

```
WRITELN(bytes);
WRITELN(storage);
WRITELN(location);
```

Note: In versions of Pascal written for 16-bit processors, MEMAVAIL returns the number of unallocated 16-bit words in memory.

String
The string functions are used with strings.

LENGTH(*string-expression*)
returns the current length of the string specified by *string-expression*.

POS(*string-expression1*,*string-expression2*)
returns the position (searching from left to right) in *string-expression2* where the substring *string-expression1* begins. If *string-expression1* cannot be found within *string-expression2*, POS returns 0. If *string-expression1* occurs more than once within *string-expression2*, POS returns the first occurrence.

For example, the program

```
100 PROGRAM exstr;
110 VAR str1,str2:STRING;
120      position:INTEGER;
130      strlenth:INTEGER;
140 BEGIN
150   str2:='TI-74 Pascal is a subset of UCSD Pascal';
160   str1:='subset of UCSD Pascal';
170   position:=POS(str1,str2);
180   WRITELN('position = ',position);
190   strlenth:=LENGTH(str2);
200   WRITELN('strlenth = ',strlenth);
210 END. (* exstr *)
```

displays the following.

```
position = 19
strlenth = 39
```

The variable `position` contains the first occurrence of `str1` (`subset of UCSD Pascal`) in `str2` (`TI-74 Pascal is a subset of UCSD Pascal`). The variable `strlenth` contains the number of characters in `str2`.

Ranking
The ranking function is used to determine the position of its expression in its set of values.

ORD(*multi-expression*)
returns the ordinal value (or the rank) of *multi-expression*, which can be any type except REAL or STRING. The ORD of an INTEGER data type is that integer value. The ORD of a CHAR data type is the character's ASCII code. The ordinal value of FALSE is 0; the ordinal value of TRUE is 1.

Some examples of using the ORD function are shown below.

Operator	Result	Comments
ORD('p')	112	The ASCII code of p is 112.
ORD(FALSE)	0	The first value in a set has an ordinal value of 0.
ORD(– 5)	– 5	The ordinal of an integer is the integer itself.
ORD('E') – 1	68	68 is the ORD('D')
ORD('M') + 1	78	78 is the ORD('N')

al Functions The following functions return a REAL value.

ABS(*real-expression*)
returns the absolute value of the *real-expression*.

ATAN(*iorr-expression*)
returns the measurement of the angle in radians whose tangent is the *integer-* or *real-expression.*

COS(*iorr-expression*)
returns the cosine of the angle whose measurement in radians is the *integer-* or *real-expression.*

EXP(*iorr-expression*)
returns the result of e^x where x is the *integer-* or *real-expression.*

LN(*iorr-expression*)
returns the natural logarithm of the *integer-* or *real-expression.*

LOG(*iorr-expression*)
returns the common logarithm of the *integer-* or *real-expression.*

PWROFTEN(*integer-expression*)
returns 10 raised to the power specified by the *integer-expression*, which must be from 0 through 37.

SIN(*iorr-expression*)
returns the sine of the angle whose measurement in radians is the *integer-* or *real-expression.*

SQR(*real-expression*)
returns the square of the *real-expression.*

SQRT(*iorr-expression*)
returns the square root of the *integer-* or *real-expression.*

Some examples of these real functions are shown below.

Function	**Result**	**Comments**
ABS(− 4.5)	4.5	ABS always returns a positive value or zero
ATAN(4.5)	1.352127381	returns angle measured in radians

Function	Result	Comments
COS(4.5)	−0.2107957994	returns cosine of 4.5 radians
EXP(4.5)	90.0171313	$e^{4.5}$ is 90.0171313
LN(4.5)	1.504077397	
LOG(4.5)	0.6532125138	
PWROFTEN(4)	10000.0	10^4 is 10000.0
SIN(4.5)	−0.9775301177	returns sine of 4.5 radians
SQR(4.5)	20.25	4.5^2 is 20.25
SQRT(4.5)	2.121320344	$\sqrt{4.5}$

Character Function

The following function returns a character.

CHR(*integer-expression*)
returns the character that corresponds to the ASCII code of *integer-expression*.

Some examples of using the CHR function are shown below.

Operation	Result
CHR(38)	&
CHR(58)	:
CHR(62)	>
CHR(90)	Z
CHR(98)	b

String Functions

The following functions return a string.

CONCAT(*string-expression1*, *string-expression2*, *string-expression3*, ... *string-expression*n)
returns the string that is all the string expressions (*string-expression1*, *string-expression2*, ... *string-expression*n) concatenated or linked together. If the string returned by CONCAT is assigned to a string variable whose declared (or default) length is less than the concatenated string, the message `Truncation warning` is displayed. The string assigned to the string variable has a length the same as the declared (or default) length of the variable.

COPY(*string-expression*,*integer-expression1*, *integer-expression2*)
returns a substring of a string expression. The substring includes the characters in *string-expression* starting at the position specified by *integer-expression1* and continuing for *integer-expression2* characters. If the length specified by *integer-expression2* is too long, the COPY function is not performed.

The program

```
100 PROGRAM excopy;
110 VAR ex1,ex2,ex3:STRING;
120     stringex,substrin:STRING;
130 BEGIN
140   ex1:='T';
150   ex2:='I-74 Pascal is a subset of ';
160   ex3:='UCSD Pascal';
170   stringex:=CONCAT(ex1,ex2,ex3);
180   substrin:=COPY(stringex,19,21);
190   WRITELN(stringex);
200   WRITELN(substrin);
210 END. (* excopy *)
```

concatenates the strings ex1, ex2, ex3 and copies 21 of the characters in the concatenated string starting at position 19. The following output is produced.

```
TI-74 Pascal is a subset of UCSD Pascal
subset of UCSD Pascal
```

Boolean Functions

The following functions return a BOOLEAN value of TRUE or FALSE.

ODD(*integer-expression*)
returns a TRUE when the integer expression is odd or a FALSE when the integer expression is even.

EOF
returns a value of TRUE or FALSE regarding the end-of-file marker.

EOLN
returns a value of TRUE or FALSE regarding the end-of-line marker.

The EOF and EOLN functions are described later in "File Handling."

Multi-Type Functions

The following functions can have expressions that are INTEGER, CHAR, or BOOLEAN; the data type of the value returned is dependent on the data type of the expression. A REAL or STRING data expression cannot be used.

PRED(*multi-expression*)
(predecessor function) returns the value that precedes the value specified by *multi-expression*.

SUCC(*multi-expression*)
(successor function) returns the value that succeeds the value specified by *multi-expression*.

For example,

PRED('B') is A
SUCC('E') is F
PRED(5) is 4
SUCC(5) is 6
PRED(TRUE) is FALSE

If the expression of the PRED function has no preceding value, an error occurs when the function is executed. Likewise, if the expression of the SUCC function has no successor value, an error occurs when the function is executed.

TYPE Declarations

You can use the VAR declaration to declare variables that are any of the predefined data types (INTEGER, REAL, STRING, CHAR, and BOOLEAN). When you declare many variables of the same type, you can help yourself remember their uses by defining them with descriptive identifiers. An identifier can be declared in a TYPE statement as a user-defined type that is one of the predefined data types.

For example, the identifier `grade` can be declared a user-defined type that is the predefined CHAR type. As shown below, variables can then be declared in a VAR declaration to be that user-defined type.

Declaring an identifier as a REAL type is shown below.

```
130 TYPE angle=REAL;
140 VAR degree:angle;
150       radian:angle;
160       grad:angle;
```

You can also declare that the type of an identifier is to be a user-defined type. In the example below, the identifier `measure` is defined to be of the user-defined type `angle`.

```
130 TYPE angle=REAL;
140        measure=angle;
150 VAR degree:angle;
160       radian:angle;
170       grad:measure;
```

Note that a TYPE declaration does not allocate memory for the identifier it is defining to be used as a variable. The VAR declaration must be used to allocate space for variables.

Data Type Formats

In Pascal you can display data unformatted or you can specify the format for the data.

Unformatted Data

In TI-74 Pascal, no leading or trailing spaces are displayed with any unformatted item. An item of CHAR data type is displayed in one column. A STRING item is displayed in the number of columns required for the current length of the string.

A REAL data item is displayed in decimal notation if its value has 10 or less digits to the left of the decimal point. If the value has more than ten significant digits, the value is rounded. In Pascal, an item displayed in decimal notation always has at least one digit to the left of the decimal point and at least one digit to the right of the decimal point.

A REAL data item with a magnitude of 9999999999.49995 or greater is displayed in scientific notation as shown below.

*mantissa*E*exponent*

An item is displayed in scientific notation according to the following conventions.

- The *mantissa* is displayed with 7 or fewer digits with one digit to the left of the decimal point and at least one digit to the right of the decimal point.
- Trailing zeros are omitted in the fractional part of the *mantissa*.
- The *exponent* is displayed with a plus or minus sign followed by a two- or three-digit *exponent*.
- When the *exponent* has two digits, the *mantissa* is limited to seven digits; when the *exponent* has three digits, the *mantissa* is limited to six digits. When necessary, the *mantissa* is rounded to the appropriate number of digits.

For example, the values 123456789012.3456789 and −0.0000009876543210 are displayed as shown below.

Value	**Scientific Notation**
123456789012.3456789	1.234568E+11
−0.0000009876543210	−9.876543E−07

The statements

```
210 address:='2301 Ash #39';
220 count:=135;
230 grade:='a';
240 result:=3940.7125;
250 WRITELN(address,count,grade,result);
```

produce the following output.

```
2301 Ash #39135a3940.7125
```

Formatted Data

You can design the format of the output data by including a width specification next to each data item in an output statement. The width specification is the number of columns that are to be used to display the item. An item is formatted when it is followed by a colon and an integer expression that specifies the field width. If an item requires fewer columns than are specified, the item is displayed right-justified with leading blanks.

If the WRITELN statement in the above example is changed to

```
250 WRITELN(address:13,count:6,grade:2,result:
     10);
```

the output that is produced is

```
 2301 Ash #39   135 a 3940.7125
```

If an item requires more columns than are specified, the item is displayed in the number of columns necessary.

For example, the output statement

```
260 WRITELN(address:10,count:2,grade:2,result:
     4);
```

produces the following output.

```
2301 Ash #39135 a3940.7125
```

Because the field-width specifications for `address`, `count`, and `result` are too small, the items are displayed in the number of columns necessary to display their values.

For a REAL data item, you can also specify the number of digits to be displayed after the decimal point by including a colon and an integer-expression after the field-width specification. The value is then rounded to the specified number of decimal digits and displayed. For example, the statement

```
270 WRITELN(address:13,count:6,grade:2,result:
     10:2);
```

produces the output

```
 2301 Ash #39   135 a   3940.71
```

For INTEGER and REAL data, the field-width specification must allow a column for a negative sign. In addition, for REAL data the field-width specification must allow a column for the decimal point.

The output displayed from various WRITELN statements is shown below.

Statement	Display	Comments
200 WRITELN(3940.7125);	3940.7125	unformatted item
210 WRITELN(3940.7125:4);	3940.7125	unformatted because field-width is too small
220 WRITELN(3940.7125:5);	3940.7125	unformatted because field-width is too small
230 WRITELN(3940.7125:10:2);	3940.71	displayed right-justified in 10 columns and rounded to 2 decimal places
240 WRITELN(3940.7125:10:3);	3940.713	displayed right-justified in 10 columns and rounded to 3 decimal places
250 WRITELN(3940.7125:10:4);	3940.7125	displayed right-justified in 10 columns and rounded to 4 decimal places
260 WRITELN(– 3940.7125:4);	–3940.7125	unformatted because field-width is too small
260 WRITELN(– 3940.7125:5);	–3940.7125	unformatted because field-width is too small
270 WRITELN(– 3940.7125:10:3);	–3940.713	displayed right-justified in 10 columns with 3 decimal places

The maximum field-width specifications for TI-74 Pascal is shown in the table below.

Data Type	Maximum Field-Width Specification
INTEGER	80
REAL	14
CHAR	1
STRING	80

Note: You can display a string with a length greater than 80 by using no format specification. The string is displayed 80 characters at a time. The **ENTER** key can be pressed to display the next 80 characters in the string until the end of the string is reached.

Positioning the Cursor

You can position the cursor anywhere in the display by using the GOTOXY statement. The general form of the GOTOXY statement is

GOTOXY (*col,row*);

where *col* is an integer expression that indicates a column and *row* is an integer expression that indicates a row. The upper-left corner is assumed to be (0,0). For the TI-74, the column specification must be in the range 0-30 and the row specification must always be 0.

If an invalid column or row specification is specified, zero is used for that specification and the warning `Implementation restriction` is displayed. If the other specification is valid, GOTOXY uses it. If both the column and row specifications are out of bounds, two warnings are displayed and the cursor is placed at (0,0).

Any subsequent input/output begins at the location specified by the GOTOXY statement. For example, the following program

```
100 PROGRAM cursor;
110 VAR code:INTEGER;
120 BEGIN
130   WRITE('Enter code (1-5): ') {$w-};
140   GOTOXY(19,0);
```

```
150   READLN(code) {$w+};
160   GOTOXY(30,0);
170   WRITELN(code);
180 END. (* cursor *)
```

prompts for a code to be entered, which is then read at column 19 and displayed in column 30.

Review Chapter 4

1. If you are using a VAR, a CONST, and a TYPE declaration, write the declarations in the order in which they must appear in a Pascal program.

____

____

____

2. From the constant declarations below, what are the types for the variables being assigned?

   CONST

   a='g'; ______________________________

   b=3; ______________________________

   c='hello'; ______________________________

   d=5.5; ______________________________

   e=FALSE; ______________________________

3. Which of the following are not valid REAL values in a Pascal program and why are they not valid?

   7.567 E01 ______________________________

   .5 ______________________________

   4.5 ______________________________

   12. ______________________________

   35.7654 ______________________________

   +3.94567e-04 ______________________________

4. After the following statements are executed, what are the values of a and b?

```
a:=5;
b:=10;
a:=b;
b:=a;
```

a ______________________________

b ______________________________

5. Given the following variable declarations

```
VAR  a:INTEGER;
     b:REAL;
     c:CHAR;
     d:STRING;
```

when the statement READ(a,b,c,d); is executed and the following data input, what are the values of a, b, c, and d?

1234   35.5, the end

a ______________________________

b ______________________________

c ______________________________

d ______________________________

6. What is the output from each of the following statements?

```
WRITELN(83.545:4);      ________________
WRITELN(83.545:5:1);    ________________
WRITELN(83.545:7:2);    ________________
WRITELN(83.545:8:4);    ________________
WRITELN(-83.545:7:4);   ________________
WRITELN(-83.545:5);     ________________
WRITELN(-83.545:2:0);   ______________________

WRITELN(83.545:2);      ______________________
```

7. What is the type and the result of each of the following expressions?

Example	Result	Type
3+5		
5/2		
11 DIV 2		
5.6<5.8		
3*5.5		
3 IN [0..5]		
11 MOD 2		
'hello'<'HELLO'		

8. Write a program that reads a string, displays the length of the string, scans the string for the letter z, and displays the result of the scan.

9. Write a program that reads two strings and uses the POS function to determine if one of the strings is a substring of the other string and then displays the results of the POS function.

10. Write a program that reads an integer and a character and displays the predecessor and the successor of each.

11. Write a program that accepts a temperature in degrees Celsius, converts it to degrees Fahrenheit, and displays the result. To convert degrees Celsius to degrees Fahrenheit, multiply the Celsius temperature by 9/5 and add 32.

Introduction

The statements in a program are normally executed sequentially from the first statement to the last. Executing each statement in the order that it appears on the program listing is known as sequential execution. You can change the order in which the statements are executed by using control statements.

In Pascal, there are three classes of control statements.

- Repetition Statements
- Conditional Branch Statements
- Unconditional Branch Statements

Repetition Statements

Repetition statements are used when a section of a program is to be repeated a number of times. The program lines that are repeated are known as a loop. By using repetition statements, you can avoid duplicating lines.

For example, suppose you want to input three numbers from the keyboard and then display their sum. A program that uses sequential execution to input three numbers and print their sum is shown below.

```
100 PROGRAM add3; (*program to add 3 numbers*)
110 VAR next,sum:INTEGER;
120 BEGIN
130    sum:=0;
140    WRITE('Enter #: ') {$w-};
150    READLN(next);
160    sum:=sum+next;
170    WRITE('Enter #: ');
180    READLN(next);
190    sum:=sum+next;
200    WRITE('Enter #: ');
210    READLN(next);
220    sum:=sum+next {$w+};
230    WRITELN('Sum is ',sum);
240 END. (* add3 *)
```

If the program is supplied the three numbers

```
36
7
19
```

the program's final output is

```
Sum is 62
```

By using a repetition statement you can eliminate some program lines. There are three kinds of repetition statements: the FOR statement, the REPEAT statement, and the WHILE statement.

The FOR Statement

The FOR statement repeats a portion of a program a specific number of times. The FOR statement uses a counter that is incremented by one each time the loop is performed. This counter is also called a control variable. You must supply the starting value and the ending value of the counter.

The general form of the FOR statement is shown below.

FOR *counter*:=*start* TO *stop* DO *statement*

The previous program can be written with a FOR statement as shown below.

```
100 PROGRAM add3; (*program to add 3 numbers*)
110 VAR next,sum,count: INTEGER;
120 BEGIN
130   sum:=0; {$w-}
140   FOR count:=1 TO 3 DO
150     BEGIN
160       WRITE('Enter #: ');
170       READLN(next);
180       sum:=sum+next
190     END; (* count *)
200   {$w+}
210   WRITELN('Sum is ',sum)
220 END. (* add3 *)
```

When a FOR statement is first executed, the counter is assigned the value of the starting value. In this program, the counter is the identifier count, which has been defined as an INTEGER variable. Count begins with a value of 1.

The FOR statement instructs the computer to perform the next statement after the word DO as many times as the counter specifies. If one statement follows the reserved word DO, only that statement is executed in the loop. If the reserved word BEGIN follows the reserved word DO, all of the statements between this BEGIN and its matching END are treated as a single compound statement and are executed in the loop.

Note there is no semicolon after the reserved word DO in a FOR statement. The FOR statement performs the statement after the reserved word DO. A semicolon before an END statement is optional; a semicolon after the END statement the loop is required to separate it from the next statement.

In a FOR statement, the loop is executed and the counter is incremented by one. The loop is executed repeatedly until the value of the counter is greater than the ending value. In the preceding example, the loop is performed three times before the computer executes the next statement.

The control variable of a FOR statement can also be decremented by one using the reserved word DOWNTO instead of TO.

The general format for using DOWNTO in a FOR statement is

FOR *counter*:=*start* DOWNTO *stop* DO *statement*

The program

```
100 PROGRAM exdownto;
110 VAR count:INTEGER;
120 BEGIN
130   FOR count:=3 DOWNTO 1 DO
140   WRITE(count,' ');
150 END. (* exdownto *)
```

displays the values of the counter.

```
3 2 1
```

In Pascal, the counter can be an INTEGER, CHAR, or BOOLEAN variable. For example, the program

```
100 PROGRAM countval;
110 VAR count:CHAR;
120 BEGIN
130   FOR count:='A'TO'G' DO
140     WRITELN(count,' Fabrics :');
150 END. (* countval *)
```

displays the following output.

```
A Fabrics :
B Fabrics :
C Fabrics :
```

```
D Fabrics :
E Fabrics :
F Fabrics :
G Fabrics :
```

The starting and stopping values can also be variables or expressions. In the program below, the computer prompts for the starting and stopping values to be entered from the keyboard. The starting and stopping values are evaluated when the FOR statement is first executed.

```
100 PROGRAM forvalue;
110 VAR startval,stopval,count:INTEGER;
120 BEGIN
130   WRITE('enter starting integer: ')
      {$w-};
140   READLN(startval);
150   WRITE('enter stopping integer: ');
160   READLN(stopval) {$w+};
170   FOR count:=startval TO stopval DO
180     WRITELN(count,' squared is ',
        SQR(count));
190 END. (* forvalue *)
```

If -1 is entered as the starting integer and 3 is entered as the stopping integer, the program produces the following output.

```
-1 squared is 1

0 squared is 0

1 squared is 1

2 squared is 4

3 squared is 9
```

The following conventions apply to a FOR statement.

- The value of the control variable can be used for computations within the compound statement, but the value of the control variable cannot be modified.
- The starting value and the stopping value for the control variable cannot be changed within the compound statement. The control variable, the starting value, and the stopping value must be of the same type (usually INTEGER, but may be any predefined type except REAL or STRING).

- If TO is used and the starting value is greater than the stopping value, the FOR statement is not executed. If DOWNTO is used and the stopping value is greater than the starting value, the FOR statement is not executed.

- If the starting value equals the stopping value, the statement is executed once.

- The control variable must be a local variable; it cannot be a VAR parameter (described later in chapter 7).

- After a FOR statement has executed, the value of its control variable is undefined.

The REPEAT Statement

Another statement that can be used in Pascal to form a loop is the REPEAT statement. The general format of a REPEAT statement is shown below.

REPEAT *statement* UNTIL *Boolean-expression*

The REPEAT statement performs all the statements following the reserved word REPEAT down to the reserved word UNTIL and then tests the *Boolean-expression* after the word UNTIL. If the *Boolean-expression* is FALSE, the statements between REPEAT and UNTIL are performed again and the expression is tested again. This process is repeated as long as the *Boolean-expression* is FALSE. When the expression becomes TRUE, the loop is no longer performed and the statement following it is executed.

A REPEAT loop is **always** executed at least once because the *Boolean-expression* is tested after the loop has been executed.

Suppose you are loading boxes in a van that can hold up to 1900 pounds. The following program can be used to add the box weights one at a time until the total weight exceeds 1900 pounds.

```
100 PROGRAM maxweigh;
110 CONST maximum=1900.0;
120 VAR count:INTEGER;
130       weight,totalwt:REAL;
140 BEGIN
150    totalwt:=0.0;
160    count:=0;  {$w-}
```

```
170    REPEAT
180      WRITE('Enter box weights:');
190      READLN(weight);
200      totalwt:=totalwt+weight;
210      count:=count+1;
220    UNTIL totalwt>maximum; {$w+}
230    WRITELN(' Last box exceeded max.');
240    WRITELN(count,' boxes have exceeded',
       maximum:10:1,'.')
250 END. (* maxweigh *)
```

The REPEAT loop executes the statements from REPEAT to UNTIL until the total weight of the boxes is greater than the maximum allowed.

Note that you can enclose the statements in a REPEAT-loop with BEGIN and END, but they are unnecessary.

The WHILE Statement

Another form of loop statement is the WHILE statement. The general form of a WHILE statement is shown below.

`WHILE` *Boolean-expression* `DO` *statement*

A WHILE statement tests the *Boolean-expression* after the word WHILE. If the *Boolean-expression* is TRUE, the statement after the word DO is performed and the *Boolean-expression* is tested again. This process is repeated as long as the expression is TRUE. When the *Boolean-expression* is FALSE, the loop is not performed and the statement after the WHILE loop is executed.

If multiple statements are to be executed in the loop, they must be bracketed together into a single compound statement by the words BEGIN and END. The *Boolean-expression* used to control the loop is written before the loop, and thus there is no natural terminator for the loop (as there is in the REPEAT statement).

The previous example could be written using a WHILE statement as shown below. The difference between the two programs is that WHILE tests the *Boolean-expression* before the loop is executed. If the condition is FALSE to begin with, the loop is not executed. In a REPEAT statement, the loop is executed before the *Boolean-expression* is tested.

```
100 PROGRAM maxweigh;
110 CONST maximum=1900.0;
120 VAR count:INTEGER;
130       weight,totalwt:REAL;
140 BEGIN
150    totalwt:=0.0;
160    count:=0;  {$w-}
170    WHILE totalwt<=maximum DO
180       BEGIN
190          WRITE('Enter box weights:');
200          READLN(weight);
210          totalwt:=totalwt+weight;
220          count:=count+1;
230       END;  {$w+}
240    WRITELN('Last box exceeded max.');
250    WRITELN(count-1,'boxes within ',
       maximum:10:1,' max.')
260 END. (* maxweigh *)
```

The *Boolean-expression* in a WHILE statement is the inverse of the one in a REPEAT statement. The WHILE loop is performed so long as the weight is less than or equal to the maximum; the REPEAT loop is performed until the weight is greater than the maximum.

The following table summarizes the differences between the REPEAT and WHILE statements.

REPEAT	**WHILE**
The loop statement is performed at least once.	The loop statement may not be performed at all.
The loop is repeated only while the *Boolean-expression* is FALSE.	The loop is repeated only while the *Boolean-expression* is TRUE.
The reserved words REPEAT and UNTIL bracket the loop statements into a single compound statement.	The reserved words BEGIN and END are used to bracket multiple loop statements into a single compound statement.

Nested Loops

In Pascal, you can have a loop appearing as part of a statement within another loop. A loop embedded (or inside) another loop is called a nested loop. Any number of loops may be embedded within a loop until a program uses all of the available memory.

The following program displays a multiplication table for integers from one through nine. The first FOR-loop control variable is used as the INTEGER value for each of the rows. The second FOR-loop is embedded or nested inside the first FOR-loop and its control-variable is used as the INTEGER value for each of the columns. Thus, the outer FOR-loop control-variable specifies a row value that is multiplied by the inner FOR-loop control-variable, which specifies each successive column value. The nine products for each row are displayed.

```
100 PROGRAM table;
110 VAR count1,count2:INTEGER;
120 BEGIN
130   WRITELN(' Multiplication table: 1-9');
140   FOR count1:=1 TO 9 DO
150     BEGIN {$w-}
160       FOR count2:=1 TO 9 DO
170         WRITE(count1*count2:3);
180       WRITELN {$w+};
190     END;
200 END. (* table *)
```

The program on the next page uses a FOR, a REPEAT, and a WHILE loop to compute the windchill factor for a given temperature. The following variables are assigned values from the keyboard.

nooftemp	the number of temperatures to be entered (1-5)
veloc1	the starting value of the velocity of the wind
veloc2	the ending value of the velocity of the wind
wincremt	the wind velocity increment
tempture	the temperature in degrees Fahrenheit

The program calculates and displays the windchill factor for the given temperature from the starting wind velocity to the ending wind velocity using the specified increment. The program is performed until the number of temperatures specified has been entered.

```
100 PROGRAM windchil;
110 VAR nooftemp,wincremt:INTEGER;
120       tempture,velocl,veloc2,tempvel:INTEGER;
130       count:INTEGER;
140       wcfactor:REAL;
150 BEGIN {$w-}
160    REPEAT
170       WRITE('# of temperatures (1-5): ');
180       READLN(nooftemp);
190    UNTIL nooftemp IN[1..5];
200    WRITE('from ? mph of wind: ');
210    READLN(velocl);
220    WRITE('to ? mph of wind: ');
230    READLN(veloc2);
240    WRITE('Enter wind increment: ');
250    READLN(wincremt);
260    FOR count:=1 TO nooftemp DO
270       BEGIN
280          tempvel:=velocl;
290          WRITE('deg fahrenheit: ');
300          READLN(tempture); {$w+}
310          WHILE tempvel<=veloc2 DO
320             BEGIN
330                wcfactor:=91.4-(0.288*SQRT(tempvel)+0.45-
340                0.019*tempvel)*(91.4-tempture);
350                WRITELN(tempture,CHR(223),tempvel:5,'mph=wcf',
360                wcfactor:5:0,CHR(223));
370                tempvel:=tempvel+wincremt;
380             END;  (* tempvel<=veloc2 *)
390             {$w-}
400     END;  (* count *)
410 END.  (* windchil *)
```

When a *Boolean-expression* is constructed, be sure that the test is a valid one. For example, in the following program to print the positive even integers less than 11, the loop never stops because the control variable never has a value of 11.

```
100 PROGRAM evensum;
110 VAR count:INTEGER;
120 BEGIN
130    count:=2;
140    WHILE count<>11 DO (* infinite loop *)
150       BEGIN
160          WRITE(count);
```

```
170         count:=count+2;
180       END (* while count <>11 *)
190 END. (* evensum *)
```

Conditional Branch Statements

Conditional branch statements are used to test *Boolean-expressions* and, depending on the results of the test, to execute a specific part of a program. Pascal provides two kinds of conditional branch statements: the IF statement for binary choices, and the CASE statement for multiple choices.

The IF Statement

The IF statement is used when you have to decide between two options. The IF statement can take two forms.

IF *Boolean-expression* THEN *statement*

or

IF *Boolean-expression* THEN *statement* ELSE *statement*

In the first form, if the *Boolean-expression* is TRUE, the statement that follows the reserved word THEN is executed. If *Boolean-expression* is not TRUE, the statement following THEN is ignored and the next statement is executed.

```
IF age>18 THEN WRITELN('eligible for prize');

IF charactr='*' THEN READLN;

IF final<60 THEN grade:="F';
```

In the second form, if the *Boolean-expression* is TRUE, the statement following THEN is executed and the ELSE part is ignored. If the *Boolean-expression* is FALSE, the statement following ELSE is executed and the THEN part is ignored.

```
IF age>18
  THEN WRITELN('eligible for prize')
  ELSE WRITELN('ineligible for prize');

IF charactr='*'
  THEN READLN
  ELSE sum:=sum+1;

IF final<60
  THEN grade:='F'
  ELSE WRITELN('You passed');
```

Note that an IF-THEN-ELSE statement is a single statement that contains two parts. A semicolon, which is a statement separator, **must not** immediately precede the reserved wor ELSE; otherwise, the ELSE part would not be considered par of the IF statement. A semicolon is placed at the end of an IF- THEN-ELSE statement to separate it from the next statement.

When multiple statements follow the word THEN or ELSE, the statements must be enclosed in the reserved words BEGIN and END.

The following program accepts a sequence of prices and computes their total. The sequence is terminated when a zer price is encountered. If the total cost is greater than $100, a discount of 15% is given.

```
100 PROGRAM iftest;
110 VAR price,total:REAL;
120 BEGIN
130   total:=0.0;
140   WRITE('Amount: ') {$w-};
150   READLN(price);
160   WHILE price<>0.0 DO
170     BEGIN
180       total:=total+price;
190       READLN(price);
200     END; (* while price is not zero *)
210   IF total>100.00 {$w+}
220     THEN WRITELN('Total price is $',total-
230           0.15*total:10:2,'* discount *')
240     ELSE WRITELN('Total price is $',
        total:10:2)
250 END. (* iftest *)
```

Nested IF statements

Nested IF statements are used when you have many conditions to test and only one is true. For example, if a student's grade average is from 90 through 100, he is given an A; from 80 through 89, a B; from 70 through 79, a C; from 60 through 69, a D; and below 60, an F. After it is determined in which group an average falls, no other conditions are tested.

```
100 PROGRAM average;
110 VAR grade:REAL;
120 BEGIN
130  READLN(grade);
140  IF grade>=90
150  THEN WRITELN('grade is A')
160   ELSE
170     IF grade>=80
180       THEN WRITELN('grade is B')
190       ELSE
200         IF grade>=70
210           THEN WRITELN('grade is C')
220           ELSE
230             IF grade>=60
240               THEN WRITELN('grade is D')
250               ELSE WRITELN('grade is F');
260   WRITELN('finished test')
270 END. (* average *)
```

An ambiguity can result when nested IFs are used. In the example

```
500 IF count<10 THEN
         WRITELN('value less than 10');
510 IF count>5 THEN WRITELN('value in range')
520 ELSE WRITELN('value out of range')
```

it is not obvious whether the ELSE belongs to the first or the second IF. The rule in Pascal is that an ELSE is part of the closest IF that has not been matched. Thus in the example above, the ELSE is part of the second IF and is performed only when the second IF is FALSE.

This pairing of IFs and ELSEs can be changed by using the reserved words BEGIN and END. The example above can be changed so that the ELSE is matched with the first IF.

```
500 IF count<10
510   THEN
520     BEGIN
530     WRITELN('value less than 10');
540     IF count>5
550       THEN WRITELN('value in range');
560     END
570   ELSE WRITELN('value out of range');
```

The ELSE is not matched with the closest IF statement because that IF statement has been closed off by an END statement.

The following program accepts lengths for the three sides of a triangle. The largest number is determined and the lengths are displayed. The program then determines whether the lengths can form a triangle and if so, whether the triangle has a right angle.

```
100 PROGRAM triangle;
110 VAR sidel,side2,side3,temp:REAL;
120 BEGIN
130  WRITE('Enter three lengths: ') {$w-};
140  READ(sidel,side2,side3) {$w+};
150  IF sidel<side2
160    THEN
170     BEGIN
180       temp:=sidel;
190       sidel:=side2;
200       side2:=temp;
210     END;
220  IF sidel<side3
230    THEN
240      BEGIN
250        temp:=sidel;
260        sidel:=side3;
270        side3:=side2;
280        side2:=temp;
290      END
300    ELSE
310      IF side2<side3
320        THEN
330          BEGIN
340            temp:=side3;
350            side3:=side2;
360            side2:=temp;
370           END;
```

```
 WRITELN('Sides of ',side1:8:1,side2:8:1,side3:8:1);
 IF side1<side2+side3
   THEN
     BEGIN
       WRITE(' form a triangle');
       IF SQR(side1)=SQR(side2)+SQR(side3)
         THEN WRITELN(' that has a right angle')
     END
   ELSE WRITELN(' do not form a triangle')
END. (* triangle *)
```

The examples thus far have tested Boolean expressions that are formed by using relational operators with arithmetic expressions. Any type of Boolean expression can be formed to test in an IF statement. By combining two or more conditions using the words AND or OR, you can test more than one condition. If AND or OR is used to test relational expressions, the expressions should be enclosed in parentheses to ensure proper evaluation.

For example, in the statements

```
IF (grade>100) OR(grade<0)
   THEN WRITE('illegal grade');
IF (speed>45) AND(speed<55)
   THEN WRITE('driving legal speed');
```

the message `illegal grade` is displayed if grade is greater than 100 or if grade is less than zero. The message `driving legal speed` is displayed only if speed is greater than 45 and also less than 55.

The order of precedence for the Boolean operators from highest to lowest priority is NOT, AND, and OR.

Thus, the IF statement

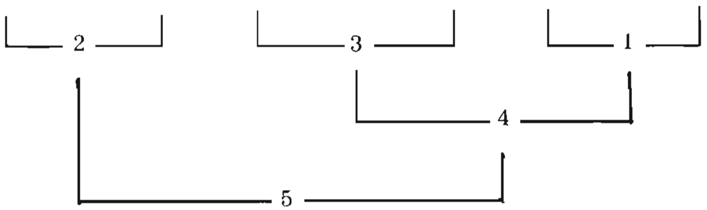
```
IF (speed>65) AND(weight<400) OR(speed>55) AND
(weight<300)
   THEN WRITE('Correct speed for jump');
```

is equivalent to the following statement.

```
IF ((speed>65) AND(weight<400))
   OR((speed>55) AND(weight<300))
THEN WRITE('Correct speed for jump');
```

The order of evaluation for the logical operators NOT, AND, and OR is shown in the diagram below.

```
IF (s>65) AND(w<400) OR(s>55) AND(w<300) AND NOT (angle<30)
```



```
THEN EXIT PROGRAM
```

The following program reads characters from the keyboard until an asterisk is entered. Each character must be a lowercase alphabetic letter, a digit from 0 through 9, or a period (.). Otherwise, a message is displayed.

```
100 PROGRAM logictst;
110 VAR ch:CHAR;
120 BEGIN
130   WRITE('Enter character: ') {$w-};
140   READLN(ch) {$w+};
150   WHILE ch<>'*' DO
160     BEGIN
170       IF NOT(ch IN['a'..'z']) AND
180          NOT(ch IN['.','0'..'9'])
190          THEN WRITELN('Illegal symbol!');
200       READLN(ch);
210     END; (* while ch is not '*' *)
220 END. (* logictst *)
```

The CASE Statement

The IF statement is used to make a decision between two cases (if a condition is TRUE or FALSE). In Pascal, the CASE statement can be used when the number of alternatives is greater than two, such as when an expression can evaluate to one of many values. This value can be an ordinal type (INTEGER, CHAR, or BOOLEAN).

A CASE statement has the general form shown below.

CASE *expression* OF
constant1:statement1;
constant2:statement2;
.
.
.
constantn:statementn;
END;

When a CASE statement is executed, the value of *expression* is compared with the constants in the constant list. If the value matches a constant, the statement that follows that constant is executed.

In the following program, a grade is entered and compared to the letters A, B, C, D, and F. An honor roll message is displayed if the grade is an A or B; a failure message is displayed if the grade is an F. No message is displayed if the grade is a C or D.

```
100 PROGRAM honoroll;
110 VAR grade:CHAR;
120 BEGIN
130   READ(grade);
140   CASE grade OF
150       'A':WRITELN('blue ribbon honor roll');
160       'B':WRITELN('red ribbon honor roll');
170       'C':  (* do nothing *);
180       'D':  (* do nothing *);
190       'F':WRITELN('failure list')
200   END; (* case grade *)
210   WRITELN('Finished');
220 END. (* honoroll *)
```

If the *expression* in the CASE statement does not evaluate to one of the given constants, all of the CASE statements are bypassed and the next statement after the CASE statement is executed. For example, lines 170 and 180 that correspond to a C or D can be made comments. When a C or D is entered, none

of the CASE statements between CASE and its correspondin
END match. Therefore, program execution continues with
line 210 as shown in the program below.

```
100 PROGRAM honoroll;
110 VAR grade:CHAR;
120 BEGIN
130   READ(grade);
140   CASE grade OF
150       'A':WRITELN('blue ribbon honor roll');
160       'B':WRITELN('red ribbon honor roll');
170       { 'C': (* do nothing *);}
180       { 'D': (* do nothing *);}
190       'F':WRITELN('failure list')
200   END; (* case grade *)
210   WRITELN('Finished');
220 END. (* honoroll *)
```

In a CASE statement, more than one constant may be
associated with a statement. For example, in the program

```
100 PROGRAM seasons;
110 VAR count: INTEGER;
120 BEGIN
130   WRITE('Enter number of month ') {$w-};
140   READLN(count); {$w+}
150   CASE count OF
160        1,2,12:WRITELN('winter month');
170        3,4,5:WRITELN('spring month');
180        6,7,8:WRITELN('summer month');
190        9,10,11:WRITELN('fall month')
200   END; (* case count *)
210 END. (* seasons *)
```

the number that is entered from the keyboard is compared
with the constants 1 through 12. If the number matches one
of the constants, the statement following that constant is
performed.

If more than one statement is to be executed after a constant
the statements must be enclosed between the reserved word
BEGIN and **END**. For example, the program

```
PROGRAM seasons;
VAR count:INTEGER;
BEGIN
  WRITE('Enter number of month ') {$w-};
  READLN(count); {$w+}
  CASE count OF
       1,2,12:BEGIN
         WRITELN('winter month');
         WRITELN('cactus, poinsettia');
       END;
   3,4,5:BEGIN
         WRITELN('spring month');
         WRITELN('tulip, rose');
       END;
    6,7,8:WRITELN('summer month');
    9,10,11:WRITELN('fall month')
  END; (* case count *)
END. (* seasons *)
```

displays two lines of output if the constant entered is **1**, **2**, **12**, **3**, **4**, or **5**.

The following conventions apply to the CASE statement.

- The constant(s) in the constant list must be of the same type as the expression and must be separated by commas when there are more than one.
- A constant should not appear more than once in a constant list; otherwise, only the first appearance of the constant is used.
- Multiple statements after a constant must be enclosed between the words BEGIN and END.
- A CASE statement must contain at least one statement preceded by a constant.
- The word END is paired with the word CASE, rather than with the word BEGIN.
- If no match occurs, program execution continues with the next statement after the CASE END statement.

Unconditional Branch Statements

The IF and CASE statements are used to perform conditional branching. The branch is performed depending upon the value of an expression. In Pascal, there is also an unconditional branch that enables you to execute another part of the program, regardless of the values of expressions. This unconditional branch is provided by means of a GOTO statement.

The GOTO Statement

The general form of a GOTO statement is

GOTO *label*

where *label* must be an integer in the range 0..9999 associated with a statement.

When a GOTO statement is performed, a branch is made to the specified labeled statement. The interpreter options remain unaltered during this branch. Program execution continues from the labeled statement. Note that a GOTO statement cannot branch into the middle of a repetition statement or a branch statement.

The statement branched to from a GOTO statement must be preceded by a *label* and a colon. Before you can use a *label*, however, you must declare it in a LABEL declaration, as explained in the next section.

LABEL Declarations

A LABEL declaration is used to declare an integer that can be used as a label. A label is an integer from 0 through 9999 followed by a colon that precedes a statement. This label is distinct from the line numbers that are used when entering program lines.

The general form for a LABEL declaration is

LABEL *integer1, integer2,..integer*n;

where *integer1, integer2,..integer*n must be values from 0 through 9999. You can declare multiple labels in a LABEL declaration by separating the *integers* with commas.

Any label that is declared must be used to label a statement. A label does not, however, have to be referenced by a GOTO statement.

Any LABEL declarations must precede CONST and VAR declarations.

Because Pascal was designed for structured programming, the use of GOTO statements is not encouraged. The variety of control statements available in Pascal usually makes a backwards jump in a program unnecessary. Occasionally however, a GOTO statement is indispensable for situations where a forward jump is needed.

The following program reads a line of input. If an asterisk (*) was entered, its position in the line is displayed; otherwise, the message `* not found` is displayed.

```
PROGRAM check;
LABEL 150;
VAR line:STRING;
    index:INTEGER;
BEGIN
 WRITE('Enter line: ') {$w-};
 READLN(line) {$w+};
 FOR index:=1 TO LENGTH(line) DO
  BEGIN
   IF line[index]='*'
    THEN
     BEGIN
      WRITELN('* is character ',index);
      GOTO 150;
     END;
  END;
 WRITELN('* not found');
 150:WRITELN(line);
END. (* check *)
```

Review Chapter 5

1. What are the three Pascal statements that can be used to construct a loop?

____

____

____

2. For conditional branching, an __________ statement can be used when there are two choices and a ________________ statement can be used when there are multiple choices.

3. In a REPEAT statement, the loop is performed until the specified Boolean expression is _________________________.

4. In a WHILE statement, the loop is performed while the specified Boolean expression is ________________.

5. Write a program that uses a FOR-loop to find the average weight of a group of people. The number of people in the group is entered, followed by their weights. Display the least and greatest weights and the average weight of the group.

6. Write a program that uses a REPEAT-loop to find the average weight of a group of people. The number of people in the group is not known; the weights are read until a zero (0) is input. Display the least and greatest weights and the average weight of the group.

7. Write a program that uses a WHILE-loop to find the average weight. This time a negative number signals the end of the input. Display the least and greatest weights and the average weight of the group.

8. What is the error in the following IF-THEN-ELSE statement?

```
IF a<>0
   THEN WRITELN('a is not zero');
   ELSE WRITELN('a is zero');
```

____

9. Write a program that accepts twelve integers with values from 1 through 12. Use an IF statement to determine whether a value less than 1 or greater than 12 is entered. If an invalid integer is entered, display a message and branch to the end of the program. Otherwise, use a CASE statement to display the name of the month that corresponds to the value entered. For example, if the number 2 is entered, the program displays `February`. If 5 is entered, the program displays `May`, and so on.

ntroduction

In programming languages, a structure called an array has been designed for storing large amounts of data in an ordered sequence. The first data value is stored in the first position of the array, the second data value is stored in the second position, and so forth. To process the data, you can access each data value from the first value to the last value, you can skip through the array and access only certain values, or you can directly access a specific value without starting at the beginning of the array.

You can use an array structure in a Pascal program if you declare the array name in the declaration section so that the interpreter can reserve storage locations for it. In the declaration, you declare the array name, the type of data that the array will hold, and how many storage locations the array will have. All the values stored in a particular array must be of the same type.

The declaration

```
VAR payments:ARRAY[1..10] OF REAL;
```

allows you to use an array called payments that has 10 storage locations for real data values. The type of data that the array can hold determines the base type of the array. In the example above, the base type of the array called payments is REAL.

Each storage location in the array is accessed by writing the name of the array followed by the position (or index). For example, payments[1] refers to the value stored in the first location of the array, payments[2] refers to the value in the second location, and so forth.

Each value in an array is referenced by the same identifier but the index is different. The values payments[1], payments[2]..payments[10] are called elements of the array. Each array element can be used like any other variable of the same data type.

The index of an array (also called a subscript) must be enclosed in brackets and can be an INTEGER or CHAR type. The least and greatest values for a subscript are the constants that are included in brackets in the declaration. The first constant specified must be less than or equal to the second constant. These constants implicitly declare the index type of the array, that is, if the index is an INTEGER type or CHAR type.

The index type tells how many values are in an array and how to access them, as shown in the examples below.

Declaration	Comments
`VAR rvals:ARRAY[1..25] OF REAL;`	The 25 REAL elements of the array `rvals` are referenced by `rvals[1]..rvals[25]`.
`range:ARRAY[-10..10] OF REAL;`	The 21 REAL elements of the array `range` are referenced by `range[-10]..range[10]`
`year:ARRAY[1975..2000] OF INTEGER;`	The 26 INTEGER elements of the array `year` are referenced by `year[1975]..year[2000]`.
`ch:ARRAY['g'..'p'] OF CHAR;`	The 10 CHAR elements of the array `ch` are referenced by `ch['g']..ch['p']`.
`st:ARRAY[1..10] OF STRING;`	The 10 elements of the array `st` are strings that are referenced by `st[1]..st[10]`.

If you attempt to use the wrong type of index or a larger or smaller value than was declared for the array, an error occurs.

Note that each array location has two quantities associated with it.

- An index (or subscript)
- The contents (or value) in the location

The following program assigns values to an array in sequential order from the first index value to the last. The VAR declaration causes the interpreter to allocate three consecutive storage locations for REAL data. The index (or subscript) of the array is an INTEGER type that can be from 1 through 3.

A FOR statement is used in this illustration because the integer control variable (which is incremented by one) can also be used as the array subscript to access each storage location in the array. The first real value entered from the keyboard is stored in the first storage location of the array payment, the second into the second location, and the third into the third location.

After the values are assigned, a FOR statement is used to access and display each element from the third down to the first.

```
100 PROGRAM exlarray;
110 VAR payments:ARRAY[1..3] OF REAL;
120       counter:INTEGER;
130 BEGIN
140   WRITE('Enter 3 real values: ') {$w-};
150   FOR counter:=1 TO 3 DO
160     READ(payments[counter]); {$w+}
170    FOR counter:=3 DOWNTO 1 DO
180     WRITELN('payments[',counter,'] is ',
190        payments[counter]);
200 END. (* program exlarray *)
```

An example of an array that stores string data is shown in the program below. Note that the strings are read in a READLN statement.

```
100 PROGRAM storestr;
110 VAR strarray:ARRAY[1..5] OF STRING;
120       index,counter:INTEGER;
130 BEGIN
140   WRITELN('Enter 5 strings');
150   FOR index:=1 TO 5 DO
160     BEGIN {$w-}
170       WRITE('Enter string ',index,':');
180       READLN(strarray[index]);
190      END; {$w+}
200    FOR index:=1 to 5 DO
210      WRITELN(strarray[index]);
220 END. (* storestr *)
```

The arrays described so far have been one-dimensional arrays. A one-dimensional array structure is used to hold the values in a list. A one-dimensional array has only one subscript written after the array name. The following program illustrates sequential access to the values in a one-dimensional array.

```
100 PROGRAM ex2array;
110 VAR multi10:ARRAY[1..4] OF INTEGER;
120       counter:INTEGER;
130 BEGIN
140    multi10[1]:=10;
150    multi10[2]:=20;
160    multi10[3]:=30;
170    multi10[4]:=40;
180    FOR counter:=1 TO 4 DO
190        WRITE(multi10[counter]:5)
200 END. (* program ex2array *)
```

The array multi10 is a one-dimensional array and the data values stored in its locations are stored just as they appear in list.

multi10[1]	multi10[2]	multi10[3]	multi10[4]
10	20	30	40

or

multi10[1]	10
multi10[2]	20
multi10[3]	30
multi10[4]	40

Declaring an Array Type

You can define an identifier as an array type by declaring the identifier in a type declaration. The base type can be any predefined type or user-defined type, except a file type (discussed in chapter 8).

The declaration

```
TYPE tally=ARRAY[1..25] OF INTEGER;
```

defines a new type called `tally`, an array with 25 elements. The data stored in the array must be of type INTEGER and the index of the array must be an integer from 1 through 25.

Note that the preceding TYPE declaration does not reserve 25 storage locations for `tally`. The TYPE declaration only defines `tally` as an ARRAY type with 25 INTEGER elements.

If the following VAR declaration is added after the TYPE declaration

```
TYPE tally=ARRAY[1..25] OF INTEGER;
VAR sales:tally;
```

the variable called `sales` is declared to be of the type `tally`. Because `tally` is an ARRAY type of 25 integers, the variable `sales` is an array that is allocated 25 storage locations for storing integers.

This type of declaration is useful especially when you have other variables with a type you have defined. For example, the statements

```
TYPE tally=ARRAY[1..25] OF INTEGER;
VAR  sales:tally;
     accounts:tally;
     parts:tally;
```

define three arrays, each with 25 integer locations. If the array size needs to be enlarged to 50, the only statement that needs to be changed is the type declaration. For example, if the declaration were changed to

```
TYPE tally=ARRAY[1..50] OF INTEGER;
```

each of the arrays `sales`, `accounts`, and `parts` would then have 50 locations.

Random Access to a One-Dimensional Array

The elements in an array can also be accessed randomly, that is, any element in an array can be directly accessed by using its array name and subscript in the array. The elements need not be accessed from the beginning of the array or in a particular order.

For example, in the program below, you determine how many integers (up to 100) to store in an array. Any element in the array can be displayed by entering the subscript of the array corresponding to that element.

```
100 PROGRAM randomac;
110 VAR count, index:INTEGER;
120       intval:ARRAY[1..100] OF INTEGER;
130       ch:CHAR;
140 BEGIN
150    REPEAT
160       WRITE('# of integers to enter: ')
          {$w-};
170       READLN(count);
180    UNTIL count IN[1..100];
190    FOR index:=1 TO count DO
200       BEGIN
210          WRITE('#: ');
220          READLN(intval[index]);
230       END;
240    WRITE('Display a value? (y or n)');
250    READLN(ch);
260    WHILE (ch='Y') OR(ch='y') DO
270       BEGIN
280          WRITE('Which value: ');
290          READLN(index);
300          IF (index>=1) AND(index<=count)
310             THEN WRITELN('Value is ',
                intval[index]) {$w+};
320          WRITE('Display a value? (y or n)')
             {$w-};
330          READLN(ch);
340       END; (* while ch=y or Y *)
350 END. (* randomac *)
```

Two-Dimensional Arrays

Often a one-dimensional array is not suitable for storing a set of data. Rather than storing the data in a one-dimensional array as a list, you could arrange the data in rows and columns (as in a table or matrix) by using a two-dimensional array.

For example, the rainfall (in inches) of a town for 10 years could be recorded in a table as shown below.

	J	F	M	A	M	J	J	A	S	O	N	D
1970	2.3	1.3	2.1	4.5	4.3	1.2	0.8	0.6	3.2	2.3	3.2	4.3
1971	1.5	2.3	2.6	5.6	3.5	0.2	0.5	1.2	3.8	3.2	4.3	2.1
1972	2.1	3.4	2.2	4.3	5.5	3.2	0.4	0.2	1.1	2.1	3.3	2.3
1973	2.2	2.4	5.3	5.1	2.1	0.3	0.5	0.1	3.2	2.5	3.4	2.1
1974	2.5	5.4	2.3	5.6	5.5	5.3	1.2	2.1	0.3	2.2	3.4	2.0
1975	1.4	2.1	3.1	6.5	6.4	2.3	1.5	1.3	1.6	3.2	2.0	3.2
1976	1.8	2.0	3.9	1.1	2.9	3.3	0.4	0.7	2.7	1.2	3.2	2.4
1977	1.1	2.1	3.4	4.1	2.9	4.0	0.5	0.2	1.8	2.9	1.2	1.1
1978	2.1	0.9	1.4	3.4	1.2	2.5	0.9	0.8	1.8	1.9	1.5	1.2
1979	1.5	2.6	2.1	3.6	4.6	2.5	0.6	0.9	1.9	1.2	3.2	1.6

To store this data in a two-dimensional array, the array must be declared at the beginning of the program with two dimensions. Declaring a two-dimensional array in a TYPE and a VAR declaration is shown below.

TYPE *identifier*=ARRAY [*m..n,p..q*] OF *type*;
VAR *identifier*: ARRAY [*m..n,p..q*] OF *type*;

where

identifier	is the user-defined name of the array.
m	is the least value for the row subscript.
n	is the greatest value for the row subscript.
p	is the least value for the column subscript.
q	is the greatest value for the column subscript.
type	is any type except file (discussed in chapter 8).

To store the values in the previous table in an array called `rainfall`, the array name `rainfall` is listed in a VAR declaration with 10 rows and 12 columns.

```
VAR rainfall:ARRAY[1970..1979,1..12] OF REAL;
```

A two-dimensional array can also be thought of as an array of elements, each of which is also an array. If you have an array with 5 elements,

array1
array2
array3
array4
array5

each element can be thought of as another array, as shown below.

array1_1	array1_2	array1_3	array1_4	array1_5
array2_1	array2_2	array2_3	array2_4	array2_5
array3_1	array3_2	array3_3	array3_4	array3_5
array4_1	array4_2	array4_3	array4_4	array4_5
array5_1	array5_2	array5_3	array5_4	array5_5

Thus, a two-dimensional array can also be declared as shown below.

TYPE *identifier*=ARRAY[*m..n*] OF ARRAY[*p..q*] OF *type*;

or

VAR *identifier*:ARRAY[*m..n*] OF ARRAY[*p..q*] OF *type*;

Accessing the Elements in a Two-Dimensional Array

The elements in a two-dimensional array are accessed by entering the array name followed by the element's row and column enclosed in brackets and separated by commas.

For example

`rainfall[1970,7]`	refers to the data value in the 1st row and the 7th column of the array `rainfall`.
`rainfall[1974,10]`	refers to the data value in the 5th row and the 10th column of the array `rainfall`.

The elements in a two-dimensional array can also be referenced as shown below.

rainfall[1970][7]	refers to the data value in the 1st row and the 7th column of the array rainfall.
rainfall[1974][10]	refers to the data value in the 5th row and the 10th column of the array rainfall.

A nested FOR statement is often used to access the elements of a two-dimensional array. The outer FOR statement processes the rows (or columns) in an array, while the inner FOR statement processes the columns (or rows). For example, in the following control structure

```
FOR row:=1970 TO 1979 DO
FOR column:=1 TO 12 DO
```

the first FOR statement is used to access each row of elements and the second FOR statement is used to access the elements in each column of a row.

The following program can be used to assign to the array rainfall the values listed in the preceding table. Note that the program is written so that the data is entered a row at a time.

```
100 PROGRAM ex4array;
110 VAR rainfall:ARRAY[1970..1979,1..12] OF
        REAL;
120         row,column:INTEGER;
130 BEGIN
140    FOR row:=1970 TO 1979 DO
150       FOR column:=1 TO 12 DO
160             READLN(rainfall[row,column]);
170 END. (* program ex4array *)
```

The elements in the array can be accessed in any order. For example, suppose you want to find the year in which the most rain fell for each month. The following program compares the elements in each row (year) of a column and determines the largest for each column, which is then displayed.

```
100 PROGRAM ex5array;
110 VAR rainfall:ARRAY[1970..1979,1..12] OF
        REAL;
120         year,row,column:INTEGER;
```

```
130      greatest:REAL;
140 BEGIN
150    FOR row:=1970 TO 1979 DO
160       FOR column:=1 TO 12 DO
170          READLN(rainfall[row,column]);
180    FOR column:=1 TO 12 DO
190       BEGIN
200          greatest:=0.0;
210          FOR row:=1970 TO 1979 DO
220             IF rainfall[row,column]>greatest
230                THEN
240                BEGIN
250                   greatest:=rainfall[row,column];
260                   year:=row;
270                END;
280          WRITELN('Year ',year,' Mo ',column,
290                'most rain= ',greatest);
300       END; (* column :=1 to 12 *)
310 END. (* program ex5array *)
```

Three-Dimensional Arrays

Some problems require an array with more than two dimensions. Suppose that in the previous example you need to store the amounts of rainfall for 5 towns. When the data is recorded on paper, the data for each town is printed on a separate page. When the data is stored in an array, a third dimension is defined to keep each town's table of rainfall.

The following declaration defines a three-dimensional array with 10 rows by 12 columns by 5 pages of REAL values.

```
VAR rainfall:ARRAY[1970..1979,1..12,1..5] OF
REAL;
```

Nested FOR-loops can be used to access the elements in the array. The outermost control structure can reference each of the 5 towns, and the table of values for each of the towns can be processed by accessing each column in each row.

The following program displays a histogram of the yearly rainfall for each of the 5 towns. Note that the starting and ending years and the numbers of columns and pages are defined as constants at the beginning of the program and can easily be changed.

First, the rainfall of 5 towns for 10 years is entered in the array `rainfall`. For each town, the number of inches of rainfall in a year is found and a colon (:) is displayed for each inch.

```
100 PROGRAM ex6array;
110 CONST startyr=1970;
120       endyr=1979;
130       colnum=12;
140       pagnum=5;
150 VAR rainfall:ARRAY[startyr..endyr,
      1..colnum,1..pagnum] OF REAL;
160     row,col,pag,scale:INTEGER;
170     total:REAL;
180 BEGIN (* program body *)
190   FOR pag:=1 TO pagnum DO
200    FOR row:=startyr TO endyr DO
210     FOR col:=1 TO colnum DO
220      BEGIN
230       WRITE('town[,'pag,'],yr[',row,
             '],mo[',col,']?') {$w-};
240       READLN(rainfall[row,col,pag]) {$w+};
250      END;
260   FOR pag:=1 TO pagnum DO
270    BEGIN
280     WRITELN('Rainfall for town # ',pag);
290     FOR row:=startyr TO endyr DO
300      BEGIN
310       WRITE(row,'- ') {$w-};
320       total:=0.0;
330       FOR col:=1 TO colnum DO
340        total:=total+rainfall[row,col,pag];
350       scale:=TRUNC(total);
360       FOR col:=1 TO scale DO WRITE(':');
370       {$w+} WRITELN;
380      END; (* row:=startyr to endyr *)
390    END; (* pag:=1 to pagnum *)
400 END. (* ex6array *)
```

Arrays of Characters

An array of characters holds a sequence of characters. Suppose you want to enter a paragraph of text from the keyboard and have a program display the number of times each of the 26 letters in the alphabet is used. If a digit from 0 through 9 is entered, the digit is replaced with its equivalent English word.

The following program reads characters until either an asterisk (*) or 1000 characters have been read. The characters are stored in a character array. Note that when text is read, a program must determine if it has reached the end of a line.

The program below uses the EOLN function (described in chapter 8) to determine if the end of the line (the [ENTER] character) has been read. If an input statement has read the end-of-line character, the next character is read for the variable. Otherwise, the variable would be assigned a space.

After the text is read, it is displayed. Any digit is replaced with its equivalent English word. The number of times each alphabetic character is used is then displayed.

```
100 PROGRAM chartest;
110 CONST maxchar=1000;
120 VAR numchar: ARRAY['a'..'z'] OF INTEGER;
130       charactr:ARRAY[1..maxchar] OF CHAR;
140       chindex,charlet:CHAR;
150       counter,position:INTEGER;
160 BEGIN (* program chartest *)
170    FOR chindex:='a' TO 'z' DO
180       numchar[chindex]:=0;
190    counter:=0;
200    READ(charlet);
210    WHILE charlet<>'*' DO
220       BEGIN
230          IF EOLN
240             THEN READ(charlet);
250          IF charlet IN['a'..'z']
260             THEN numchar[charlet]:=
                        numchar[charlet]+1;
270          counter:=counter+1;
280          charactr[counter]:=charlet;
290          READ(charlet);
300          IF counter=maxchar
310             THEN charlet:='*';
320       END; (* charlet <>'*' *)
330    WRITELN;
340    FOR position:=1 TO counter DO
350       BEGIN
360          IF charactr[position] IN['0'..'9']
370             THEN CASE charactr[position] OF
380                     '0':WRITE('zero ');
390                     '1':WRITE('one ');
400                     '2':WRITE('two ');
410                     '3':WRITE('three ');
420                     '4':WRITE('four ');
430                     '5':WRITE('five ');
440                     '6':WRITE('six ');
450                     '7':WRITE('seven ');
460                     '8':WRITE('eight ');
```

```
470                  '9':WRITE('nine ');
480                 END (* case statement *)
490            ELSE WRITE(charactr[position]);
500        END; (* position:=1 to counter *)
510      WRITELN;
520      FOR chindex:='a' TO 'z' DO
530        WRITELN(chindex,': ',numchar[chindex])
540 END. (* program chartest *)
```

Packed Arrays

In Pascal the data can be stored in an array in the minimum amount of storage by preceding the word ARRAY with PACKED. By packing data into arrays, you can save memory space. Some routines require packed arrays for processing.

The declaration

```
170 VAR letter:PACKED ARRAY[1..26] OF CHAR;
```

allocates 26 storage locations for the array `letter`. The elements of `letter` must be characters.

A PACKED ARRAY OF CHAR and a STRING data type are similar, but not synonymous. The length of a PACKED ARRAY OF CHAR is always that specified in its declaration; the length of a string can change during program execution. A string's length is its dynamic length, that is, the number of characters last assigned to it.

You can assign to a STRING data type the value of a string constant and to a CHAR data type the value of a character constant. You cannot assign to a PACKED ARRAY OF CHAR the value of a string identifier because the string's length is dynamic. However, you can assign to a PACKED ARRAY OF CHAR a quoted string if their lengths are the same.

For example, suppose the following variables are declared.

```
110 VAR str1,str2:STRING;
120       chararay:PACKED ARRAY[1..28] OF CHAR;
130       ch:CHAR;
```

Then the following assignments are valid.

```
250 str1:='This string is 28 characters';
260 str2:='This is 10';
270 ch:='a';
280 chararay:='This string is 28 characters';
```

However, the following assignment is invalid.

```
300 chararay:=str1;
```

You can reference a PACKED ARRAY OF CHAR with its identifier and one less dimension than you declared in the declaration section. For a PACKED ARRAY OF CHAR with one dimension, you can access the entire array with only the identifier. For example, if the following arrays are declared

```
VAR pac1:PACKED ARRAY[1..80] OF CHAR;
    pac2:PACKED ARRAY[1..80] OF CHAR;
```

the identifiers pac1 and pac2 can be used to access all 80 elements in the respective arrays.

In the following program, values are entered from the keyboard and assigned to pac1. Because pac1 and pac2 are each a PACKED ARRAY OF CHAR of the same length, one assignment statement (line 170) can be used to assign the elements of pac1 to pac2.

```
100 PROGRAM entire;
110 VAR pac1:PACKED ARRAY[1..80] OF CHAR;
120     pac2:PACKED ARRAY[1..80] OF CHAR;
130     count: INTEGER;
140 BEGIN
150    FOR count:=1 TO 80 DO
160      READ(pac1[count]);
170    pac2:=pac1;
180 END. (* entire *)
```

The following relational operators can be used to compare elements in one PACKED ARRAY OF CHAR with an equal number of elements in another PACKED ARRAY OF CHAR.

equality	=	returns a TRUE result if each array element in one array is equal to its corresponding element in another array.
not equal to	<>	returns a TRUE result if any array element in one array is unequal to its corresponding element in another array.

less than	<	returns a TRUE result if the first element in the left array that is not equal to its corresponding element in the right array is less than that element.
less than or equal to	<=	returns a TRUE result if the first element in the left array that is not equal to its corresponding element in the right array is less than that element or if the left array equals the right array.
greater than	>	returns a TRUE result if the first element in the left array that is not equal to its corresponding element in the right array is greater than that element.
greater than or equal to	>=	returns a TRUE result if the first element in the left array that is not equal to its corresponding element in the right array is greater than that element or if the left array equals the right array.

For example, suppose the declarations

```
VAR pacname1:PACKED ARRAY[1..4] OF CHAR;
    pacname2:PACKED ARRAY[1..4] OF CHAR;
```

and the assignments

```
pacname1:='Glen';
pacname2:='Gary';
```

have been made. Then the results returned by the following operations are as shown.

Operation	Result	Comments
pacname1 = pacname2;	FALSE	Every character in Glen is not equal to its corresponding character in Gary.
pacname1<>pacname2;	TRUE	At least one character in Glen is not equal to its corresponding character in Gary.

Operation	Result	Comments
pacname1<pacname2;	FALSE	The second character in Glen (l) is not less than the second character in Gary (a).
pacname1<=pacname2;	FALSE	The second character in Glen (l) is not less than or equal to the second character in Gary (a).
pacname1>pacname2;	TRUE	The second character in Glen (l) is greater than the second character in Gary (a).
pacname1>=pacname2;	TRUE	The second character in Glen (l) is greater than or equal to the second character in Gary (a).

You can also use the assignment and relational operators described above with two- or three-dimensional packed arrays and specify one less dimension than you declared in the declaration.

For example, in the following program the array ch is declared a packed array with two dimensions. Therefore, the identifier can be used with one subscript. In the first FOR-loop, five characters are assigned to the five elements in the first row of the array (ch[1]). In the second FOR-loop, ch is used with one dimension. The reference ch[index] accesse all of the elements of the row specified by index. The reference ch[1] accesses all of the elements in the first row. Therefore, the values of the elements in the first row are assigned to the elements in each successive row. Each row of the array is then displayed.

```
100 PROGRAM expack;
110 VAR ch:PACKED ARRAY[1..10,1..5] OF CHAR;
120      index:INTEGER;
130 BEGIN
140   FOR index:=1 TO 5 DO
150     READ(ch[1,index]);
160   FOR index:=2 TO 10 DO
170     ch[index]:=ch[1];
180   WRITELN('Array assigned');
```

```
190   FOR index:=1 TO 10 DO
200     WRITELN(ch[index])
210 END. (* expack*)
```

If the following characters are entered, the output is as shown.

Input: **SCOTT**
Output:
```
Array assigned
SCOTT
SCOTT
SCOTT
SCOTT
SCOTT
SCOTT
SCOTT
SCOTT
SCOTT
SCOTT
```

Review Chapter 6

1. In the array declaration, VAR `test:ARRAY['a'..'z'] OF INTEGER;`, the base type of the array `test` is __________ and the index type is __________.

2. The index type of an array can be a __________________ o
____ type

3. How many storage locations are reserved for the array `sales` in the following declaration?

```
TYPE sales=ARRAY[1..100] OF INTEGER;
```

4. Write a program that reads a word and displays it backwards.

5. Write a program that reads 15 integers and displays them i descending numerical order.

6. If a program contains the following declarations

```
VAR pac1:PACKED ARRAY[1..10] OF CHAR;
    pac2:PACKED ARRAY[20..30] OF CHAR;
    pac3:PACKED ARRAY[1..18] OF CHAR;
    pac4:PACKED ARRAY[1..10,1..10] OF CHAR;
    st1:STRING[10];
    st2:STRING;
    ch:CHAR;
```

which of the following statements are invalid and why?

```
st1:='hello';                     ____________
st2:='18 characters long';        ____________
pac1:=st2;                        ____________
pac3:=st2;                        ____________
pac1:='18 characters long';       ____________
pac3:='18 characters long';       ____________
pac1:=pac2;                       ____________
pac2<>pac3;                       ____________
pac4[1]:=pac1;                    ____________
pac4:=pac1;                       ____________
```

hapter 7—Procedures and Functions

ntroduction

A program can become rather long and hard to read when all the statements are written in one long sequence. It is easier to read, write, understand, and debug a program if you group statements into blocks that accomplish a specific task. By using a top-down design, you can produce programs that are organized into blocks of programming tasks. First you write the general outline of the program and then define each step in greater detail.

Suppose you want a program to display a message and four asterisks, a message and six asterisks, and a message and four asterisks. The general outline of this program is shown below.

```
PROGRAM usingpro;
BEGIN
   display message and four asterisks
   display message and six asterisks
   display message and four asterisks
END.
```

To perform each of the steps in the outline program, each step must be defined in greater detail. The following statements display a message and four asterisks.

```
WRITE('**');
WRITE(' four asterisks ');
WRITELN('**');
WRITELN('****');
```

The next group of statements display a message and six asterisks.

```
WRITE('***');
WRITE(' six asterisks ');
WRITELN('***');
WRITELN('******');
```

By declaring each group of statements to be a procedure, you can organize the program into blocks of programming tasks. When a section of a program body is declared to be a procedure (or a function), that section can be executed several times in a program but need be written only once. All procedure and function declarations must appear immediately before the BEGIN of the program body.

The statements in a procedure or function are executed (or called) at any point in a main program body or another procedure or function body where the procedure or function identifier appears. The difference between a procedure and a function is that a procedure is used like a statement to perform a routine, whereas a function is used like a variable to supply a value that may be used in an expression. Procedures are used to make a program modular and easier to understand. A function is used to compute a single value, which is then assigned to the function identifier.

Procedure Declarations

Procedures enable you to:

- write shorter programs by not replicating code.
- divide a problem into smaller independent subproblems.
- alter a program more easily because the alteration can be made in a procedure without affecting other procedures.
- write programs that are easier to understand because the programs are broken into logical sections.

In simplest form, a procedure appears as shown below. Notice that a procedure follows the pattern of a Pascal program in general. Just as the term program block refers to all of the declarations and statements in a program, the term procedure block refers to all of the declarations and statements in a procedure. However, in TI-74 Pascal, procedures and functions cannot be declared within a procedure.

procedure heading	`PROCEDURE identifier;`
declarations	`LABEL declarations`
	`CONST declarations`
	`TYPE declarations`
	`VAR declarations`
procedure body	`BEGIN`
	`   statements`
	`END;`

Procedure Block

All declarations used must appear at the beginning of a procedure block in the same order as that of a program. A procedure must have a BEGIN and an END, just as a program does. However, the END statement of a procedure is followed by a semicolon (;); the END statement of a program is followed by a period (.). It is good programming practice to place the name of the procedure in a comment following END to improve program readability.

In the preceding program sections that display a message and four asterisks and a message and six asterisks, each program section can be declared to be a procedure, as shown below. Line 120 declares the first section as a procedure named astrisk4 and line 200 declares the second section as a procedure named astrisk6.

```
120 PROCEDURE astrisk4;
    (* Message with four asterisks *)
130   BEGIN
140     WRITE('**');
150     WRITE(' four asterisks ');
160     WRITELN('**');
170     WRITELN('****');
180   END;  (* astrisk4 procedure *)
200 PROCEDURE astrisk6;
    (* Message with six asterisks *)
210   BEGIN
220     WRITE('***');
230     WRITE(' six asterisks ');
240     WRITELN('***');
250     WRITELN('******');
260   END; (* astrisk6 procedure *)
```

After a procedure has been defined in a declaration, the procedure can be executed in the program body by using its name as you would a statement. At each point where the name of the procedure is written, the body of the procedure is executed as if it were inserted into the program at that point.

In the program body, suppose you want to execute the first group of statements, then the second group of statements, and again the first group of statements. The program body would then contain the statements shown on the next page.

```
100 PROGRAM usingpro;
    (* example program using procedures *)
110    (* procedure declaration *)
120 PROCEDURE astrisk4;
    (* Message with four asterisks *)
130 BEGIN
140   WRITE('**');
150   WRITE(' four asterisks ');
160   WRITELN('**');
170   WRITELN('****');
180 END; (* astrisk4 *)
190 PROCEDURE astrisk6;
    (* Message with six asterisks *)
200 BEGIN
210   WRITE('***');
220   WRITE(' six asterisks ');
230   WRITELN('***');
240   WRITELN('******');
250 END; (* astrisk6 *)
260 BEGIN (*  program body *)
270   astrisk4; (* Executes astrisk4 proc *)
280   astrisk6; (* Executes astrisk6 proc *)
290   astrisk4; (* Executes astrisk4 proc *)
300 END.  (* usingpro *)
```

The statements in the program body that contain astrisk4 and astrisk6 are called procedure calls. A procedure call causes the statements defined as that procedure in the declaration section to be executed.

Function Declarations

Functions are used like variables in expressions. When an expression is evaluated,

- the value stored in a variable location is used where the variable name appears in that expression.
- the value of a function is computed where the function name appears in that expression with the data input to it.

In simplest form, a function appears as shown below. Notice that a function follows the pattern of a Pascal program in general. A function block contains a declaration section and a statement section. However, in TI-74 Pascal, procedures and functions cannot be declared within a function.

function heading	`FUNCTION identifier:type;`
declarations	`LABEL declarations`
	`CONST declarations`
	`TYPE declarations`
	`VAR declarations`
function body	`BEGIN`
	`statements` (including at least one executed statement that assigns a value to the name of the function)
	`END;`

Function Block

A FUNCTION declaration, like a VAR declaration, must end with a colon followed by a data type and a semicolon. The data type indicates the type of value that the function returns. Pascal functions can return INTEGER, REAL, BOOLEAN, or CHAR type values.

All declarations used must appear at the beginning of a function block in the same order as that of a program. A function must have a BEGIN and an END, just as a program does. However, the END statement of a function is followed by a semicolon (;); the END statement of a program is followed by a period (.). It is good programming practice to place the name of the function in a comment following END to improve program readability.

Ending a Procedure or Function

A procedure or function terminates and returns to its caller when the END statement is encountered. You can, however, terminate a procedure or function before the END statement by using the EXIT or HALT procedures. You can also have a procedure or function terminate program execution by using either the EXIT or HALT procedures.

Note that a GOTO statement cannot be used to branch out of or into a procedure or function. A GOTO statement used in a procedure or function must branch to a statement in the block containing the GOTO statement.

In the first example below, the procedure `return` uses the EXIT procedure to terminate execution of the procedure after the message `return is terminating` is displayed at line 280. Program execution continues at line 360.

In the second example, the procedure `return` uses the EXIT procedure to terminate execution of the program after the message `return is terminating` is displayed at line 280. Note that the EXIT procedure can use the identifier of the program or the reserved word PROGRAM to terminate program execution.

In the third example, the procedure `return` uses the HALT procedure to terminate program execution after the message `return is terminating` is displayed at line 280. The HALT at line 290 turns on the error indicator and displays the message `Programmed Halt`.

Example 1

```
100 PROGRAM usingpro; (* example program using procedures *)
110     (* procedure declaration *)
120 PROCEDURE astrisk4; (* Message with four asterisks *)
130 BEGIN
140   WRITE('**');
150   WRITE(' four asterisks ');
160   WRITELN('**');
170   WRITELN('****');
180 END; (* astrisk4 *)
190 PROCEDURE astrisk6; (* Message with six asterisks *)
200 BEGIN
210   WRITE('***');
```

```
  WRITE(' six asterisks ');
  WRITELN('***');
  WRITELN('******');
END; (* astrisk6 *)
PROCEDURE return; (* termination procedure *)
BEGIN
  WRITELN('return is terminating');
  EXIT(return);
  WRITELN(' before this statement is displayed');
END; (* procedure return *)
BEGIN (*  program body *)
  astrisk4; (* Executes astrisk4 procedure *)
  astrisk6; (* Executes astrisk6 procedure *)
  return; (* Executes return procedure *)
  astrisk4; (* Executes astrisk4 procedure *)
END.  (* usingpro *)
```

Example 2

```
PROGRAM usingpro; (* example program using procedures *)
   (* procedure declaration *)
PROCEDURE astrisk4; (* Message with four asterisks *)
BEGIN
  WRITE('**');
  WRITE(' four asterisks ');
  WRITELN('**');
  WRITELN('****');
END; (* astrisk4 *)
PROCEDURE astrisk6; (* Message with six asterisks *)
BEGIN
  WRITE('***');
  WRITE(' six asterisks ');
  WRITELN('***');
  WRITELN('******');
END; (* astrisk6 *)
PROCEDURE return; (* termination procedure *)
BEGIN
  WRITELN('return is terminating');
  EXIT(usingpro);
  WRITELN(' before this statement is displayed');
END; (* procedure return *)
BEGIN (*  program body *)
  astrisk4; (* Executes astrisk4 procedure *)
  astrisk6; (* Executes astrisk6 procedure *)
  return; (* Executes return procedure *)
  astrisk4; (* Executes astrisk4 procedure *)
END.  (* usingpro *)
```

Example 3

```
PROGRAM usingpro; (* example program using procedures *)
   (* procedure declaration *)
PROCEDURE astrisk4; (* Message with four asterisks *)
BEGIN
  WRITE('**');
  WRITE(' four asterisks ');
  WRITELN('**');
  WRITELN('****');
END; (* astrisk4 *)
PROCEDURE astrisk6; (* Message with six asterisks *)
BEGIN
  WRITE('***');
  WRITE(' six asterisks ');
  WRITELN('***');
  WRITELN('******');
END; (* astrisk6 *)
PROCEDURE return; (* termination procedure *)
BEGIN
  WRITELN('return is terminating');
  HALT;
  WRITELN(' before this statement is displayed');
END; (* procedure return *)
BEGIN (*  program body *)
  astrisk4; (* Executes astrisk4 procedure *)
  astrisk6; (* Executes astrisk6 procedure *)
  return; (* Executes return procedure *)
  astrisk4; (* Executes astrisk4 procedure *)
END.  (* usingpro *)
```

arameters

Procedures and functions can optionally be supplied values for use in their routines. Supplying a value to a procedure or a function increases its utility because the defined routine can be used to perform operations on any number of values.

The values passed to a procedure or a function are called parameters. When a procedure or function is called, the main program specifies the parameters to be used. These values are called actual parameters and are included in parentheses after the procedure or function identifier.

When a procedure or function identifier appears in a program body, it is called a procedure or function call, respectively. When a procedure or function call that includes actual parameters is encountered in a program, the current value of each actual parameter is passed to the procedure or function. The routine then uses these values in its calculations.

A procedure or function that is passed a value must list in its declaration the variable that is to receive the passed value. The variables listed in a procedure or function declaration are called formal parameters. All formal parameters must have their type (the type of data that is to be stored there) defined in the declaration.

For example, if an integer value is passed to the procedure graph, its declaration must include the variable to which the integer value is assigned, followed by a colon and the reserved word INTEGER as shown in the example below.

```
PROCEDURE graph(formpar1:INTEGER);
```

The interpreter reserves memory space for the variables listed as formal parameters so that these variables need not appear in a VAR declaration within the procedure or function. Variables in a procedure or function that are not parameters and are not declared in the main program, however, are defined in a VAR declaration within the procedure or function. Note that reserved words cannot appear as formal parameters.

The following program prompts for the number of asterisks that are to be displayed. After a number is input to the program, the procedure graph is called and the program passes the number entered from the keyboard to the procedure. The procedure then displays the specified number of asterisks.

```
100 PROGRAM exprocl;
110 VAR times:INTEGER;
120 PROCEDURE graph(count:INTEGER);
130 VAR counter:INTEGER;
140 BEGIN {$w-}
150    FOR counter:=1 TO count DO
160       WRITE('*'); {$w+}
170    WRITELN;
180 END; (* graph *)
190 BEGIN (* program exprocl *)
200    WRITE('Enter # of char. to display: ')
       {$w-};
210    READLN(times) {$w+};
220    graph(times);
230 END.  (* program exprocl *)
```

The procedure declaration of graph requires that any reference to graph in the program body must include an integer expression in parentheses. This expression is evaluated and its value passed to graph when the procedure call is executed. Graph then assigns the value passed to it to the variable called count.

The procedure graph could be altered so that a program can specify both the character that is displayed and the number of times the character is displayed. In this case, the procedure graph requires two formal parameters that have different types and must be separated by semicolons in the declaration. Any call to graph in the main program must then include two actual parameters (which can be any two expressions) provided that the first evaluates to an integer and the second represents a character. The two parameters are separated by a comma.

The following program illustrates a procedure call that passes two parameters.

```
100 PROGRAM exprocl;
110 VAR times:INTEGER;
120       prcharac:CHAR;
130 PROCEDURE graph(count:INTEGER;
                    charactr:CHAR);
140 VAR counter:INTEGER;
150 BEGIN {$w-}
160    FOR counter:=1 TO count DO
170       WRITE(charactr); {$w+}
180    WRITELN;
```

```
190   END; (* graph *)
200   BEGIN (* program exprocl *)
210     WRITE('Enter character: ') {$w-};
220     READLN(prcharac);
230     WRITE('Enter # of char. to display: ');
240     READLN(times) {$w+};
250     graph(times,prcharac);
260 END.  (* program exprocl *)
```

The order, number, and type of the actual parameters must correspond exactly to the order, number, and type of the formal parameters. Actual parameters are separated by commas. Formal parameters of the same type can be listed together, separated by commas, with the type specified once at the end of the list. Formal parameters of different types must be separated from each other by semicolons.

For example, if the procedure finddata is passed three integers, one real number, and a character, the following formal declaration could be used.

```
150 PROCEDURE finddata(number1,number2,
160 number3:INTEGER;realval:REAL;
    charactr:CHAR);
```

The actual parameters in a call to this finddata procedure must include three integer expressions, a real expression, and a character expression, such as the one shown below.

```
finddata(5,10,15,4.5,'a');
```

lobal nd Local dentifiers

The identifiers declared after a program heading and before any procedure or function declarations are called global identifiers. They may be used in any part of the program, including within a procedure or function.

An identifier declared in a procedure or function is called a local identifier and can be used only within the procedure or function in which it is declared. A local identifier is undefined outside its procedure or function.

A local identifier supersedes a global identifier. If the same identifier is declared to be both global and local, a reference to the identifier in the procedure or function where it is declared accesses the identifier declared in that procedure or function.

Declaring the same identifier as both global and local, though is not a good programming practice and can lead to problems, as described later in this section.

In the following program, for example, the identifier times defined immediately after the program heading is a global variable and can be used anywhere in the program except in graph. The identifier times defined in the procedure graph is a local variable and can be used only in the procedure graph. While the procedure graph is executing, the value of its local variable times increments from 1 through the input number. The global variable times remains unchanged.

```
100 PROGRAM exprocl;
110 VAR times:INTEGER;
120       prcharac:CHAR;
130 PROCEDURE graph(count:INTEGER;
      charactr:CHAR);
140 VAR times:INTEGER;
150 BEGIN {$w-}
160    FOR times:=1 TO count DO
170       WRITE(charactr); {$w+}
180    WRITELN;
190 END; (* graph *)
200 BEGIN (* program exprocl *)
210    WRITE('Enter character: ') {$w-};
220    READLN(prcharac);
230    WRITE('Enter # of char. to display: ');
240    READLN(times); {$w+}
250    graph(times,prcharac);
260 END.  (* program exprocl *)
```

Passing Information

Parameters can be used to pass information to and from a procedure or function. A parameter that only passes information to a procedure or a function is called a value parameter. Using a value parameter results in a one-way transfer of data. A value parameter can be a constant, a variable, or an expression whose value is passed to a procedure or function.

A parameter that passes information to a procedure and returns information back to the calling program is called a VAR (or reference) parameter. Using a VAR parameter results in a two-way transfer of data. A VAR parameter must be a variable because information is stored in it.

One-Way Transfer

A value parameter is declared by including the name of the parameter and its type in a procedure or function heading. The interpreter reserves space for each value parameter in the heading. When a procedure or function is called, the value of each actual parameter in the call is stored in its corresponding value parameter.

If an actual parameter corresponds to a value parameter, its value is not affected by the called routine. When a procedure or function changes a formal parameter that is a value parameter, its corresponding actual parameter remains unchanged.

Two-Way Transfer

Procedures and functions are much more useful, however, when they can return information to the calling program. Normally, if one value is calculated by a program section and returned to the calling program, the section is declared a function. When multiple values are returned, the section is declared a procedure.

User-Defined Functions

In a function, the function name appears like a variable and is assigned a value. This value must be of the same data type as was declared for the function. Suppose you need to determine which of three real values is largest. The following program accepts three real values and uses a function to determine which value is largest. The function name is then assigned the value of the largest real number.

```
100 PROGRAM example;
110 VAR num1,num2,num3:REAL;
120 FUNCTION largest(val1,val2,val3:REAL):REAL;
130 VAR greater,greatest:REAL;
140 BEGIN
150   IF val1>val2
160     THEN greater:=val1
170     ELSE greater:=val2;
180   IF greater>val3
190     THEN greatest:=greater
200     ELSE greatest:=val3;
210   largest:=greatest;
220 END; (* function largest *)
230 BEGIN
240   WRITE('Enter three values: ') {$w-};
250   READLN(num1,num2,num3) {$w+};
260   WRITELN('Largest #: ',
                  largest(num1,num2,num3));
270 END. (* program example *)
```

Note that a function can have a value that is an INTEGER, REAL, BOOLEAN, or CHAR type and yet have parameters that are of another type. In the following example, the function same has parameters that are three real values but the value that it returns is a Boolean value. Three real values are input to the function same and if any two of the three sides are equal, the function has a value of TRUE. If no two of the three sides are equal, the function has a value of FALSE. If the value of same is TRUE, a message is displayed that at least two sides are equal. If the value is FALSE, a message is displayed that no sides are equal.

```
100 PROGRAM triangle;
110 VAR num1,num2,num3:REAL;
120 FUNCTION largest(val1,val2,val3:REAL):REAL;
130 VAR greater,greatest:REAL;
140 BEGIN
150    IF val1>val2,
160      THEN greater:=val1
170      ELSE greater:=val2;
180    IF greater>val3
190      THEN greatest:=greater
200      ELSE greatest:=val3;
210    largest:=greatest;
220 END; (* function largest *)
230 FUNCTION same(num1,num2,num3:REAL):BOOLEAN;
240 BEGIN
250    same:=TRUE;
260    IF ABS(num1-num2) >0.0001
270      THEN
280        IF ABS(num2-num3) >0.0001
290          THEN
300             IF ABS(num1-num3) >0.0001
310                THEN same:=false
320 END; (* function same *)
330 BEGIN
340    WRITE('Enter three values: ') {$w-};
350   READLN(num1,num2,num3) {$w+};
360   WRITELN('Largest # entered: ',
370      largest(num1,num2,num3));
380   IF same(num1,num2,num3)
390     THEN
          WRITELN('At least 2 sides are equal')
400     ELSE WRITELN('No sides are equal');
410 END. (* program triangle *)
```

Caution: If you use a variable name in the parameter list of a user-defined function called by a user-defined function or procedure, and then use that same variable name within your program, unexpected and incorrect results can occur. The following program shows an example of this mistake.

```
100 PROGRAM badreslt;
110 VAR i,j:INTEGER;
120 FUNCTION func1(i:INTEGER):INTEGER;
130 BEGIN
140     func1:=i*2;
150 END; (*func1*)
160 BEGIN   (*badreslt*)
170     i:=4;
180     j:=func1(func1(i));
190     WRITELN('I= ',i,' J= ',j);
200 END.    (*badreslt*)
```

The function call in line 180 produces an erroneous result because the variable i is used both in the main program and in the parameter list of func1 called by a function (in this case, itself). To correct this program, change the variable i in lines 120 and 140 to a unique name such as fi. You could also use a different variable within the program, such as k in lines 110, 170, 180, and 190.

This error only happens when a function is used in the parameter list of a function or procedure call and the same variable is used both in the parameter list and in the function or procedure referenced.

To avoid this kind of problem, use unique variable names in each section of a program if you are using more than one user-defined function.

ser-Defined rocedures

When multiple values are returned from a routine, the routine should be declared a procedure. Parameters are used to transfer information out of a procedure by declaring them in the procedure heading as VAR (for variable) parameters. A VAR parameter includes every identifier between the reserved word VAR and the next colon and type identifier. The reserved word VAR can appear more than once in a procedure heading.

For example, in the procedure declaration

```
PROCEDURE ex(VAR angle:REAL;count:INTEGER;
   VAR side1,side2,side3:REAL);
```

the identifier angle is a VAR parameter whose type is REAL, the identifier count is a value parameter whose type is INTEGER, and the identifiers side1, side2, side3 are VAR parameters whose types are REAL.

The interpreter allocates no storage locations for VAR parameters; the memory location of each actual parameter in the procedure call is used as the memory location of its corresponding formal VAR parameter. Thus, the calling program references a memory location by the identifier listed as the actual parameter, whereas the procedure references the same location by the identifier listed as the formal VAR parameter.

For example, if the procedure ex is declared as shown

```
PROCEDURE ex(VAR angle:REAL;count: INTEGER;
   VAR side1,side2,side3:REAL);
```

and the procedure is called by the following statement

```
ex(radian,quantity,value1,value2,value3);
```

the variables angle and radian share the same location as do the variables side1 and value1, side2 and value2, and side3 and value3.The variable count is allocated memory space when the procedure is called and the value of quantity is stored there.

If a procedure changes an identifier that is declared to be a VAR parameter, the value in that location, which is also the location of the actual parameter, is changed. When control returns to the calling program, the value of the actual parameter is what was stored there by the procedure.

Note that information passed in a VAR parameter is said to be passed by reference. Because the value of a VAR parameter can be changed by a procedure, all actual parameters corresponding to VAR parameters must be variables.

VAR parameters and value parameters can appear in any order in a procedure heading. The actual parameters in the procedure call must be in the same order.

Suppose that in the previous program you want to sort from largest to smallest the real numbers that are input. The program would then contain a procedure that returns three values in the VAR parameters passed to it.

```
100 PROGRAM triangle;
110 VAR num1,num2,num3:REAL;
120 PROCEDURE largest(VAR side1,side2,
              side3:REAL);
130 VAR temp:REAL;
140 BEGIN
150   IF side1<side2
160     THEN
170       BEGIN
180         temp:=side1;
190         side1:=side2;
200         side2:=temp;
210       END;
220   IF side1<side3
230     THEN
240       BEGIN
250         temp:=side1;
260         side1:=side3;
270         side3:=side2;
280         side2:=temp;
290       END
300     ELSE
310       IF side2<side3
320         THEN
330           BEGIN
340             temp:=side3;
350             side3:=side2;
360             side2:=temp;
370           END;
380  END; (* procedure largest *)
390  BEGIN
400   WRITE('Enter three sides: ') {$w-};
410   READLN(num1,num2,num3) {$w+};
420   largest(num1,num2,num3);
430   WRITELN('Sides are ',num1:7:2,
            num2:7:2,num3:7:2);
440 END. (* program triangle *)
```

Note that the declaration

`PROCEDURE largest(VAR side1,side2,side3:REAL);`

- Defines side1, side2, and side3 as REAL values.
- Declares side1, side2, and side3 as variables within the procedure largest.
- Defines side1, side2, and side3 as VAR parameters, thus allowing the values of the corresponding actual parameters to be changed.

If a section of program changes the value of a global parameter or performs input or output, the section has side effects (that is, the program section has an effect other than through its parameters). It is better to avoid side effects wher possible by adding parameters. However, performing input and output in a procedure or function cannot be avoided by using more parameters.

To summarize, parameters used only to pass information to a procedure are called value parameters and can be any expression, including a constant or a variable. Parameters that are used both to pass information to a procedure and to return values from the procedure are called VAR (or reference) parameters, and must be variables.

A procedure or function may not be passed as a parameter to another routine. Constants, elements of a packed array, and FOR loop counters may not be passed as VAR parameters. A file (discussed in chapter 8) can be passed only as a VAR parameter.

Array Parameters

An array identifier can appear as the parameter of a procedure. Individual elements of an array can be passed to a procedure as well as constants, variables, and expressions. Any of the following types of reference to the elements of an array can be used in a call to a procedure.

```
realval[5]
```

```
realval[8]*0.5+6
```

```
realval[index]
```

Note that the data type of a parameter must be included in the procedure heading. An array description such as `ARRAY[`*1..n*`] OF` *type*;, however, is not allowed. An array type must be declared and used in the procedure heading.

For example, the heading

```
(*   ERROR   *)
150 PROCEDURE sums(grade:ARRAY[1..25] OF
    INTEGER);
(*   ERROR   *)
```

is not allowed and causes an error. The array grade can be declared a type as shown below and its declared type included in the heading.

```
150 TYPE arraypar=ARRAY[1..25] OF INTEGER;
180 PROCEDURE compute(grade:arraypar);
```

In a procedure, an array parameter should be specified as a VAR parameter. The procedure can access the array in the calling program, rather than copying the array into the procedure. Memory space is saved by passing an array as a VAR parameter.

Suppose you have three salespersons' records, each of which contains the number of items that the individual has sold in each quarter of a year.

Salesperson #1

	Item 1	Item 2
Quarter 1	20	35
Quarter 2	60	75
Quarter 3	30	28
Quarter 4	38	59

Salesperson #2

	Item 1	Item 2
Quarter 1	30	45
Quarter 2	34	87
Quarter 3	40	79
Quarter 4	56	43

Salesperson #3

	Item 1	Item 2
Quarter 1	25	49
Quarter 2	83	54
Quarter 3	67	98
Quarter 4	23	56

Suppose you want to calculate the total quantity of sales for each salesperson, the quarterly sales for each salesperson, and the total amount of sales per item. The following program accepts the data from the three salespersons' records and uses arrays to calculate the sales amounts.

```
100 PROGRAM salesrec;
110 CONST quarter=4;
120       items=2;
130       people=3;
140 TYPE amount=ARRAY[1..quarter,1..items,1..people] OF INTEGER;
150 cost=ARRAY[1..items] OF REAL;
160 VAR sales:amount;
170       price:cost;
180       netsales,qrtsales,itemsale:REAL;
190       totsales:REAL;
200       count:INTEGER;
210 PROCEDURE item(part:INTEGER; VAR sarr:amount;
220            VAR salepr:cost;VAR tsales:REAL);
230 VAR season,person:INTEGER;
240 BEGIN
250  tsales:=0.0;
260  FOR person:=1 TO people DO
270   FOR season:=1 TO quarter DO
280    tsales:=tsales+sarr[season,part,person]*salepr[part];
290 END; (* procedure item *)
300 PROCEDURE yearpart(season:INTEGER; VAR sarr:amount;
310            VAR salepr:cost; VAR qsales:REAL);
320 VAR part,person:INTEGER;
```

```
330 BEGIN
340   qsales:=0.0;
350   FOR part:=1 TO items DO
360    FOR person:=1 TO people DO
370     qsales:=qsales+sarr[season,part,person]*salepr[part];
380 END; (* procedure yearpart *)
390 PROCEDURE total(person:INTEGER; VAR sarr:amount;
400     VAR salepr:cost;VAR tsales:REAL);
410 VAR season,part:INTEGER;
420 BEGIN
430  tsales:=0.0;
440  FOR season:=1 TO quarter DO
450   FOR part:=1 TO items DO
460    tsales:=tsales+sarr[season,part,person]*salepr[part];
470 END; (* procedure total *)
480 PROCEDURE initiliz(VAR sarr:amount;
490    VAR salepr:cost);VAR season,part,person:INTEGER;
500 BEGIN
510  FOR person:=1 TO people DO
520   BEGIN
530     WRITELN(' Enter sales for person #',person);
540     FOR season:=1 TO quarter DO
550       FOR part:=1 TO items DO
560         BEGIN
570           REPEAT
580             WRITE(' Quarter ',season,' item ',
                part,':') {$w-};
590             READLN(sarr[season,part,person]);
600           UNTIL sarr[season,part,person] IN[1..32767];
610     END; {$w+}
620   END;
630  FOR part:=1 TO items DO
640   BEGIN
650    REPEAT
660     WRITE(' Enter price of item ',part) {$w-};
670     READLN(salepr[part]);
680    UNTIL NOT(salepr[part]<0.0)
685    OR(salepr[part]>10000.0);
```

(program continued on next page)

```
  END; {$w+}
END; (* procedure initiliz *)
BEGIN (* program body *)
  initiliz(sales,price);
  totsales:=0.0;
  FOR count:=1 TO people DO
   BEGIN
    total(count,sales,price,netsales);
    WRITELN('Sales for # ',count,': $',netsales:12:2);
    totsales:=totsales+netsales;
   END; (* count:=1 to people *)
  WRITELN('Total sales: $',totsales:12:2);
  FOR count:=1 TO quarter DO
   BEGIN
    yearpart(count,sales,price,qrtsales);
    WRITELN(' Quarter ',count,' sales: $',qrtsales:12:2);
   END; (* count:=1 to quarter *)
  FOR count:=1 TO items DO
   BEGIN
    item(count,sales,price,itemsale);
    WRITELN('Item ',count,' sales: $',itemsale:12:2);
   END; (* count:=1 to items *)
END. (* program salesrec *)
```

If the data in the preceding examples is entered for the sales persons and $1000.00 and $2000.00 are entered for the prices of items 1 and 2, respectively, the output is as shown below.

```
Sales for #1: $    542000.00
Sales for #2: $    668000.00
Sales for #3: $    712000.00
Total sales: $   1922000.00
Quarter 1 sales: $    333000.00
Quarter 2 sales: $    609000.00
Quarter 3 sales: $    547000.00
Quarter 4 sales: $    433000.00
Item 1 sales: $    506000.00
Item 2 sales: $   1416000.00
```

The FORWARD Declaration

In Pascal, a procedure or function can call another procedure or function only if it has already been declared in the program. However, when many procedures and functions are called, it may be impossible to define each one before it is called. Therefore, Pascal provides a declaration called FORWARD that allows you to use a procedure or function identifier in a routine before it has been defined.

The reserved word FORWARD is written in place of the procedure or function block. If this procedure or function has parameters, they are specified in the FORWARD declaration and not in the declaration that contains the routine's block. A procedure or function identifier that appears in the FORWARD declaration may be used even though its program block has not been defined previously.

In the example below, the procedure change can use procedure a1 or a2 or the function compute even though the blocks for these routines have not been defined at that point.

```
100 PROGRAM main;
110 PROCEDURE a1(largest:REAL);FORWARD;
120 PROCEDURE a2(smallest:REAL);FORWARD;
130 FUNCTION compute(x,y:REAL):REAL;FORWARD;
140 PROCEDURE change(deg,rad:REAL);
150 VAR factor:REAL;
160       val,yval:REAL;
170 BEGIN
        .
        .
        .
        a1(xval);
        factor:=compute(xval,yval);
        a2(yval);
        .
        .
        .
    END; (* procedure change *)
    PROCEDURE a1;
      BEGIN
      .
      .
      END; (* procedure a1 *)
    PROCEDURE a2;
      BEGIN
      .
      .
      END; (* procedure a2 *)
```

(program continued on next page)

```
FUNCTION compute;
  BEGIN
  .
    compute:= real_expression
  .
  END; (* function compute *)
BEGIN
  .
  .
  .
END. (* main *)
```

Intrinsic Procedures

Pascal provides a number of pre-defined procedures, called intrinsic procedures, which can be accessed by writing the name of the procedure in place of a statement. The intrinsic procedures DELETE, INSERT, and STR are used with string data. The intrinsic procedures FILLCHAR, MOVELEFT, and MOVERIGHT can be used with multiple types of arguments.

String Procedures

The following string procedures are used to manipulate strings.

DELETE(*string-variable,integer-expression1,integer-expression2*)

returns in *string-variable* the string that results when the number of characters specified by *integer-expression2* are omitted from *string-variable* starting at the position specified by *integer-expression1*. Both *integer-expression1* and *integer-expression2* must be positive integers. If the number of characters specified by *integer-expression2* is more than the number of characters that can be deleted, no characters are deleted.

INSERT(*string-expression,string-variable,integer-expression*)

inserts *string-expression* into *string-variable* starting at the position specified by *integer-expression*.

STR(*integer-expression,string-variable*)

returns in *string-variable* the string representation of *integer-expression*.

The following program uses the INSERT procedure to insert into the string hello a string entered from the keyboard. The string hello is then displayed. The number of unallocated bytes returned by MEMAVAIL is represented as a string by the procedure STR. The procedure INSERT then inserts the string into numbytes. The DELETE procedure then deletes the inserted characters from numbytes, and MEMAVAIL is called to determine how many unallocated bytes of memory are left. The number is changed to its string representation by STR and inserted into numbytes, which is then displayed.

```
100 PROGRAM greeting;
110 VAR instr:STRING[24];
120     hello:STRING[100];
130     numbytes:STRING[100];
140     nbytes:INTEGER;
150     strnum:STRING;
160 BEGIN
170   hello:='Hi there,  ! You are using Pascal
          on a computer!';
180   numbytes:='You have  bytes of memory left';
190   WRITE('Enter your name: ') {$w-};
200   READLN(instr) {$w+};
210   INSERT(instr,hello,11);
220   WRITELN(hello);
230   nbytes:=MEMAVAIL;
240   STR(nbytes,strnum);
250   INSERT(strnum,numbytes,10);
260   WRITELN(numbytes);
270   DELETE(numbytes,10,LENGTH(strnum));
280   nbytes:=MEMAVAIL;
290   STR(nbytes,strnum);
300   INSERT(strnum,numbytes,10);
310   WRITELN(numbytes);
320 END. (* greeting *)
```

Array Procedures

The array procedures are normally used with arrays; however, these procedures may be used with any other data types (except files). These procedures are FILLCHAR, MOVELEFT, and MOVERIGHT.

FILLCHAR(*multi-variable,integer-expression,character-expression*)

FILLCHAR fills a specified number of bytes starting at the location specified by *multi-variable* with the character specified by *character-expression*. *Integer-expression* specifies the number of bytes that are filled. FILLCHAR can be used to fill a specific number of character positions with blanks or zeros.

MOVELEFT(*multi-variable1,multi-variable2,integer-expression*)

moves the number of characters specified by *integer-expression* from *multi-variable1* to *multi-variable2.*

MOVERIGHT(*multi-variable1,multi-variable2,integer-expression*)

moves the number of characters specified by *integer-expression* from *multi-variable1* plus *integer-expression* minus 1 to *multi-variable2* plus *integer-expression* minus 1.

In the example on the next page, FILLCHAR fills the array ch with asterisks and displays the array. MOVELEFT copies 13 bytes (characters) of pac2 into pac1. The first character of pac2 is moved to the 31st character of pac1, the second character of pac2 is moved to the 32nd character of pac1, and so on until 13 characters have been moved. The arrays pac1 and pac2 are then displayed.

MOVERIGHT copies 19 bytes of pac1 starting at pac1[11] + 19 minus 1 into pac1 starting at pac1[6] + 19 minus 1. The character in pac1[11] + 19 minus 2 is then moved into pac1[6] + 19 minus 2, the character in pac1[11] + 19 minus 3 to pac1[6] + 19 minus 3, and so on until 19 bytes have been moved. The array pac1 is then displayed. Note that if a byte is modified and its contents then moved, the new character in the byte is moved.

```
PROGRAM example;
TYPE charray=PACKED ARRAY[1..40] OF CHAR;
VAR ch:charray;
     pac1:PACKED ARRAY[1..43] OF CHAR;
     pac2:PACKED ARRAY[1..13] OF CHAR;
BEGIN
   FILLCHAR(ch,40,'*');
   WRITELN('ch is ',ch);
   pac1:='move characters from the left or the right.';
   pac2:='one at a time';
   MOVELEFT(pac2,pac1[31],13);
   WRITELN(pac1);
   WRITELN(pac2);
   MOVERIGHT(pac1[11],pac1[6],19);
   WRITELN(pac1);
END. (* example *)
```

Output:

```
ch is ****************************************
move characters from the left one at a time
one at a time
move left left left left left one at a time
```

Caution: *FILLCHAR, MOVELEFT, and MOVERIGHT* explicitly perform as you tell them. You can have FILLCHAR, MOVELEFT, and MOVERIGHT write over system data and thus have to reset the computer to continue operation. Use caution when you specify parameters for these procedures.

Recursion

In Pascal, a procedure or a function can call itself, a feature known as recursion. A routine cannot call itself indefinitely, however, or an overflow condition occurs. A recursive routine must contain a method of termination. When the condition of termination is met, the recursive routine returns control to the point where the procedure or function was originally called.

A useful example of a recursive routine application is calculating a factorial. A factorial is defined as the product of all the positive integers up to a given integer, including the product of the given integer. The factorial of an integer is written with the integer followed by an exclamation mark. For example, the factorial of 4 is written as 4!.

By definition the factorial of zero is 1. The factorials of the integers from 1 through 5 are computed as shown on the next page.

Given Integer	Factorial
1	1
2	1*2
3	1*2*3
4	1*2*3*4
5	1*2*3*4*5

To use a recursive function to find the factorial of an integer, you must first define the function as equal to 1 when the given integer is 1. Thus the function factoral(1) is defined to be equal to 1 (factoral = 1). The factorial of 2 then becomes 1*2, which can be written as factoral(1)*2. In the factorial of 3, 1*2*3, the product 1*2 can be replaced by factoral(2) and 1*2*3 becomes factoral(2)*3. The table below illustrates how a factorial is found for the first five positive integers.

Integer	Factorial	Function	Function Computation
2	1*2	factoral(2)	1*2 factoral(1)*2
3	1*2*3	factoral(3)	1*2*3 (1*2)*3 factoral(2)*3
4	1*2*3*4	factoral(4)	1*2*3*4 (1*2*3)*4 factoral(3)*4
5	1*2*3*4*5	factoral(5)	1*2*3*4*5 (1*2*3*4)*5 factoral(4)*5

From the examples above then, the general formula

factoral(n)= factoral($n-1$)*n

can be derived for computing the factorial of an integer. The function is written as a recursive function in the following program.

```
100 PROGRAM recursiv;
110 VAR intnum:INTEGER;
120 FUNCTION factoral(n:INTEGER):INTEGER;
130 VAR fact:INTEGER;
140 BEGIN
150   IF n>=1
160     THEN fact:=factoral(n-1)*n
170     ELSE fact:=1;
180   factoral:=fact;
190 END; (* factoral *)
200 BEGIN (* program body *)
210   REPEAT
220     WRITE('Enter integer (1-7): ') {$w-};
230     READLN(intnum) {$w+};
240   UNTIL intnum IN[1..7];
250   WRITELN(intnum,' factoral is: ',
       factoral(intnum));
260 END. (* program recursiv *)
```

Note that in the program, the REPEAT loop continues until a number from 1 through 7 is entered.

Another example of a recursive routine is shown in the following program in which text is entered from the keyboard and displayed in reverse order.

```
100 PROGRAM transpos;
110 PROCEDURE chario;
120  VAR charactr:CHAR;
130  BEGIN (* procedure chario *)
140    READ(charactr);
150     IF charactr<>' '
160    THEN chario;
170  WRITE(charactr);
180  END; (* chario *)
190 BEGIN (* program transpos *)
```

(program continued on next page)

```
200   WRITELN('Enter word followed by a blank: ');
210   chario;
220   WRITELN;
230 END. (* program transpos *)
```

In this program, the recursive procedure chario is used to read and display some entered characters. When chario is first called, a character is read into charactr. If the character is not a blank, chario is called again. The character read this time is stored in the variable that is part of the first recursive call to chario. The following paragraphs describe how the characters of the string 'hello' are read and saved.

In the first call to the procedure chario, the character 'h' is stored in the variable charactr that is local to this call of chario (for simplicity, this variable is referred to as charactr-1).

Because the character was not a blank, another call to chario is made and the character 'e' is read. This character is stored in the variable charactr that is local to this call of chario (charactr-2). Chario is called to store the characters 'l', 'l', and 'o' in the variables charactr-3, charactr-4, charactr-5, local to the 3rd, 4th, and 5th calls to chario, respectively.

The 5th call to chario calls chario the 6th time to read another character when the blank character is read. This 6th call to chario is now finished and thus returns to its caller (the 5th call to chario).

The WRITE statement follows this call. The character in the variable charactr that is local to the 5th call is the character 'o'. This 5th call is complete and control returns to the 4th call, which executes the WRITE statement and displays the letter 'l'. Each call to chario returns to the previous call until the letters 'l', 'e', and 'h' have been displayed.

After the first call is finished, control is returned to the main program block and the program terminates.

Review Chapter 7

1. A procedure contains two parts. They are

____

____

2. A statement in a program body that contains the name of a procedure is known as a

   ____ .

3. Write a program that displays the following. Use two procedures to define the output.

```
* * * * Concept * * * *

           *
        *     *
     *           *

* * * Summation * * *
           *
          ***
        *******

* * * * Concept * * * *

           *
        *     *
     *           *

* * * Summation * * *
           *
          ***
        *******
```

4. If the following declarations are made in the program example and the procedure ex1, does the procedure use the value of the global variable duplicat in line 300?

```
100 PROGRAM example;
110 VAR duplicat:REAL;
.
.
.
290 PROCEDURE ex1;
300 VAR a,duplicat:REAL;
310 BEGIN
.
.
.
350 a:=duplicat;
```

Introduction

Pascal programs use input and output statements to communicate with the keyboard, the display, and peripheral devices such as a printer. Input and output statements transfer data to and from a file (a collection of data that has a declared name).

When the computer is sending data to or receiving data from an external device, the I/O display indicator is turned on. You cannot use the keyboard at this time (including the **OFF** key). If a file is open when you press the **OFF** key, the file is automatically closed before the computer is turned off.

When you use a LIST, OLD, or SAVE command, the Pascal interpreter allows you to use a printer or a mass-storage device by referencing the device's code number. Saving a program and executing a stored program are discussed in this chapter under "Program Storage and Execution."

When you run a Pascal program, the interpreter automatically opens three files for your program. These files are defined to be files of type INTERACTIVE and are called INPUT, OUTPUT, and KEYBOARD.

INPUT	refers to the input device. If you do not specify otherwise in an input statement, the interpreter uses the console device to obtain data. The console is defined to be the display and keyboard combined. When an input statement uses the console device, any character typed at the keyboard is displayed.
KEYBOARD	refers to the input device. If you specify that an input statement uses KEYBOARD, any character typed at the keyboard is not displayed and therefore, the cursor does not move.
OUTPUT	refers to the output device. If you do not specify otherwise in an output statement, the interpreter uses the console device to display data.

Data Format

When a Pascal program stores, updates, or writes data to a peripheral device, the data is recorded in ASCII characters to a file. All files processed by Pascal statements must be in ASCII format.

Data Records

When an input or output statement accesses a file, it retrieves or stores a record of data. A record consists of fields of data. The value of each variable in an output statement is written in a field of a record.

The maximum length of a record varies with the peripheral device being used. Pascal uses a default specification for each device. For a printer device, the maximum record length is 80 bytes.

When a WRITELN statement is executed, the values are written to an output buffer with an end-of-line marker that sets the end of the record. The length of a record written by WRITELN is the number of characters written by WRITELN, providing the number of characters is not greater than the maximum length allowed for the peripheral device. If WRITELN attempts to write a record longer than one allowed for the device, the record is repeatedly broken into records that are the maximum allowed until the last record has a length of the maximum or less.

When a WRITE statement is executed, the values are written to an output buffer. The WRITE statement allows the next output statement to write its fields of data after the previous statement's data. The data is not actually transferred to the device until either the maximum number of characters allowed for the record length of the device is reached in the buffer or until a WRITELN, READ, or READLN statement is executed.

File Organization

With TI-74 Pascal, files are accessed sequentially; data must be read in sequence from beginning to end.

Initializing a Mass-Storage Medium

If you are using a mass-storage device other than a cassette recorder, you must use the FORMAT command to initialize a new medium before you can use it. For example, the command

```
FORMAT 110
```

initializes or formats the medium on peripheral device 110. Note that if you format a medium that already has data on it, the existing data is lost. Refer to the peripheral manuals for information on formatting other media.

Deleting a File

The DEL command can be used to delete a file from a mass-storage device. For example, the command

```
DEL '1.payroll'
```

deletes the file `payroll` on device 1.

File-Processing Keywords

Pascal provides the following statements and declarations for file handling.

File Declaration

If you want to input or output data from a device other than the console, you must declare an identifier for the device file and the type of the file in a VAR declaration. In TI-74 Pascal, the type of a file must be defined to be type TEXT.

For example,

```
VAR printer:TEXT;
```

declares the file-identifier `printer` to be a file of the predefined type TEXT.

A TEXT file consists of a sequence of lines, each of which is a record. Each line consists of a sequence of characters terminated by an end-of-line marker. After the last end-of-line marker is an end-of-file marker.

Opening and Closing a File

Pascal provides the intrinsic procedures RESET and REWRITE to open a file and the intrinsic procedure CLOSE to save or delete a file. RESET and REWRITE open a file and specify the identifier that is used in the program to access the file. If RESET or REWRITE attempts to open a file that is already open, an error occurs.

In this manual, the identifier associated with an open file is called file-identifier. A file-identifier must be declared in the program as type TEXT.

The RESET Procedure
RESET is used to open an existing file for input. The file is positioned to the first record. For example, the statements

```
VAR file1:TEXT;
BEGIN
RESET(file1,'7.address');
```

declare the file-identifier file1 as a TEXT file, open the file named address located on device 7 with the file-identifier file1, and position the file to the first record.

You can also use RESET to position a file back to the beginning of the file, but you must first close the file. The statements

```
130 VAR file1:TEXT;
140     x:REAL;
150 BEGIN
160   RESET(file1,'1.address');
170   READLN(file1,x);
180   CLOSE(file1);
190   RESET(file1,'1.address');
```

open the file address on device 1 with the file-identifier file1, read one value from the file, close the file, and then position the file back to the first field in the first record.

After you close a file, you can use the file-identifier to open another file as well as open the file with another file-identifier.

If you attempt to open a write-only device such as a printer with RESET, an error occurs.

The REWRITE Procedure
REWRITE is used to open a file for output. If the file does not already exist, REWRITE creates a file containing only the end-of-file marker. If the file already exists, REWRITE deletes the existing file and creates a new file containing only the end-of-file marker.

For example, the statements

```
130 VAR file1:TEXT;
140 BEGIN
150 REWRITE(file1,'7.address');
```

open a file with a file-identifier of file1 on device 7. If the file address already exists, REWRITE deletes the file address and creates a new file address containing only an end-of-file marker.

If you attempt to open a read-only device with REWRITE, an error occurs.

The CLOSE Procedure

CLOSE is used to close an open file. After the file is closed, the file-identifier used to open the file is then no longer associated with it. Certain options may be included in a call to CLOSE, as shown below.

Filename opened with:	RESET	REWRITE
CLOSE(file-identifier)	closes the file	deletes the file
CLOSE(file-identifier,LOCK)	closes the file	closes the file
CLOSE(file-identifier,PURGE)	deletes the file	deletes the file

Generally, to close and save a file, you should use a CLOSE as shown below.

```
CLOSE(file_identifier,LOCK)
```

Note that if you attempt to close a file for a write-only device (such as a printer) with CLOSE(file-identifier) or CLOSE(file-identifier,PURGE), an error occurs.

When a Pascal program finishes normal execution, the interpreter automatically closes any open files, thus preserving the contents of the files. Files already closed in a program are not affected.

File Input and Output

In Pascal, the EOLN and the EOF functions are used to determine the status of the end-of-line or end-of-file character. The routines READ, READLN, WRITE, and WRITELN are provided for accessing elements of a file.

The EOLN and EOF Functions

The EOLN function is used to test the status of the end-of-line marker. For a TEXT file, the EOLN function returns a TRUE result if the next character to be read is the end-of-line character. For an INTERACTIVE file (INPUT or KEYBOARD) the EOLN function returns a TRUE result if the end-of-line character was the last character read.

The EOF function enables you to test the status of the end-of-file marker. For a TEXT file, the EOF function returns a TRUE result if the next character to be read is the end-of-file marker. Note that the EOF function cannot be TRUE at the end of the last line; the EOF function is TRUE after the last end-of-line character has been read. Therefore, a READLN statement should precede an EOF test.

For an INTERACTIVE file, there is no end-of-file marker.

File Input with READ and READLN
The READ and READLN statements can be used to read values from a file by preceding the list of variables with the file-identifier. If no file-identifier appears before the list of variables, the interpreter assumes that input is from the file INPUT or the keyboard. READ and READLN read the different data types from a file the same way they read values from the keyboard (except as noted above for the end-of-file marker and end-of-line marker).

A file-identifier listed in READ or READLN must be defined as a TEXT file. A Boolean type variable cannot appear in a READ or READLN.

The following program transfers a line from file `file1` to `file2`. Note that the EOLN function is used to determine when the end of the line has been reached. EOLN is TRUE when the input buffer pointer is pointing to the end of the line and FALSE otherwise.

```
100 PROGRAM linetran;
110 VAR ch:CHAR;
120      file1,file2:TEXT;
130 BEGIN
140   RESET(file1,'1.data1');
150   REWRITE(file2,'2.data2');
160   WHILE NOT EOLN(file1) DO
170     BEGIN
180       READ(file1,ch);
190       WRITE(file2,ch);
200     END; (* while *)
210   WRITELN(file2);
220   CLOSE(file1,LOCK);
230   CLOSE(file2,LOCK);
240 END. (* linetran *)
```

The program on the next page transfers an entire file from `file1` to `file2`. Note that after the last character on a line is read, the input cursor is pointing to the end-of-line marker. A READLN should be executed to move the cursor to the first character in the next line.

The end-of-file condition is TRUE only after the last end-of-line character is read. Therefore, a test for an end-of-file condition should be made after a READLN has been executed. When the end-of-file condition becomes TRUE, the program ends.

```
100 PROGRAM filetran;
110 VAR ch:CHAR;
120     file1,file2:TEXT;
130 BEGIN
140   RESET(file1,'1.data1');
150   REWRITE(file2,'2.data2');
160   WHILE NOT EOF(file1) DO
170     BEGIN
180       WHILE NOT EOLN(file1) DO
190         BEGIN
200           READ(file1,ch);
210           WRITE(file2,ch);
220         END; (* while not EOLN *)
230       READLN(file1);
240       WRITELN(file2);
250     END; (* while not EOF *)
260 END. (* filetran *)
```

The following program requires that the correct code be entered from the keyboard before the program will run. In this example, the code **394$** must be entered. The code is entered from the file KEYBOARD and therefore not displayed. If the correct code is not entered, a programmed HALT occurs. The program prompts to determine whether the donations made last year to charity are to be printed. The program then accepts the total amount of money to be donated to charity and lists the donations to a printer. When the total has been exceeded, a message is displayed.

```
100 PROGRAM donation;
110 VAR numcomp:INTEGER;
120     cause,code:STRING;
130     totmoney,money,moneylef:REAL;
140     f1,fpr:TEXT;
150     ch:CHAR;
160 PROCEDURE getamoun;
170   BEGIN
180     WRITE('Enter recipient: ');
190     READLN(cause);
200     WRITE('Enter amount to donate: ');
210     READLN(money);
220   END; (* getamount *)
```

```
PROCEDURE lastyear;
  BEGIN {$w+}
    WRITELN('Load tape: ');
    RESET(f1.'1.donate');
    REWRITE(fpr,'20');
    WHILE NOT EOF(f1) DO
      BEGIN
        READLN(f1,cause);
        READLN(f1,money);
        WRITE(fpr,cause:25,money:14:2);
      END;
    CLOSE(f1,LOCK);
    CLOSE(fpr,LOCK);
  END; (* lastyear *)
BEGIN
  numcomp:=1;
  WRITE('Enter code: ') {$w-};
  READLN(KEYBOARD,code);
  IF code<>'394$'
    THEN HALT
   ELSE
   WRITE('Last year''s list? (Y or N)');
  READLN(ch);
  IF (ch='y') OR(ch='Y')
    THEN lastyear;
      WRITE('Enter total donation: ');
      READLN(totmoney);
      moneylef:=totmoney;
      REWRITE(f1,'1.year');
      REWRITE(fpr,'20');
      getamoun;
      WHILE moneylef-money>=0.0 DO
        BEGIN
          WRITELN(fpr,cause:25,money:14:2);
          WRITELN(f1,cause:25);
          WRITELN(f1,money:14:2);
          moneylef:=moneylef-money;
          getamoun;
          numcomp:=numcomp+1;
        END; (* while money left *)
  IF numcomp=1 {$w+}
    THEN WRITELN('More than ',totmoney:10:2,
      ' given to 1 cause')
    ELSE WRITELN('Total donation > ',
          totmoney:10:2);
END. (* donation *)
```

The PAGE Procedure

The procedure PAGE is used to write a form feed (page advance) character to a file. If the file is the display, the display is cleared and the cursor moved to column 1. The PAGE procedure is not supported by the PC-324.

In the program below, a line of output is sent to a printer and the PAGE procedure then sends a form feed character to the device. The printer skips to the start of the next page and then prints the second line of output.

```
100 PROGRAM print;
110 VAR f1:TEXT;
120 BEGIN
130   REWRITE(f1,'20');
140   WRITELN(f1,'first line');
150   PAGE(f1);
160   WRITELN(f1,'second line');
170   CLOSE(f1,LOCK);
180 END. (* print *)
```

I/O Status

When an I/O error occurs during execution of a Pascal program, the program is usually aborted. For example, if a program attempts to read from a mass-storage device and the correct medium is not loaded, the interpreter aborts the program. An interpreter option, however, allows you to check an input/output operation and then take appropriate action in the program.

Before attempting an input/output operation, you turn off the automatic input/output check by including a `$i-` immediately after an opening comment delimiter. For example, when the interpreter encounters the comment

```
{$i-}
```

automatic I/O checking is suspended. To turn the checking back on, enter the comment

```
{$i+}
```

at the point where automatic I/O checking is to be resumed.

After you have turned the I/O check off, the program can check the status of I/O operations by calling IORESULT. If you are reading input from a mass-storage device, you can check whether the correct medium is loaded with a routine such as that shown in the program on the next page.

```
100 PROGRAM iocheck;
110 CONST badtape=3;
120 VAR file1:TEXT;
130       iocode:INTEGER;
140       a,b,c:REAL;
150 BEGIN
160(*$i- turn off automatic I/O checking *)
170    REPEAT
180       RESET(file1,'1.data');
190       iocode:=IORESULT;
200       IF iocode=badtape
210         THEN WRITELN('Load correct tape:
                     then press ENTER');
220       IF(iocode<>0) AND(iocode<>badtape)
230         THEN HALT;
240    UNTIL iocode=0;
250    (*$i+ turn on automatic I/O checking *)
260    READLN(file1,a,b,c);
270    WRITELN(a:5,b:5,c:5);
280 END. (* program iocheck *)
```

Refer to appendix I in the *TI-74 Learn Pascal Reference Guide* for the I/O status codes returned by IORESULT.

Review— Chapter 8

1. The three predefined INTERACTIVE files are

   ____

   ____

   ____

2. The maximum length of a record is dependent upon the __________ being used.

3. All files in TI-74 Pascal must be organized and accessed ______________________________.

4. At the end of each record in a TEXT file is an __________ marker.

5. In the statement

   RESET(file1,'1.comps');

   the file-identifier is ______________________
   the device-code is ______________________
   the name of the file on the device is ______________

6. What is the error in the following statements?

```
150 REWRITE(file1,'7.account');
160 READ(file1,a);
```

7. Which option is used with CLOSE to ensure that a file is always saved when it is closed?

   ____

8. For an INTERACTIVE file, the EOLN is true when ____________________.

9. The EOF function is true on a TEXT file when __________.

10. Write a program that displays a multiplication table of the integers from 1 through 12. Use a procedure to display lines between the rows of values and at the top and bottom.

11. Write a program that prints the characters corresponding to ASCII codes 31 through 127 to the printer whose device code is 20.

12. Write a program that reads a TEXT file stored on device 7 and prints the data on device 20. The printed data should be double-spaced.

13. Use the program in 12 and modify it so that if an asterisk is read, the printer sends a form feed character.

Answer Key

This answer key contains the answers to the questions in the Reviews at the ends of chapters 2 through 9.

Chapter 2

1. run "pascal"
2. bye
3. line number
4. character string
5. apostrophes
6. period after the word END
7. DEL
8. SAVE
9. SAVE "1.myprog" should be written with apostrophes as
   SAVE '1.myprog'
10. OLD '7.myprog'

Chapter 3

1. program heading
   program block
2. BEGIN
   END
3. statements
4. define
5. LABEL
   CONST
   TYPE
   VAR
   PROCEDURE/FUNCTION
6. identifier
7. measure
   account1
   (5percent does not begin with a letter)
   (printheader is truncated to 8 characters)
   (END is a reserved word)
   (sales-tx contains a character other than a letter or a digit)

8. numeric
   character
   string
   Boolean

9. REAL

10. 32767   - 32767

11. two apostrophes

12. The opening comment delimiter (* has a space between the two characters.

13. NUM
    REN

14. 80

15. No semicolon between the two WRITELNs.

16. (*   *)
    {   }

17. Turns off the wait option

18.
```
The answer is 10

The answer is
10
```

19.
```
100 PROGRAM ex19;
110 BEGIN
120   WRITELN('5+5 is ',5+5);
130 END. (* ex19 *)
```

20.
```
100 PROGRAM ex20;
110 BEGIN
120   WRITELN('***The results are listed below***');
130   WRITELN('    x=5');
140   WRITELN('    y=10');
150 END. (* ex20 *)
```

21. END
    HALT
    EXIT

22. semicolon

23. No **BEGIN**
    No period after **END**

hapter 4

1. CONST
   TYPE
   VAR
2. CHAR
   INTEGER
   STRING
   REAL
   BOOLEAN
3.

7.567 E01	no space allowed
.5	no digit to the left of the decimal point
12.	no digit to the right of the decimal point

4. 10
   10
5. a 1234
   b 35.5
   c ,
   d the end
6. 83.545
   -83.5
   -83.55
   -83.5450
   -83.545
   -83.545
   -83.545
   83.545
7.

8	INTEGER
2.5	REAL
5	INTEGER
TRUE	BOOLEAN
16.5	REAL
TRUE	BOOLEAN
1	INTEGER
FALSE	BOOLEAN

8.
```
100 PROGRAM example8;
110 VAR st:STRING;
120 BEGIN
130    WRITE('String: ') {$w-};
140    READLN(st);
150    WRITELN('String length is ',
        LENGTH(st)) {$w+};
160    WRITELN(SCAN(LENGTH(st),='z',st));
170 END. (* example8 *)
```

9.
```
100 PROGRAM example9;
110 VAR st1,st2:STRING;
120 BEGIN
130   WRITE('String1: ') {$w-};
140   READLN(st1);
150   WRITE('String2: ');
160   READLN(st2); {$w+}
170   IF POS(st1,st2) <>0
180     THEN WRITELN('String1 in String2 at ', POS(st1,st2))
190     ELSE IF POS(st2,st1) <>0
200     THEN WRITELN('String2 in String1 at '
          ,POS(st2,st1))
210     ELSE WRITELN('No substrings exist -POS = 0');
220 END. (* example9 *)
```

10.
```
100 PROGRAM ex10;
110 VAR code:INTEGER;
120       ch:CHAR;
130 BEGIN
140   WRITE('Enter integer: ') {$w-};
150   READLN(code);
160   WRITE('Enter character: ');
170   READLN(ch); {$w+}
180   WRITELN('Predecessor of ',code,'  = ',PRED(code));
190   WRITELN('Successor of ',code,' = ',SUCC(code));
200   WRITELN('Predecessor of ',ch,' = ',PRED(ch));
210   WRITELN('Successor of ',ch,' = ',SUCC(ch));
220 END. (* ex10 *)
```

11.
```
100 PROGRAM ex11;
110 VAR celsius:REAL;
120 BEGIN
130   WRITE('Enter deg C: ') {$w-};
140   READLN(celsius) {$w+};
150   WRITELN(celsius,' deg C. = ',
160        celsius*9/5+32,' deg F.')
170 END. (* ex11 *)
```

Chapter 5

1. FOR
   WHILE
   REPEAT

2. IF
   CASE

3. true

4. true

5.
```
100 PROGRAM example5;
110 VAR count,index,least,greatest:INTEGER;
120     weight,total,average:INTEGER;
130 BEGIN
140   least:=MAXINT;
150   greatest:=0;
160   total:=0;
170   REPEAT
180     WRITE(' Enter # in group: ') {$w-};
190     READLN(count);
200   UNTIL count>0;
210   FOR index:=1 TO count DO
220     BEGIN
230       WRITE(' Weight #',index,' : ');
240       READLN(weight);
250       IF weight<least
              THEN least:= weight;
260       IF weight>greatest
              THEN greatest:=weight;
270       total:=total+weight;
280     END; {$w+}
290   WRITELN('Least weight is: ',least);
300   WRITELN('Greatest weight is: ',
                 greatest);
310   WRITELN('Average weight is: ',
                 total/count);
320 END. (* example5 *)
```

6.
```
100 PROGRAM example6;
110 VAR count,least,greatest:INTEGER;
120     weight,total,average:INTEGER;
130 BEGIN
140   least:=MAXINT;
150   greatest:=0;
160   total:=0;
170   count:=1;
180   WRITELN('Enter weights, enter 0 to stop');
190     WRITE('Weight #',count,': ') {$w-};
200     READLN(weight);
210   REPEAT
220     IF weight<least THEN least:=weight;
230     IF weight>greatest THEN greatest:=weight;
240     total:=total+weight;
250     count:=count+1;
260     WRITE('Weight #',count,': ');
270     READLN(weight);
```

```
280    UNTIL weight=0; {$w+}
290    WRITELN('Least weight is: ',least);
300    WRITELN('Greatest weight is: ',greatest);
310    WRITELN('Average weight is: ',total/count-1);
320 END. (* example6 *)
```

7.
```
100 PROGRAM example7;
110 VAR count,least,greatest:INTEGER;
120     weight,total,average:INTEGER;
130 BEGIN
140    least:=MAXINT;
150    greatest:=0;
160    total:=0;
170    count:=1;
180    WRITELN('Enter weights, neg. # to stop');
190    WRITE('Weight #',count,': ') {$w-};
200    READLN(weight);
210    WHILE weight>0 DO
220       BEGIN
230          IF weight<least THEN least:=weight;
240          IF weight>greatest THEN greatest:=weight;
250          total:=total+weight;
260          count:=count+1;
270          WRITE('Weight #',count,': ');
280          READLN(weight);
290       END; {$w+}
300    WRITELN('Least weight is: ',least);
310    WRITELN('Greatest weight is: ',greatest);
320    WRITELN('Average weight is: ',total/count-1);
330 END. (* example7 *)
```

8. semicolon before ELSE

9.
```
100 PROGRAM example9;
110 LABEL 9999;
120 VAR count,index:INTEGER;
130 BEGIN
140    WRITELN('Enter 12 integers (1-12):');
150    FOR index:=1 TO 12 DO
160       BEGIN
170          WRITE('#',index,': ') {$w-};
180          READLN(count); {$w+}
190          IF(count<1) OR (count>12)
200             THEN
210                BEGIN
220                   WRITELN('Invalid entry');
230                   GOTO 9999;
240                END;
```

```
250         CASE count OF
260             1:WRITELN('January');
270             2:WRITELN('February');
280             3:WRITELN('March');
290             4:WRITELN('April');
300             5:WRITELN('May');
310             6:WRITELN('June');
320             7:WRITELN('July');
330             8:WRITELN('August');
340             9:WRITELN('September');
350             10:WRITELN('October');
360             11:WRITELN('November');
370             12:WRITELN('December');
380         END; (* case *)
390       END; (* FOR-loop *)
400     9999:WRITELN('Finished');
410 END. (* example9 *)
```

Chapter 6

1. INTEGER
   CHAR

2. CHAR
   INTEGER

3. None—a type declaration defines the identifier sales as an array type. A VAR declaration is used to allocate storage for arrays.

4.
```
100 PROGRAM inout;
110 VAR ch:ARRAY[1..80] OF CHAR;
120     count,index:INTEGER;
130 BEGIN
140   index:=0;
150   WRITE('Enter word: ') {$w-};
160   REPEAT
170     index:=index+1;
180     READ(ch[index]);
190   UNTIL ch[index]=' ';
200   FOR count:=index-1 DOWNTO 1 DO
210     WRITE(ch[count]);
220   {$w+} WRITELN;
230 END. (* inout *)
```

5.
```
100 PROGRAM example5;
110 CONST maxnum=15;
120 TYPE intarray=ARRAY[1..maxnum] OF INTEGER;
130 VAR sort:intarray;
```

```
    index,fixed,temp:INTEGER;
BEGIN
  FOR index:=1 TO maxnum DO
    BEGIN {$w-}
      WRITE('Enter integer #',index,': ');
      READLN(sort[index]);
    END; {$w+}
  FOR fixed:=2 TO maxnum DO
    BEGIN
      FOR index:=maxnum DOWNTO fixed DO
       BEGIN
         IF sort[index]>sort[index-1]
           THEN
             BEGIN
               temp:=sort[index-1];
               sort[index-1]:=sort[index];
               sort[index]:=temp;
             END; (* swap adjacent elements *)
       END; (* one pass through array *)
    END; (* all elements are sorted *)
  WRITELN('Descending order of integers: ');
  FOR index:=1 TO maxnum DO
  WRITELN('Integer #',index,': ',sort[index]);
END. (* example5 *)
```

6. valid
   valid
   invalid—a string variable cannot be assigned to a packed array of char.
   invalid—a string variable cannot be assigned to a packed array of char.
   invalid—the string constant is too long to be assigned to the array.
   valid
   invalid—the two arrays are not the same length.
   invalid—the two arrays are not the same length.
   valid
   invalid—illegal number of subscripts with array pac4.

Chapter 7

1. statements
   declarations

2. procedure call

3.
```
PROGRAM example3;
PROCEDURE concept;
  BEGIN
  WRITELN('* * * * Concept * * * *');
```

```
140   WRITELN;
150   WRITELN('           *');
160   WRITELN('        *     *');
170   WRITELN('     *           *');
180   WRITELN;
190 END; (* procedure concept *)
200 PROCEDURE sum;
210 BEGIN
220   WRITELN('* * * Summation * * *');
230   WRITELN('          *');
240   WRITELN('         ***');
250   WRITELN('       *******');
260   WRITELN;
270 END; (* procedure concept *)
280 BEGIN (* program body *)
290   concept;
300   sum;
310   concept;
320   sum;
330 END. (* example3 *)
```

4. No, the procedure ex1 uses the local variable duplicat.

Chapter 8

1. INPUT
   KEYBOARD
   OUTPUT

2. device

3. sequentially

4. end-of-line

5. file1
   1
   comps

6. The READ in line 160 is attempting to read from a file opened for output.

7. LOCK

8. the last character read was the end-of-line marker.

9. the next character to be read is the end-of-file marker.

```
10. 100 PROGRAM table;
    110 VAR count1,count2:INTEGER;
    120     pr:TEXT;
    130 PROCEDURE lines;
    140   BEGIN
    150     WRITELN(pr,'------------------------------------
            ------------------');
    160   END; (* lines *)
    170 BEGIN
    180   REWRITE(pr,'50');
    190   WRITELN(pr);
    200   WRITELN(pr,'Multiplication table for integers: 1
            to 12':46);
    210   lines;
    220   WRITELN(pr);
    230   WRITE(pr,'  ');
    240   FOR count1:=1 TO 12 DO
    250     WRITE(pr,count1:4);
    260   WRITELN(pr);
    270   lines;
    280   FOR count1:=1 TO 12 DO
    290     BEGIN
    300       WRITE(pr,count1:2);
    310       FOR count2:=1 TO 12 DO
    320         WRITE(pr,count1*count2:4);
    330       WRITELN(pr);
    340       lines;
    350     END; (* count1 all rows *)
    360   lines;
    370 END. (* table *)
```

```
11. 100 PROGRAM ex11;
    110 VAR index:INTEGER;
    120     pr:TEXT;
    130 BEGIN
    140   REWRITE(pr,'20');
    150   FOR index:=31 TO 127 DO
    160     WRITELN(pr,CHR(index));
    170 END. (* ex11 *)
```

```
12. 100 PROGRAM ex12;
    110 VAR filein,pr:TEXT;
    120     st:STRING;
    130 BEGIN
    140   RESET(filein,'7.file10');
    150   REWRITE(pr,'20');
    160   WHILE NOT EOF(filein) DO
    170     BEGIN
    180       READLN(filein,st);
    190       WRITELN(pr,st);
    200       WRITELN(pr);
    210     END;
    220 END. (* ex12 *)
```

13.
```
    100 PROGRAM ex13;
    110 VAR filein,pr:TEXT;
    120     ch:CHAR;
    130 BEGIN
    140   RESET(filein,'7.file10');
    150   REWRITE(pr,'20');
    160   WHILE NOT EOF(filein) DO
    170     BEGIN
    180       READ(filein,ch);
    190       IF ch='*'
    200         THEN PAGE(pr);
    210       WRITE(pr,ch);
    220       IF EOLN(filein)
    230         THEN
    240           BEGIN
    250             READLN(filein);
    260             WRITELN(pr);
    270             WRITELN(pr);
    280           END; (* EOL and blank line *)
    290     END; (* end-of-file *)
    300 END. (* ex13 *)
```

Index

A
ABS—64, 69
Absolute value—64, 69
Actual parameter—131, 136
Algorithm—17
Allocation of memory—136, 139
AND—58, 60, 61, 95
Apostrophe—21, 27, 40
Argument—64
Arithmetic operators—50, 51, 53
Array—103
  declaration—103, 107
  parameters—138, 139
  procedures—146
  of arrays—110
  of characters—115
  type—107, 138
ASCII—55, 56, 153
Assignment
  operator—42
  statements—42, 43
ATAN—70
Automatic Power Down™—10

B
Base type—107
BASIC command level—9, 10
BEGIN—19, 23, 87, 92, 93, 99, 121, 123, 125
Binary operator—51, 54
Blocks—122
BOLDFACE—19
Boolean
  constant—21
  functions—73
  operators—57
Boolean-expression—86, 87, 91, 95, 96
BOOLEAN type—42
Brackets—10, 41, 103
BREAK—8, 32
Breakpoint—32, 33
BYE—10

C
CASE—91, 97, 99
CHAR—38, 46
Character
  array—116
  constant—21
  function—71
  operators—55
CHAR type—40, 41, 103
Check input/output—26, 160
CHR—71
CLOSE—154, 156
Closing a file—156
CLR—8, 13, 14, 32
Column—78
Command—18
Comments—25, 26
Compiler—17
Compound statement—83, 87
CON—32
CONCAT—72
Conditional branch statements—82, 91
CONST—19, 36, 41
Constant—20, 21, 36
Constant declarations—36
CONTINUE—32
Control
  statements—82
  variable—83, 84, 86
COPY—72
COS—70
Counter—84
CTL—13
Cursor—28, 47, 78, 152, 158

D
Data
  format—76, 153
  records—153
Debugging a program—32
Decimal notation—75
Declarations—18, 19
  order—19
Default options—27
DEL—14, 154
DELETE—144, 145
Deleting a file—154, 155
Delimeter—20
Device number—152
DIV—51, 60
DOWNTO—84, 86
Dynamic length—41

E
Editing program lines—12, 13
Elements—103, 107, 138
ELSE—91, 92, 93
Empty string—41
END—19, 23, 29
End-of-file marker—154, 155, 156
End-of-line
  character—47, 154, 156
  condition—158
  marker—153, 154, 155, 156, 158
Ending a procedure or function—126
ENTER—11, 28, 32
EOF—73, 156
EOLN—73, 156
Equal to—52, 55, 57, 58
Equality—57
  testing for—63
Error
  codes—31
  handling—30, 31
Errors—12
EXIT—30
Execution—11
  errors—30
  of a stored program—15
EXP—70
Exponent—39, 40, 75
Expression—36, 62

F
Factorial—147, 148
FALSE—42, 52, 57, 73
Field-width specification—74, 75, 76, 78
File
  declaration—154
  handling—152
  type—152
File-identifier—155, 156, 157
File-processing keywords—154
FILLCHAR—144, 146, 147
Flow of control—82
FN—10
FOR—83, 85, 89, 105, 112, 118
Form feed—160
Formal parameters—129
FORMAT—153
Formatted data—74, 76
FORWARD—143
FUNCTION—19
Function—64, 121, 122, 129
  block—125
  body—125
  call—132, 143
  declarations—125
  heading—125

G
Global—138
  identifiers—132
GOTO—100, 101, 126
GOTOXY—78
Greater than—52, 55, 57, 58
Greater than or equal—52, 55, 57, 58

H
HALT—29, 30, 126, 158
High-level languages—17
Histogram—112

I
i (input/output check)—26, 27, 160
Identifier—20, 131, 143
IF—91, 92, 93
Imperative—30, 64, 66
IN—53, 58
Indentation—23, 24
Index—103
Index type—104
Indexing—41
Initializing
  a mass-storage medium—153
  the Pascal system—10
INPUT—152, 157
Input—18
  buffer—47, 48, 153
  statements—44, 48, 152, 155, 156
INSERT—144, 145
Installing a cartridge—9
INTEGER—38
Integer functions—64
Integer operators—50, 51
Integer-to-real conversion—39, 44
INTEGER type—38, 46, 103

INTERACTIVE—152, 157
Interpreter—17, 24, 25, 103, 129, 136, 152, 156, 157
Interpreter options—25, 26, 27
Intrinsic procedures—144
I/O
  STATUS—160
  display indicator—152
IORESULT—26, 27, 160, 161
Italics—21

K
KEYBOARD—152, 158
Keywords—10, 20

L
LABEL—19, 100
Leading blanks—45
LENGTH—68
Length of strings—40, 41
Less than—52, 55, 57, 58
Less than or equal—52, 55, 57, 58
Lexicographical—56
Line
  length—24
  numbering—22
  renumbering—23
LIST—10, 152
Listing a program—11
LN—70
LOCK—156
LOG—70
Logical operators—50, 53, 56, 57, 58
Loop—82, 89
Loop control—85
Lowercase characters—20, 56

M
Magnitude—39, 45, 46
Mantissa—39, 75
Maximum field width—78
Maximum length—40, 153
MAXINT—21, 38
MEMAVAIL—66, 68, 145
Memory functions—65, 66
MOD—51, 60
MOVELEFT—144, 146, 147
MOVERIGHT—144, 146, 147
Multi-type functions—73
Multiple-line statements—24
Multiple-statement lines—24

N
Nested
  IF—93
  loops—89, 112
NEW—11, 15
NEW ALL—11, 15
NOT—58, 60
Not equal to—52, 55, 57, 58
Null string—41
NUM—22
Numeric constants—21
Numeric functions—64
Numerical accuracy—63

O
ODD—73
OLD—15, 152
One-dimensional—105
Opening a file—154
Operator precedence—60
Operators—50
OR—58, 60, 95
ORD—69
Order of precedence—95
Ordinal—63, 97
OUTPUT—152
Output—18
  buffer—153
  statements—18, 27, 28, 152
Overlay—10

P
PACKED ARRAY OF CHAR—115
Packed arrays—115
PAGE—160
Page advance—160
Parameters—129, 135, 136, 138, 143
Parentheses—44, 61, 62
Pascal System Initialized—10, 11
Pass by reference—136
Peripheral device—152, 153
POS—68
Positioning the cursor—78
Precedence, operator—60

Procedure—121, 122, 129
  block—122, 143
  body—122
  call—123, 132, 143
  declarations—122, 123
  heading—122, 123
PRED—73
PROCEDURE—19
PROGRAM—23
Program
  block—18, 19, 121
  body—19, 121, 123, 129
  divisions of—18
  execution—15, 123
  heading—18
  identifier—18
  lines—12, 22
  storage—12
Prompts—49
PROTECTED—15
Punctuation—24
PURGE—156
PWROFTEN—70

R
Ranking functions—64, 69
READ—44, 47, 48, 153, 157
READLN—44, 47, 48, 153, 157
RUN command—11, 32
Real
  functions—69
  numbers—21
  operators—53
REAL type—39, 46
Record—153
Record length—153
Recursion—147
Reference—138
Relational operators—50, 52, 54, 55, 56, 57, 117
Renumbering program lines—23
REPEAT—83, 86, 87, 88, 149
Repetition statements—82
Reposition a file—155
Reserved symbols—21, 22
Reserved words—20
RESET—154, 155
REWRITE—154, 155
Right-justified—75
ROUND—64
Rounding values—75, 76
Row—78
RUN—10
run "pascal"—10
Running a program—11, 15

S
SAVE—15, 152
SCAN—66
Scientific notation—39, 45, 46, 74
Semicolon—24, 84, 91, 123, 125
Sequential access—153
Set membership—52
SIN—70
SIZEOF—66
SQR—64, 70
SQRT—70
Statements—18, 19
Statements on multiple lines—24
Stepwise refinement—17, 121
STR—144, 145
STRING—37, 46, 115
String
  constant—21
  functions—64, 68, 72
  operators—56
  procedures—144
STRING type—40
Structured programming—17, 101
Subscript—103, 104
SUCC—73
Syntax, Pascal—20
Syntax errors—17

T
Terminating program execution—29
Termination of recursive routines—147
Terms—62
TEXT file—154, 157
Three-dimensional arrays—112, 118
TO—84, 86
Top-down design—17, 121
TRUE—42, 52, 57, 73
TRUNC—64
Truncation—63
Two-character symbols—21, 22

Two-dimensional arrays—109, 118
TYPE declarations—19, 74, 107, 109

U
UCSD Pascal—7
Unary operators—50, 54
UNBREAK—33
Unconditional branch statements—82, 100
Unformatted data—74
UNTIL—86
Uppercase characters—20, 56
User-defined
  functions—135
  identifiers—20
  procedures—135
  type—74

V
Value parameter—136, 138
VAR declaration—37, 74, 107, 109, 129
VAR parameter—129, 132, 136, 137, 138
Variable declarations—37
Variables—36
VERIFY—15

W
w (wait)—26, 27
Wait—26, 27
Warnings—31
WHILE—83, 87, 88
Width—75
WRITE—28, 153
WRITELN—28, 76, 153
Writing a program—11

TEXAS INSTRUMENTS

Printed in Taiwan R.O.C.

1060293-0101