DISK INCLUDED

teach yourself... Pascal

SECOND EDITION

- Provides the fundamentals of Turbo Pascal programming
- Covers advanced features such as dynamic memory management, units, and strings
- Learn object-oriented programming in Turbo Pascal

MIS: PRESS

MARK GOODWIN

teach yourself . . .
Pascal

Second Edition

by Mark Goodwin

A Subsidiary of
Henry Holt and Co., Inc.

First printing.

ISBN 1-55828-328-5

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

MIS:Press books are available at special discounts for bulk purchases for sales, promotions, premiums, fundraising, or educational use. Special editions or book excerpts can also be created to specification.

For details contact: Special Sales Director
MIS:Press
a subsidiary of Henry Holt and Company, Inc.
115 West 18th Street
New York, New York 10011

Project Development Manager: Debra Williams Cauley
Copy Editing and Production: Solstice Communications, Inc., Virginia

Dedication

To Crystal: You hadn't arrived yet when I wrote the first edition of this book, but you have filled all of our lives with love.

Table of Contents

Introduction

In the early 1970s, Niklaus Wirth designed a new programming language called *Pascal.* Mr. Wirth's original intention for the Pascal programming language was to use it as an aide to teach computer programming. Consequently, it is an excellent programming language for the beginning programmer to start with. Even though Pascal is such a good language for beginners, it provides more than enough capabilities for even the most advanced programmers.

Today, there are two basic types of Pascal: *ANSI Pascal* and *Turbo Pascal.* Although ANSI Pascal is supposed to be a standard for all Pascal compilers, Turbo Pascal is by far the most dominant Pascal compiler in use. Therefore, the form of the Pascal programming language that Turbo Pascal adheres to is more of a standard than the ANSI standard. Accordingly, this book is written to teach the reader how to program in the Turbo Pascal dialect of Pascal.

What This Book Will Teach You

This book is intended to teach even a beginning programmer how to program in the Pascal programming language. It covers all of the basic features of Pascal:

- the structure of a Pascal program,
- procedures and functions,
- program flow, data types,
- arrays,
- records, and
- pointers.

It also instructs the reader how to use many of Pascal's advanced features:

- dynamic memory management,
- units,
- strings,
- console input/output, and
- file input/output.

Finally, this book shows how Turbo Pascal can be used to perform object-oriented programming.

What This Book Won't Teach You

This book is not intended to teach you every little detail about Turbo Pascal. That is the job of the reference manuals that come with your compiler. It would be beyond the scope of this book to even try to provide all of the details about Turbo Pascal.

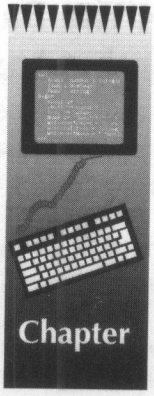
Additionally, this book is not intended to teach you a lot of fancy algorithms (methods for problem solving). That type of instruction is better suited for a more general book on advanced programming. Therefore, it would be inappropriate to bog down the beginning programmer with such things as linked list, B-trees, and sorting methods.

What You Need to Use This Book

To use this book, you need an IBM-PC or compatible computer and Turbo Pascal. You also need a lot of patience and perseverance to become an accomplished Pascal programmer. No matter how well-written this book is, the only way to become a good computer programmer is to write programs, more programs, and even more programs. You'll learn more about programming by successfully tracking down your first bug than I or anyone else could teach you in hours of instruction. Think of this book as a guide or a map. It gets you going in the right direction, but it is totally up to you to arrive at the proper destination. So if things seem a little hazy at first, stick with it. With a little patience, you'll quickly get the hang of Pascal programming.

What You Need to Use This Book

What This Book Won't Teach You



1

A Simple First Program

The first step in understanding the Pascal programming language is to become familiar with the components that make up a Pascal program. Accordingly, this chapter's first lesson acquaints the beginning programmer with all of the Pascal programming language's basic components. After these essential components have been presented, this chapter concludes with a simple first program.

Lesson 1: The Basic Components of the Pascal Programming Language

Keywords

All programming languages use a special set of words to perform certain functions. These special words are called *keywords.*

Some programmers like to refer to keywords as *reserved words.* The use of keywords and reserved words are interchangeable and either term is considered acceptable.

Table 1.1 presents a complete list of the Turbo Pascal keywords. Because a programming language's keywords all serve a special purpose, they can never be used in a program for anything other than their intended purpose.

Table 1.1 *The Turbo Pascal keywords*

absolute	else	in	object	shr
and	end	index	of	string
array	export	inherited	or	then
assembler	exports	inline	packed	to
asm	external	interface	private	type
begin	far	interrupt	procedure	unit
case	file	label	program	until
const	for	library	public	uses
constructor	forward	mod	record	var
destructor	function	name	repeat	virtual
div	goto	near	resident	while
do	if	nil	set	with
downto	implementation	not	shl	xor

Identifiers

As their name implies, *identifiers* are names that are used in a Pascal program to "identify" something in the program. For example, program variables, named constants, procedures, and functions all require a name. Consequently, they are all assigned a unique identifier. When constructing an identifier, the Pascal programmer must keep the following three rules in mind:

1. An identifier's first character must either be a letter or an underscore character (_).
2. Digits (0, 1, 2, 3, 4, 5, 6, 7, 8, and 9) can be used in an identifier.
3. An identifier can be of any length, but only the first 63 characters of the identifier name are significant.

Using the above rules, the following are some examples of valid identifiers:

```
First_Reading
_last_page
count
b32_45a
_32845
```

Table 1.2 lists some examples of invalid identifiers and the reasons why they violate the Pascal identifier rules:

Table 1.2 *Identifiers and their violations*

Identifier	Reason for being invalid
8times	Starts with a digit.
next loop	Space between **next** and **loop**.
name$	**$** is an invalid identifier character.
count*three	* is an invalid identifier character.

Constants

As with all other programming languages, any data found in a Pascal program that never changes its value is called a *constant.* Table 1.3 lists some examples of constants:

Table 1.3 *Examples of constants*

Constant	Type
'Hello'	String constant
'a'	Character constant
#13	Character constant
'Another'	String constant
3.14	Real constant
12345	Integer constant
-32.45	Real constant
456	Integer constant

The Pascal programming language also permits the programmer to name a constant. Once it has been assigned, the constant's name can be freely substituted for the constant's value. The following are some examples of named constants:

```
table_length = 1000;
Authors_Name = 'Mark Goodwin';
```

Variables

Although constants are certainly a useful tool for the Pascal programmer to use, *variables* are an even more useful type of data. As its name implies, a variable is a type of data whose value can be changed throughout the life of a Pascal program. Unlike constants that can be referred to by their literal values, a variable must always have an identifier name.

Operator

The Pascal *operators* are a collection of symbols and keywords that are used to build expressions. Table 1.4 presents a list of the Pascal operators. As this table illustrates, the Pascal operators can be used to perform a wide variety of functions.

Table 1.4 *The Pascal operators*

Operator	Class
@	Unary
NOT	Boolean
*	Multiplication
/	Multiplication
DIV	Multiplication
MOD	Multiplication
AND	Boolean
SHL	Multiplication
SHR	Multiplication
+	Addition
-	Addition
OR	Boolean
XOR	Boolean
=	Relational
<>	Relational
<	Relational
<=	Relational
>	Relational
>=	Relational
IN	Relational

The following are some examples of expressions built from the Pascal operators:

```
3 + 5 / 2
3 <> 4
3 = 5
32.15 / 3.035
```

Statements

Simply put, a Pascal program *statement* is a collection of identifiers, keywords, operators, and constants that perform a specific action. The following are some examples of Pascal program statements:

```
name := 'John Doe';
count : integer;
count := 32 * 55;
```

Note in the above examples how a Pascal statement ends with a semicolon (**;**).

Multiple program statements can be defined as a **begin..end** statement block to express a single idea. The following is an example of a block statement:

```
begin
   count := count + 1;
   Writeln(count);
end
```

The main body of a Pascal program is nothing more than a **begin..end** statement block.

Comments

A Pascal program *comment* is exactly what its name implies. A comment is simply a comment for the programmer's benefit and serves no function as far as the program's execution is concerned. Although they don't affect the program's execution, program comments are a valuable tool for documenting the program. Strategically placed comments can greatly assist in illustrating a program's inner workings. Indeed, many times a program will require modification at some future date. While a program's implementation (a fancy word for how it is written) can seem quite clear when it was originally created, it won't be anywhere near as clear even a week or two down the road. Consequently, comments are one of the Pascal programmer's most valuable tools.

A Pascal comment is created by either enclosing whatever the programmer wants to say in braces or by enclosing the comment in a **(* *)** pair. The following are some examples of Pascal comments:

```
{ open the file and read in the data }
(* close the file if an error has occurred *)
```

Procedures and Functions

One of the most valuable features provided by the Pascal programming language is its support for *procedures* and *functions*. A procedure is a collection of program statements that have been given a name. Furthermore, a procedure or a function can have it's own constants and variables. Essentially, a procedure is nothing more than a mini program. Whenever an executing program encounters a procedure's name, it branches away from the part of the program it is currently executing and execute the procedure's associated statements. A Pascal function is similar to a procedure except that a function returns a value after its associated statements have been executed.

Lesson 2: A First Program

With the basic Pascal components covered in Lesson 1, it is now possible to write our first Pascal program. This first program is presented below in Listing 1.1.

Listing 1.1

```
{ first.pas - A first Pascal program }
program First;

const
      number = 3;

var
      count, result : integer;

function multiply(n1, n2 : integer) : integer;
begin
      multiply := n1 * n2;
end;

begin
      count := 2;
      result := count * number;
      Writeln(result);
      result := multiply(count, number);
      Writeln(result);
end.
```

Although Listing 1.1 is fairly short and really doesn't do much, it serves a very important purpose by illustrating how the Pascal programming language's basic components are brought together in a complete program. To better understand the basic structure of a Pascal program, let's go down through the program a line at a time.

```
{ first.pas - A first Pascal program }
```

is a comment. It simply states the program's file name and provides a brief description.

```
program First;
```

uses the Pascal keyword **program** to assign the identifier **First** as the program's name. Although assigning a program name isn't absolutely necessary, it is generally considered good programming practice to do so.

```
const
```

is the Pascal keyword for defining constants.

```
number = 3
```

assigns the constant value of **3** to identifier **number**.

```
var
```

is the Pascal keyword for defining variable identifiers.

```
count, result : integer;
```

defines two variables, **count** and **result**, with type **integer**.

```
function multiply(n1, n2 : integer) : integer;
```

defines a function called **multiply**. Furthermore, the function expects two **integer** arguments, **n1** and **n2**, and returns a value of type **integer**.

```
begin
```

defines the starting point for the **multiply** function's body.

```
multiply := n1 * n2;
```

multiplies the function's arguments and assigns the result as the return value.

```
end;
```

defines the end of the **multiply** function's body.

```
begin
```

defines the starting point for the program's main body. This is where the program starts executing.

```
count := 2;
```

assigns an initial value, **2**, to the variable **count**.

```
result := count * number;
```

multiplies the variable **count** by the constant **number** and assigns the result to the variable **result**.

```
Writeln(result);
```

displays **result**'s value.

```
result := multiply(count, number);
```

calls the function **multiply**, which simply multiplies **count** by **number**. The result of the function call is assigned to the variable **result**.

```
Writeln(result);
```

displays **result**'s value.

```
end.
```

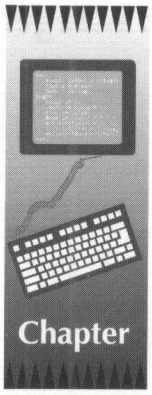
defines the end of the program's main body. Note that a period follows the **end** keyword and not a semicolon. As Lesson 1 stated, a semicolon is used to signify the end of a statement. However, it is always necessary to use a period to signify the end of the program.

Besides showing how the various components of Pascal are used in an actual program, Listing 1.1 also illustrates how the use of white space (spaces, tabs, and double spaced lines) can make a program more readable. You should note that the use of white space is strictly optional. Nevertheless, a program will be almost illegible without at least a minimal amount of white space. For example, Listing 1.2 presents the program **first.pas** stripped of all of its unnecessary white space.

Listing 1.2

```
{ first.pas - A first Pascal program }program
First;const number = 3;

var count, result : integer;function multiply(n1, n2 :
integer) : integer;

begin multiply := n1 * n2; end;begin count := 2;result
:= count * number;

Writeln(result);result := multiply(count,
number);Writeln(result);end.
```

It almost goes without saying that Listing 1.2 is a mess and the version presented in Listing 1.1 is clearly superior. Consequently, it is traditional to write Pascal programs with a fair degree of white space.



2

Predefined Data Types

In Chapter 1, you learned that the Pascal programming language supports two very distinct forms of data: constants and variables. Because the types of data a Pascal program is called upon to handle can vary a great deal, the Pascal programming language comes equipped with a very rich set of predefined data types. To show how Pascal can meet the needs for almost any data handling requirements, this chapter takes a detailed look at all of Pascal's predefined data types.

Lesson 3: Integers

The most basic of the Pascal data types are called *integers*. Simply put, an integer data type can represent whole numbers. The following are some examples of integer constants:

```
32457
-43
0
167
-2335678
$FF
```

You may be wondering what the constant **$FF** is in the above example. The integer $FF is the way the number 255 is represented using the hexadecimal number system. The hexadecimal number system is base 16 and is represented by the digits **0..9** and the letters **A..F** or **a..f.** Table 2.1 clearly displays how numbers are represented by the hexadecimal number system.

Table 2.1 *The hexadecimal number system (base 16)*

Digit	Represents
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
A	10
B	11
C	12
D	13
E	14
F	15

Because the hexadecimal number system is base 16, it's easy to determine that the constant **$FF** is **255** by performing the following calculation:

```
F * 16 + F = 255 or 15 * 16 + 15 = 255
```

Because a small whole number, such as the number **2**, doesn't require as much memory to store as a larger whole number, such as the number **356678**, Pascal offers five very distinct integer data types: **ShortInt**, **Byte**, **Integer**, **Word**, and **LongInt**. Table 2.2 shows the range of numbers these five integer data types can represent.

Table 2.2 *The Pascal integer types*

Data type	Range of values	Size in bytes
ShortInt	-128 to 127	1
Byte	0 to 255	1
Integer	-32768 to 32767	2
Word	0 to 65535	2
LongInt	-2147483648 to 2147483647	4

As Table 2.2 illustrates, a **LongInt** takes two times the amount of memory as an **Integer** and four times the amount of memory as a **ShortInt**. Therefore, the efficient Pascal programmer always strives to use the smallest possible data type. For example, an integer variable that never holds a value less than 0 or greater than 255 should be defined as a **Byte** variable instead of as an **Integer**, **Word**, or **LongInt** variable. Not only do the smaller data types require a great deal less memory than their larger counterparts, the computer can perform operations, such as addition and subtraction, on the smaller data types at much greater speeds.

Table 2.3 illustrates the format for defining integer variables. As this table shows, you can define more than one variable per statement by separating the variable identifiers with commas. The following are some examples of integer variable definitions:

```
number : Integer;
small_number : Byte;
offset : Word;
AccountNumber, AccountBalance : LongInt;
```

Table 2.3 *Defining an integer variable*

Var	
identifier : integer data type;	
identifier , identifier : integer data type;	
Where:	
identifier	is the variable's name.
integer data type	is **ShortInt**, **Byte**, **Integer**, **Word**, or **LongInt**.

In Chapter 1, you saw how constants can be named. The Pascal programming language also supports *typed constants*. Although the value of a Pascal named constant never changes, a typed constant's value can be changed. Basically, a typed constant acts like a variable with an initial value. Table 2.4 illustrates how typed integer constants are defined. Unlike variable definitions, you can only define one constant per definition statement. The following are some examples of typed integer constant definitions:

```
top_row : Integer = 0;
bottom_row : Integer = 24;
left_column : Byte = 0;
right_column : Byte = 79;
CashAccount : LongInt = 100000;
```

Table 2.4 *Defining a typed integer constant*

Const	
identifier : integer data type = constant;	
Where:	
identifier	is the constant's name.
integer data type	is **ShortInt**, **Byte**, **Integer**, **Word**, or **LongInt**.
constant	a constant value or expression.

To illustrate the use of Pascal's integer data types, Listing 2.1 presents a short program, which defines a number of integer variables, constants, and displays their assigned values.

Listing 2.1

```
{ list2-1.pas - Define and display a variety of integers }
program integers;

const
      short_const : shortint = -1;
      byte_const : byte = $3E;
      integer_const : integer = 3245;
      word_const : word = 45667;
      longint_const : longint = 1000000;

var
      short_var : shortint;
      byte_var : byte;
      integer_var : integer;
      word_var : word;
      longint_var : longint;

begin
      short_var := 22;
      byte_var := 254;
      integer_var := -5563;
      word_var := $2224;
      longint_var := -32;
```

Listing 2.1—Continued

```
        writeln('short_const    = ', short_const);
        writeln('byte_const     = ', byte_const);
        writeln('integer_const  = ', integer_const);
        writeln('word_const     = ', word_const);
        writeln('longint_const  = ', longint_const);
        writeln('short_var      = ', short_var);
        writeln('byte_var       = ', byte_var);
        writeln('integer_var    = ', integer_var);
        writeln('word_var       = ', word_var);
        writeln('longint_var    = ', longint_var);
end.
```

Lesson 4: Real Numbers

Although Pascal's integer types are quite useful and can meet the needs for a wide variety of numeric data, many types of numeric data require a fractional part to maintain a high degree of accuracy. Following are some examples of real numbers:

```
-32.36789
1.5E+2
.000000056789
55.67
-25.3999999
.333333333
```

To be able to represent real numbers efficiently, Pascal offers five different real number types: **Single**, **Real**, **Double**, **Extended**, and **Comp**.

Real number types can also be called *floating-point* types. Additionally, the **Comp** data type is a little unique. It is used to store extremely large integers and doesn't save a number's fractional part.

Table 2.5 illustrates the range of numbers the five real number data types can represent.

Table 2.5 *The Pascal real number types*

Data type	Range of values	Size in bytes	Significant digits
Single	1.5E-45 to 3.4E+38	4	7-8
Real	2.9E-39 to 17E+38	6	11-12
Double	5.0E-324 to 1.7E+308	8	15-16
Extended	3.4E-4951 to 1.1E+4932	10	19-20
Comp	-9.2E+18 to 9.2E+18	8	19-20

As with the integer types, Pascal's real number types take a varying amount of memory to store. Consequently, you should always try to use the smallest real number type possible for a given task. Also like the integer types, calculations are performed a lot faster on the smaller real number types than on the larger real number types.

Table 2.6 shows the format for defining real number variables. As it illustrates, you can define more than one variable per statement by separating the variable identifiers with commas. The following are some examples of real number variable definitions:

```
AccountBalance : Double;
degrees : Single;
Population : Comp;
CityBudget : Extended;
Credit, Debit : Real;
```

Table 2.6 *Defining a real number variable*

Var	
identifier : real number data type;	
identifier, identifier : real number data type;	
Where:	
identifier	is the variable's name.
real number data type	is **Single**, **Real**, **Double**, **Extended**, or **Comp.**

As with their integer counterparts, Pascal supports real number typed constants. Table 2.7 shows how real number typed constants are defined. Like the integer typed constants, you can only define one constant per definition statement. The following are some examples of typed real number constant definitions:

```
CashAccount : Double = -456.37;
WallHeight : Single = 32.33678;
degrees : Extended = .000000678;
```

Table 2.7 *Defining a typed real number constant*

Const	
identifier : real number data type = constant;	
Where:	
identifier	is the constant's name.
real number data type	is **Single**, **Real**, **Double**, **Extended**, or **Comp.**
constant	a constant value or expression.

To illustrate the use of Pascal's real number data types, Listing 2.2 presents a brief program, which defines a number of real number variables and constants and displays their assigned values.

Listing 2.2

```
{ list2-2.pas - Define and display a variety of real
numbers }
program real_numbers;

{$E+} { Turbo Pascal 8087 Emulation Directive - Omit
For QuickPascal }
{$N+} { Turbo Pascal 8087 Directive - Omit For
QuickPascal }

const
     single_const : single = 32.3;
     real_const : real = -0.0000032;
     double_const : double = 666.788888;
     extended_const : extended = 999.999;
     comp_const : comp = 32456789;

var
   single_var : single;
   real_var : real;
   double_var : double;
   extended_var : extended;
   comp_var : comp;

begin
     single_var := -45.667;
     real_var := 32.4568;
     double_var := 10000.34;
```

Listing 2.2

```
    extended_var := 55000.0003;
    comp_var := -4567;
    writeln('single_const   = ', single_const);
    writeln('real_const     = ', real_const);
    writeln('double_const   = ', double_const);
    writeln('extended_const = ', extended_const);
    writeln('comp_cont      = ', comp_const);
    writeln('single_var     = ', single_var);
    writeln('real_var       = ', real_var);
    writeln('double_var     = ', double_var);
    writeln('extended_var   = ', extended_var);
    writeln('comp_var       = ', comp_var);
end.
```

Lesson 5: Boolean Data Types

Many expressions in a computer program return either a true or false result. Unlike most other programming languages, Pascal provides a data type just for handling true/false values. This predefined data type is known as the *Boolean* data type. Because it represents only two logical values, true or false, the Boolean data type always holds either a *True* value or a *False* value.

Table 2.8 illustrates the format for defining Boolean variables. As this table illustrates, you can define more than one variable per statement by separating the variable identifiers with commas. The following are some examples of Boolean variable definitions:

```
Flag : Boolean;
IOResult : Boolean;
answer1, answer2 : Boolean;
```

```
On_Off_Flag : Boolean;
Error_Flag : Boolean
```

Table 2.8 *Defining a Boolean variable*

Var	
	identifier : **Boolean;**
	identifier, identifier : **Boolean;**
Where:	
identifier	is the variable's name.

As with integers and real numbers, typed Boolean constants can be defined with Pascal. Table 2.9 illustrates how typed Boolean constants are defined. Like other typed constants, you can only define one typed Boolean constant per definition statement. The following are some examples of typed Boolean constant definitions:

```
Flag : Boolean = False;
True_Result : Boolean = True;
Not_On : Boolean = False;
```

Table 2.9 *Defining a typed Boolean constant*

Const	
	identifier : **Boolean** = constant;
Where:	
identifier	is the constant's name.
constant	a constant value or expression.

To illustrate the use of Pascal's Boolean data type, Listing 2.3 presents a brief program, which defines a number of Boolean variables, constants and, displays their assigned values.

Listing 2.3

```
{ list2-3.pas - Define and display a variety of
booleans }
program booleans;

const
      false_flag : boolean = false;
      not_on_flag : boolean = false;

var
      flag : boolean;
      ioresult : boolean;

begin
      flag := false;
      ioresult := true;
      writeln('false_flag  = ', false_flag);
      writeln('not_on_flag = ', not_on_flag);
      writeln('flag        = ', flag);
      writeln('ioresult    = ', ioresult);
end.
```

Lesson 6: Characters

Quite often the result of an action is a character of data. Some examples of actions resulting in characters are keyboard input, display output, printer output, and some forms of disk input/output. To properly deal with character data, Pascal offers the *Char* data type. Table 2.10 illustrates the three forms of valid Pascal character data. The following are some examples of character data:

```
^M
#255
'k'
'y'
#10
```

Table 2.10 *Pascal character data representations*

Type	Representation
Control Characters	are represented by using the carat symbol(^) followed by a control letter. For example, **^A** would represent the control character 1, **^B** would represent the control character 2, etc.
Readable Characters	are represented in the form 'character'. For example, the letter **a** would be represented by **'a'**.
All Characters	are represented by using the number sign (#) followed by the character's ASCII code number. For example, the letter **g** would be represented by **#103**.
Where:	
Control Characters	are the ASCII characters 0 through 31.
Readable Characters	are the alphabetic, numeric, and punctuation characters.
All Characters	are any character in the ASCII code table.

Table 2.11 shows the format for defining character variables. As this table illustrates, you can define more than one variable per statement by separating the variable identifiers with commas. The following are some examples of character variable definitions:

```
Key : Char;
ReturnCode : Char;
First_Initial, Middle_Initial : Char;
DiskIO : Char;
PrinterCode : Char;
```

Table 2.11 *Defining a character variable*

Var	
identifier **: Char;**	
identifier, identifier **: Char;**	
Where:	
identifier	is the variable's name.

As with Pascal's other data types, Pascal supports typed character constants. Table 2.12 illustrates how typed character constants are defined. Like other typed constants, you can only define one constant per definition statement. The following are some examples of typed character constant definitions:

```
CR : Char = ^M;
PrinterCode : Char = 'B';
LF : Char = #10;
ErrorCode : Char = 'E';
```

Table 2.12 *Defining a type character constant*

Const	
identifier : **Char** = constant;	
Where:	
identifier	is the constant's name.
constant	is a constant value of expression.

To illustrate the use of Pascal's character data type, Listing 2.4 presents a brief program, which defines a number of character variables and constants and displays their assigned values.

Listing 2.4

```
{ list2-4.pas - Define and display a variety of char-
acters }
program characters;

const
     CR : char = ^M;
     LF : char = #10;

var
   a_character : char;
   another_character : char;

begin
     a_character := 'a';
     another_character := 'b';
     writeln('CR                = ', CR);
     writeln('LF                = ', LF);
     writeln('a_character       = ', a_character);
     writeln('another_character = ', another_character);
end.
```

Lesson 7: Strings

Although all of the previously described Pascal data types are important, perhaps the most important data a Pascal program is called upon to handle is *string* data. Whether it's a word processing program or just a simple utility program, strings are by far the most prevalent type of computer data. To meet the needs that string handling imposes upon a computer language, Pascal offers the *String* data type. The following are some examples of string data:

```
'This is a sample string'
'This is another sample string'
'This a more complex'#13#10'string.'
'I''m a string, too!'
```

Note the use of the double apostrophe ('') in the last example. Because Pascal strings are *delimited* (fancy word for surrounded) by apostrophes, you must use a double apostrophe to signify an apostrophe inside of a string. Failure to use a double apostrophe confuses the compiler into thinking the string is much shorter than it really should be.

Table 2.13 illustrates the format for defining string variables. As this table illustrates, you can define more than one variable per statement by separating the variable identifiers with commas. Additionally, Table 2.13 shows that an optional length can be specified for a string. If a length isn't specified, the defined string can be up to 255 characters in length. The following are some examples of string variable definitions:

```
Name : String[30];
City, State, Zip : String;
DisplayLine : String[80];
Address : String[30];
Response : String;
```

Table 2.13 *Defining a string variable*

Var	
	identifier **: String;**
	identifier **: String**[*length*]**;**
	identifier, identifier **: String;**
	identifier, identifier **: String**[*length*]**;**
Where:	
identifier	is the variable name.
length	is the string length, which must range from 1 to 255.

Like all of the previously mentioned data types, Pascal supports typed string constants. Table 2.14 illustrates how typed string constants are defined. As with other typed constants, you can only define one typed constant per definition statement. The following are some examples of typed string constant definitions:

```
name : String = 'Jane Smith';
city : String[30] = 'Los Angeles';
State : String[2] = 'NV';
zipcode : String = '05501';
```

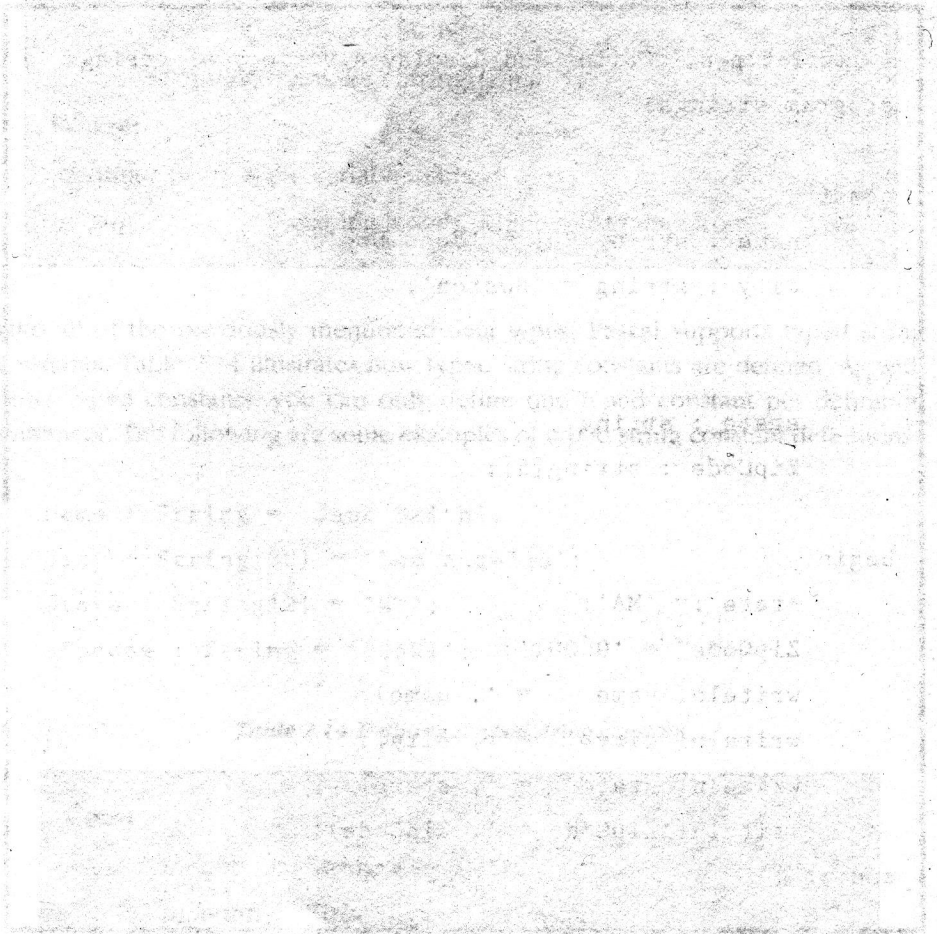
Table 2.14 *Defining a typed string constant*

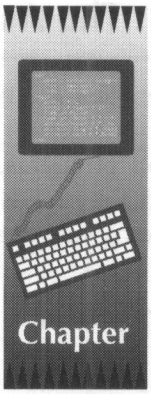
Const	
	indentifier **: String** = constant;
	indentifier **: String**[*length*] = constant;
Where:	
identifier	is the constant name.
constant	is the constant value or expression.
length	is the maximum length of the string, which must range from 1 to 255.

To illustrate the use of Pascal's string data type, Listing 2.5 presents a brief program that defines a number of character variables and constants and displays their assigned values.

Listing 2.5

```
{ list2-5.pas - Define and display a variety of strings }
program strings;

const
      name : string[20] = 'John Doe';
      city : string = 'Boston';

var
      state : string;
      ZipCode : string[5];

begin
      state := 'MA';
      ZipCode := '00001';
      writeln('name    = ', name);
      writeln('city    = ', city);
      writeln('state   = ', state);
      writeln('ZipCode = ', ZipCode);
end.
```





3

The Pascal Operators

In Chapter 2, you learned how data can be represented in a program by using the Pascal programming language's wide variety of predefined data types. Unfortunately, just knowing how data is stored really isn't all that useful. The Pascal programmer must also know how to manipulate data. Consequently, this chapter shows how data can be manipulated by using the Pascal operators. When combined with other variables and constants, the Pascal operators can be used to build extremely powerful and useful expressions.

Lesson 8: The Assignment Operator

As its name implies, the Pascal *assignment operator* (:=) assigns the result of an expression to a variable or typed constant. Because of its extensive use in Chapter Two's programs, you should already be somewhat familiar with its usage. Table 3.1 illustrates the use of the assignment operator. The following are some examples of how the assignment operator is used:

```
flag := False;
count := count + 1;
key := ReadKey;
Name := FirstName + ' ' + MiddleInitial + ' ' +
LastName;
pi := 22 / 7;
```

Table 3.1 *The Pascal assignment operator*

identifier := expression;	
Where:	
identifier	is a variable or typed constant name.
expression	is a valid Pascal expression.

To illustrate the use of the Pascal assignment operator, Listing 3.1 displays a brief program, which assigns values to a wide variety of variables.

Listing 3.1

```
{ list3-1.pas - Demonstrate the use of the Pascal
assignment operator }
program assignment_operator;

var
      count, number : integer;
      flag : boolean;
      Name : string;

begin
      count := 1;
      count := count + 1;
      flag := False;
      Name := 'John' + ' ' + 'Q. ' + 'Public';
      writeln('count = ', count);
      writeln('flag  = ', flag);
      writeln('Name  = ', Name);
end.
```

Lesson 9: The Unary Plus Operator

The Pascal *unary plus operator* (+) simply maintains the sign of an expression. In other words, it doesn't do a thing. This may seem to be a ludicrous statement, but it's quite true. The unary plus operator is simply ignored by Pascal and is only included in the language definition to prevent the compiler from generating unnecessary syntax errors. Table 3.2 illustrates the use of the unary plus operator. The following are some examples of the unary plus operator's proper use:

```
+count
+1.234
+recordnumber
```

Table 3.2 *The Pascal unary plus operator*

+expression	
Where:	
expression	is a valid Pascal expression.

Listing 3.2 displays a brief program that illustrates how the Pascal unary plus operator is used in an actual program.

Listing 3.2

```
{ list3-2.pas - Demonstrate the use of the Pascal
unary plus operator }
program unary_plus_operator;

var
      n1, n2 : integer;
      r1 : real;

begin
      r1 := +32.333;
      n1 := -23;
      n2 := +n1;
      writeln('n1 = ', n1);
      writeln('n2 = ', n2);
      writeln('r1 = ', r1);
end.
```

Lesson 10: The Unary Minus Operator

The Pascal *unary minus operator* (-) negates the value of an expression. If the expression is negative, the unary minus operator makes it positive. If the expression is positive, the unary minus operator makes it negative. Table 3.3 illustrates the use of the unary minus operator. The following are some examples of the unary minus operator's proper use:

```
-n1
-2.345678
-count
```

Table 3.3 *The Pascal unary minus operator*

-expression	
Where:	
expression	is valid Pascal expression.

Listing 3.3 displays a brief program that demonstrates how the Pascal unary minus operator is used in an actual program.

Listing 3.3

```
{ list3-3.pas - Demonstrate the use of the Pascal
unary minus operator }
program unary_minus_operator;

var
     n1, n2 : integer;
     r1 : real;

begin
```

Listing 3.3—Continued

```
        r1 := -32.333;
        n1 := -23;
        n2 := -n1;
        writeln('n1 = ', n1);
        writeln('n2 = ', n2);
        writeln('r1 = ', r1);
end.
```

Lesson 11: The Addition Operator

The Pascal *addition operator* (+) adds two expressions together. Table 3.4 illustrates the use of the addition operator. The following are some examples of the addition operator's proper use:

```
1 + 1
n1 + 3
33.333 + 500
22 + n2
i + j
```

Table 3.4 *The Pascal addition operator*

expression + expression	
Where:	
expression	is a valid Pascal expression.

Listing 3.4 displays a brief program that demonstrates how the Pascal addition operator is used in an actual program.

Listing 3.4

```
{ list3-4.pas - Demonstrate the use of the Pascal
addition operator }
program addition_operator;

var
      n1, n2 : integer;
      r1 : real;

begin
      r1 := -32.4567 + 33 + 0.67;
      n1 := 1 + 45;
      n2 := n1 + 1;
      writeln('n1 = ', n1);
      writeln('n2 = ', n2);
      writeln('r1 = ', r1);
end.
```

Lesson 12: The Subtraction Operator

The Pascal *subtraction operator* (-) subtracts the result of one expression from the result of another expression. Table 3.5 illustrates the use of the subtraction operator. The following are some examples of the subtraction operator's proper use:

```
2 - 3
33.456 - 1.325
n1 - g - c
55 - n1 - 3
6 - 1
```

Table 3.5 *The Pascal subtraction operator*

expression - expression **Where:** *expression* is a valid Pascal expression.

Listing 3.5 displays a brief program that illustrates how the Pascal subtraction operator is used in an actual program.

Listing 3.5

```
{ list3-5.pas - Demonstrate the use of the Pascal sub-
traction operator }
program subtraction_operator;

var
      n1, n2 : integer;
      r1 : real;

begin
      r1 := 32.4567 - 33 - 0.67;
      n1 := 1 - 45;
      n2 := n1 - 1;
      writeln('n1 = ', n1);
      writeln('n2 = ', n2);
      writeln('r1 = ', r1);
end.
```

Lesson 13: The Multiplication Operator

The Pascal *multiplication operator* (*) multiplies the result of one expression by the result of another expression. Table 3.6 illustrates the use of the multiplication operator. The following are some examples of the multiplication operator's proper use:

```
3 * 4
n * pi
3.43 * 0.5
99 * 6
x * y * z
```

Table 3.6 *The Pascal multiplication operator*

expression * expression	
Where:	
expression	is a valid Pascal expression.

Listing 3.6 displays a brief program that illustrates how the Pascal multiplication operator is used in an actual program.

Listing 3.6

```
{ list3-6.pas - Demonstrate the use of the Pascal mul-
tiplication operator }
program multiplication_operator;

var
      n1, n2 : integer;
      r1 : real;
```

Listing 3.6—Continued

```
begin
      r1 := 32.4567 * 27 * 0.5;
      n1 := 1 * 45;
      n2 := n1 * n1;
      writeln('n1 = ', n1);
      writeln('n2 = ', n2);
      writeln('r1 = ', r1);
end.
```

Lesson 14: The Real Number Division Operator

The Pascal *real number division operator* (/) divides the result of one expression by the result of another expression. As you will soon see, the Pascal division operators are a little unique. All of the previously covered arithmetic operators (+, -, and *) all return the same data type as the expressions they are applied to. For example, the addition of two integer expressions returns an integer result. The real number division operator differs from the other arithmetic operators by always returning a real number result. It doesn't matter whether the expressions being divided are integers or real numbers. The calculated result is always returned as a real number. Table 3.7 illustrates the use of the real number division operator. The following are some examples of the real number division operator's proper use:

```
22 / 7
15 / n1
x / z / y
count / 2
15.333 / 2.1023
```

Table 3.7 *The Pascal real number division operator*

expression / expression	
Where:	
expression	is a valid Pascal expression.

Listing 3.7 displays a brief program that demonstrates how the Pascal real number division operator is used in an actual program.

Listing 3.7

```
{ list3-7.pas - Demonstrate the use of the Pascal real
number division   operator}
program division_operator;

var
      r1 : real;

begin
      r1 := 32.4567 / 27 / 0.5;
      writeln('r1 = ', r1);
end.
```

Lesson 15: The Integer Division Operator

The Pascal *integer division operator* (**div**) divides the result of one integer expression by the result of another integer expression. The integer division operator always returns an integer result. Table 3.8 illustrates the use of the integer division operator. The following are some examples of the integer division operator's proper use:

```
22 div 7
n div 3
x div y div 2
16 div 8
5555 div n1
```

Table 3.8 *The Pascal integer division operator*

integer expression **div** integer expression	
Where:	
integer expression	is a valid Pascal integer expression.

Listing 3.8 displays a short program that demonstrates how the Pascal integer division operator is used in an actual program.

Listing 3.8

```
{ list3-8.pas - Demonstrate the use of the Pascal
integer division operator }
program integer_division_operator;

var
      n1, n2 : integer;

begin
      n1 := 3400 div 16;
      n2 := n1 div 3;
      writeln('n1 = ', n1);
      writeln('n2 = ', n2);
end.
```

Lesson 16: The Remainder Operator

The Pascal *remainder operator* (**mod**) figures a remainder by dividing the result of one integer expression by the result of another integer expression. The remainder operator always returns an integer result. Table 3.9 illustrates the use of the remainder operator. The following are some examples of the remainder operator's proper use:

```
count mod 5
33 mod 2
45 mod n
x mod y mod z
n1 mod n2
```

Table 3.9 *The Pascal remainder operator*

integer expression **mod** integer expression	
Where:	
integer expression	is a valid Pascal integer expression.

Listing 3.9 displays a brief program that illustrates how the Pascal remainder operator is used in an actual program.

Listing 3.9

```
{ list3-9.pas - Demonstrate the use of the Pascal
remainder operator }
program remainder_operator;

var
      n1, n2 : integer;
```

Listing 3.9—Continued

```
        n1, n2 : integer;

begin
        n1 := 3400 mod 16;
        n2 := n1 mod 3;
        writeln('n1 = ', n1);
        writeln('n2 = ', n2);
end.
```

Lesson 17: The Logical Negation Operator

The Pascal *logical negation operator* (**not**) negates the result of a Boolean expression. If the Boolean expression is equal to **True**, the logical negation operator makes it **False**. Otherwise, the logical negation operator returns a **True** result for **False** Boolean expressions. Table 3.10 presents a truth table for the logical negation operator. This table illustrates how the logical negation operator performs its function.

Table 3.10 *Logical negation truth table*

Value X	Value not X
True	False
False	True

Table 3.11 illustrates the use of the logical negation operator. The following are some examples of the logical negation operator's proper use:

```
not flag
not False
not error_flag
```

Table 3.11 *The Pascal logical negation operator*

not Boolean expression	
Where:	
Boolean expression	is a valid Pascal Boolean expression.

Listing 3.10 displays a brief program that illustrates how the Pascal logical negation operator is used by displaying a logical negation truth table.

Listing 3.10

```
{ list3-10.pas - Demonstrate the Pascal logical nega-
tion operator }
program logical_negation_operator;

begin
      writeln('Logical Negation Truth Table');
      writeln('============================');
      writeln('Value    Value    Result');
      writeln('X                 Not X');
      writeln('----------------');
      writeln('True              ', not True);
      writeln('False             ', not False);
      writeln('============================');
end.
```

Lesson 18: The Logical And Operator

The Pascal *logical and operator* (**and**) compares two Boolean expressions and returns a **True** result only if both of the Boolean expressions are equal to **True**. Otherwise, the logical and operator returns a **False** result. Table 3.12 presents a truth table for the logical and operator. This table illustrates how the logical and operator performs its function.

Table 3.12 *Logical and truth table*

Value X	Value Y	Value X and Y
True	True	True
True	False	False
False	True	False
False	False	False

Table 3.13 illustrates the use of the logical and operator. The following are some examples of the logical and operator's proper use:

```
flag and True
error and EndOfFile
keypressed and flag
```

Table 3.13 *The Pascal logical and operator*

Boolean expression **and** Boolean expression **Where:** *Boolean expression* is a valid Pascal Boolean expression.

Listing 3.11 displays a short program that demonstrates how the Pascal logical and operator is used by displaying a logical and truth table.

Listing 3.11

```
{ list3-11.pas - Demonstrate the Pascal logical and
operator }
program logical_and_operator;

begin
      writeln('Logical And Truth Table');
      writeln('============================');
      writeln('Value    Value    Result');
      writeln('X        Y        X AND Y');
      writeln('----------------');
      writeln('True     True     ', True and True);
      writeln('True     False    ', True and False);
      writeln('False    True     ', False and True);
      writeln('False    False    ', False and False);
      writeln('============================');
end.
```

Lesson 19: The Logical Or Operator

The Pascal *logical or operator* (**or**) compares two Boolean expressions and returns a **True** result if either of the Boolean expressions is equal to **True**. Otherwise, the logical or operator returns a **False** result. Table 3.14 presents a truth table for the logical or operator. This table illustrates how the logical or operator performs its function.

Table 3.14 *Logical or truth table*

Value X	Value Y	Result X or Y
True	True	True
True	False	True
False	True	True
False	False	False

Table 3.15 illustrates the use of the logical or operator. The following are some examples of the logical or operator's proper use:

```
flag or True
error or EndOfFile
keypressed or mouseclicked
```

Table 3.15 *The Pascal logical or operator*

Boolean expression **or** Boolean expression	
Where:	
Boolean expression	is a valid Pascal Boolean expression.

Listing 3.12 displays a brief program that demonstrates how the Pascal logical or operator is used by displaying a logical or truth table.

Listing 3.12

```
{ list3-12.pas - Demonstrate the Pascal logical or
operator }
program logical_or_operator;
```

Listing 3.12—Continued

```
begin
      writeln('Logical Or Truth Table');
      writeln('============================');
      writeln('Value    Value    Result');
      writeln('X        Y        X OR Y');
      writeln('-----------------');
      writeln('True     True     ', True or True);
      writeln('True     False    ', True or False);
      writeln('False    True     ', False or True);
      writeln('False    False    ', False or False);
      writeln('============================');
end.
```

Lesson 20: The Exclusive Or Operator

The Pascal *exclusive or operator* (**xor**) compares two Boolean expressions and returns a **True** result if both of the Boolean expressions are different. Otherwise, the exclusive or operator returns a **False** result. Table 3.16 presents a truth table for the exclusive or operator. This table illustrates how the exclusive or operator performs its function.

Table 3.16 *Exclusive or truth table*

Value X	Value Y	Result X xor Y
True	True	False
True	False	True
False	True	True
False	False	False

Table 3.17 illustrates the use of the exclusive or operator. The following are some examples of the exclusive or operator's proper use:

```
flag xor True
error xor False
keypressed xor mouseclicked
```

Table 3.17 *The Pascal exclusive or operator*

Boolean expression **xor** Boolean expression	
Where:	
Boolean expression	is a valid Boolean expression.

Listing 3.13 displays a short program that illustrates how the Pascal exclusive or operator is used by displaying an exclusive or truth table.

Listing 3.13

```
{ list3-13.pas - Demonstrate the Pascal exclusive or
operator }
program exclusive_or_operator;

begin
      writeln('Exclusive Or Truth Table');
      writeln('============================');
      writeln('Value    Value    Result');
      writeln('X        Y        X XOR Y');
      writeln('True     True     ', True xor True);
      writeln('True     False    ', True xor False);
```

Listing 3.13—Continued

```
        writeln('False    True     ', False xor True);
        writeln('False    False    ', False xor False);
        writeln('============================');
end.
```

Lesson 21: The Bitwise Negation Operator

The Pascal *bitwise negation operator* (**not**) negates the result of an integer expression. The bitwise negation operator performs its intended function by inverting the value of each of an integer's **bits**. If you are unfamiliar with the term bit, Figure 3.01 should be of assistance. Each byte of memory (one character of memory) is comprised of eight bits. Each bit either holds the value of 1 or 0. By simply inverting each of the integer expression's bits, the bitwise negation operator effectively negates the expression.

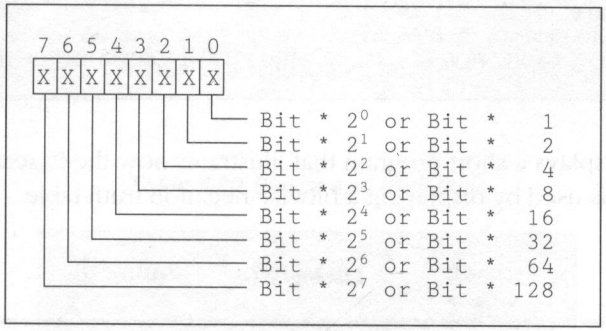


Figure 3.01 *A byte of bits*

Table 3.18 displays a truth table for the bitwise negation operator. This table illustrates how the bitwise negation operator performs its function.

Table 3.18 *Bitwise negation truth table*

Bit Value X	Result not X
1	0
0	1

Table 3.19 displays the use of the bitwise negation operator. The following are some examples of the bitwise negation operator's proper use:

```
not mask
not pixels
not bit_mask
```

Table 3.19 *The Pascal bitwise negation operator*

not integer expression	
Where:	
integer expression	is a valid Pascal integer expression.

Listing 3.14 displays a short program that illustrates how the Pascal bitwise negation operator is used by displaying a bitwise negation truth table.

Listing 3.14

```
{ list3-14.pas - Demonstrate the Pascal bitwise nega-
tion operator }
program bitwise_negation_operator;


const
      one = not 0;
      zero = not 1;
```

Listing 3.14—Continued

```
begin
      writeln('Bitwise Negation Truth Table');
      writeln('============================');
      writeln('Value    Value    Result');
      writeln('X                 not X');
      writeln('----------------');
      writeln('1                 ', not one);
      writeln('0                 ', not zero);
      writeln('----------------');
end.
```

Lesson 22: The Bitwise And Operator

The Pascal *bitwise and operator* (**and**) compares two integer expressions and returns the result of anding them bit-by-bit. Table 3.20 presents a truth table for the bitwise and operator. This table illustrates how the bitwise and operator performs its function.

Table 3.20 *Bitwise and truth table*

Bit value X	Bit value Y	Result X and Y
1	1	1
1	0	0
0	1	0
0	0	0

Table 3.21 displays the use of the bitwise and operator. The following are some examples of the bitwise and operator's proper use:

```
value and BitMask
ShiftPressed and 1
Pixels and $F0
```

Table 3.21 *The Pascal bitwise and operator*

integer expression **and** integer expression	
Where:	
integer expression	is a valid Pascal integer expression.

Listing 3.15 presents a short program that demonstrates how the Pascal bitwise and operator is used by displaying a bitwise and truth table.

Listing 3.15

```
{ list3-15.pas - Demonstrate the Pascal bitwise and
operator }
program bitwise_and_operator;

begin
      writeln('Bitwise And Truth Table');
      writeln('============================');
      writeln('Value    Value    Result');
      writeln('X        Y        X and Y');
      writeln('----------------');
```

Listing 3.15—Continued

```
        writeln('1          1          ', 1 and 1);
        writeln('1          0          ', 1 and 0);
        writeln('0          1          ', 0 and 1);
        writeln('0          0          ', 0 and 0);
        writeln('-------------------');
end.
```

Lesson 23: The Bitwise Or Operator

The Pascal *bitwise or operator* (**or**) compares two integer expressions and returns the result of oring them bit-by-bit. Table 3.22 presents a truth table for the bitwise or operator. This table illustrates how the bitwise or operator performs its function.

Table 3.22 *Bitwise or truth table*

Bit value X	Bit value Y	Result X or Y
1	1	1
1	0	1
0	1	1
0	0	0

Table 3.23 illustrates the use of the bitwise or operator. The following are some examples of the bitwise or operator's proper use:

```
pixel or 1
flags or mask
shiftmask or 2
```

Table 3.23 *The Pascal bitwise or operator*

integer expression **or** integer expression	
Where:	
integer expression	is a valid Pascal integer expression.

Listing 3.16 presents a short program that illustrates how the Pascal bitwise or operator is used in an actual program by displaying a bitwise or truth table.

Listing 3.16

```
{ list3-16.pas - Demonstrate the Pascal bitwise or
operator }
program bitwise_or_operator;

begin
      writeln('Bitwise Or Truth Table');
      writeln('============================');
      writeln('Value    Value    Result');
      writeln('X        Y        X or Y');
      writeln('----------------');
      writeln('1        1        ', 1 or 1);
      writeln('1        0        ', 1 or 0);
      writeln('0        1        ', 0 or 1);
      writeln('0        0        ', 0 or 0);
      writeln('----------------');
end.
```

Lesson 24: The Bitwise Exclusive Or Operator

The Pascal *bitwise exclusive or operator* (**xor**) compares two integer expressions and returns the result of xoring them bit-by-bit. Table 3.24 presents a truth table for the bitwise exclusive or operator. This table illustrates how the bitwise exclusive or operator performs its function.

Table 3.24 *Bitwise exclusive or truth table*

Bit value X	Bit value Y	Result X or Y
1	1	0
1	0	1
0	1	1
0	0	0

Table 3.25 illustrates the use of the bitwise exclusive or operator. The following are some examples of the bitwise exclusive or operator's proper use:

```
pixels xor $FF
ErrorFlag xor mask
ShiftFlag xor 1
```

Table 3.25 *The Pascal bitwise exclusive or operator*

integer expression **xor** integer expression	
Where:	
integer expression	is a valid Pascal integer expression.

Listing 3.17 presents a brief program that illustrates how the Pascal bitwise exclusive or operator is used in an actual program by displaying a bitwise exclusive or truth table.

Listing 3.17

```
{ list3-17.pas - Demonstrate the Pascal bitwise xor
operator }
program bitwise_xor_operator;

begin
      writeln('Bitwise Xor Truth Table');
      writeln('============================');
      writeln('Value    Value    Result');
      writeln('X        Y        X xor Y');
      writeln('-----------------');
      writeln('1        1        ', 1 xor 1);
      writeln('1        0        ', 1 xor 0);
      writeln('0        1        ', 0 xor 1);
      writeln('0        0        ', 0 xor 0);
      writeln('-----------------');
end.
```

Lesson 25: The Bitwise Shift Left Operator

The Pascal *bitwise shift left operator* (**shl**) shifts all of the bits in an integer expression to the left by the number of places specified by another integer expression. Essentially, the shift left operator multiplies an integer value two times for every position to the left it is shifted. For example, the expression **4 shl 2** would have the same effect as the expression **4 * 4**.

Table 3.26 illustrates the use of the shift left operator. The following are some examples of the shift left operator's proper use:

```
mask shl 1
number shl count
4 shl count
```

Table 3.26 *The Pascal shift left operator*

integer expression **shl** count	
Where:	
integer expression	is a valid Pascal integer expression.
count	is a valid Pascal integer expression, which specifies the number of places to shift the preceding integer by.

Listing 3.18 displays a short program that illustrates how the Pascal shift left operator is used in an actual program.

Listing 3.18

```
{ list3-18.pas - Demonstrate the Pascal bitwise shift
left operator }
program bitwise_shift_left_operator;

var
      n1, n2 : integer;

begin
      n1 := 4 shl 1;
      n2 := n1 shl 2;
      writeln('n1 = ', n1);
      writeln('n2 = ', n2);
end.
```

Lesson 26: The Bitwise Shift Right Operator

The Pascal *bitwise shift right operator* (**shr**) shifts all of the bits in an integer expression to the right by the number of places specified by another integer expression. Essentially, the shift right operator divides an integer value two times for every position to the right it is shifted. For example, the expression **63 shr 1** would have the same effect as the expression **63 div 2**.

Table 3.27 illustrates the use of the shift right operator. The following are some examples of the shift right operator's proper use:

```
mask shr 2
number shr count
45 shr times
```

Table 3.27 *The Pascal shift right operator*

integer expression **shr** count	
Where:	
integer expression	is a valid Pascal integer expression.
count	is a valid Pascal integer expression, which specifies the number of places to shift the preceding integer by.

Listing 3.19 displays a short program that illustrates how the Pascal shift right operator is used in an actual program.

Listing 3.19

```
{ list3-19.pas - Demonstrate the Pascal bitwise shift
right operator }
program bitwise_shift_right_operator;

var
      n1, n2 : integer;

begin
      n1 := 64 shr 1;
      n2 := n1 shr 2;
      writeln('n1 = ', n1);
      writeln('n2 = ', n2);
end.
```

Lesson 27: The String Concatenation Operator

The Pascal *string concatenation operator* (+) is used to combine characters and strings to form an even larger string. Table 2.28 illustrates the use of the string concatenation operator. The following are some examples of the string concatenation operator's proper use:

```
'Washington' + 'D.C.'
#13 + String
FirstName + MiddleInitial + LastName
```

Table 3.28 *The Pascal string concatenation operator*

string expression + string expression
string expression + character expression
character expression + string expression
character expression + character expression
Where:
string expression is a valid Pascal string expression.
character expression is a valid Pascal character expression.

Listing 3.20 displays a brief program that demonstrates how the Pascal string concatenation operator is used in an actual program.

Listing 3.20

```
{ list3-20.pas - Demonstrate the Pascal string con-
catenation operator }
program string_concatenation_operator;

var
      FirstName, Lastname, Name : string;
      MiddleInitial : char;

begin
      FirstName := 'John';
      LastName := 'Smith';
      MiddleInitial := 'D';
      Name := FirstName + #32 + MiddleInitial + '. ' +
      LastName;
      writeln('Name = ', Name);
end.
```

Lesson 28: The Equal To Operator

The Pascal *equal to operator* (=) compares two expressions to see if they are equal in value. If the two expressions are equal in value, the equal to operator returns a value of **True**. Otherwise, the equal to operator returns a value of **False** to indicate that the two expression aren't equal in value.

Table 3.29 illustrates the use of the equal to operator. The following are some examples of the equal to operator's proper use:

```
Flag = True
n = 1
15.0 = Diameter
```

Table 3.29 *The Pascal equal to operator*

expression = expression	
Where:	
expression	is a valid Pascal expression.

Listing 3.21 displays a short program that illustrates how the Pascal equal to operator is used in an actual program.

Listing 3.21

```
{ list3-21.pas - Demonstrate the Pascal equal to oper-
ator }
program equal_to_operator;

begin
      writeln('1 = 1 is ', 1 = 1);
      writeln('2 = 1 is ', 2 = 1);
      writeln('1 = 2 is ', 1 = 2);
end.
```

Lesson 29: The Not Equal To Operator

The Pascal *not equal to operator* (<>) compares two expressions to see if they are unequal in value. If the two expressions aren't equal in value, the not equal to operator returns a value of **True**. Otherwise, the not equal to operator returns a value of **False** to indicate that the two expressions are equal in value.

Table 3.30 illustrates the use of the not equal to operator. The following are some examples of the not equal to operator's proper use:

```
Flag <> True
count <> 2
MouseButton <> Clicked
```

Table 3.30 *The Pascal not equal to operator*

expression <> expression	
Where:	
expression	is a valid Pascal expression.

Listing 3.22 displays a brief program that illustrates how the Pascal not equal to operator is used in an actual program.

Listing 3.22

```
{ list3-22.pas - Demonstrate the Pascal not equal to
operator }
program not_equal_to_operator;

begin
      writeln('1 <> 1 is ', 1 <> 1);
      writeln('2 <> 1 is ', 2 <> 1);
      writeln('1 <> 2 is ', 1 <> 2);
end.
```

Lesson 30: The Greater Than Operator

The Pascal *greater than operator* (>) compares two expressions to see if the first expression is greater than the second expression. If the first expression is greater than the second expression, the greater than operator returns a value of **True**. Otherwise, the greater than operator returns a value of **False** to indicate that the first expression is less than or equal to the second expression.

Table 3.31 illustrates the use of the greater than operator. The following are some examples of the greater than operator's proper use:

```
n > 1
count > maximum
mouse_column > 80
```

Table 3.31 *The Pascal greater than operator*

expression > expression	
Where:	
expression	is a valid Pascal expression.

Listing 3.23 displays a brief program that demonstrates how the Pascal greater than operator is used in an actual program.

Listing 3.23

```
{ list3-23.pas - Demonstrate the Pascal greater than
operator }
program greater_than_operator;

begin
      writeln('1 > 1 is ', 1 > 1);
      writeln('2 > 1 is ', 2 > 1);
      writeln('1 > 2 is ', 1 > 2);
end.
```

Lesson 31: The Greater Than Or Equal To Operator

The Pascal *greater than or equal to operator* (**>=**) compares two expressions to see if the first expression is greater than or equal to the second expression. If the first expression is greater than or equal to the second expression, the greater than or equal to operator returns a value of **True**. Otherwise, the greater than or equal to operator returns a value of **False** to indicate that the first expression is less than the second expression.

Table 3.32 illustrates the use of the greater than or equal to operator. The following are some examples of the greater than or equal to operator's proper use:

```
count >= 55
DisplayRow >= 23
n >= 5
```

Table 3.32 *The Pascal greater than or equal to operator*

expression >= expression	
Where:	
expression	is a valid Pascal expression.

Listing 3.24 displays a short program that demonstrates how the Pascal greater than or equal to operator is used in an actual program.

Listing 3.24

```
{ list3-24.pas - Demonstrate the Pascal greater than
or equal to operator }
program greater_than_or_equal_to_operator;

begin
      writeln('1 >= 1 is ', 1 >= 1);
      writeln('2 >= 1 is ', 2 >= 1);
      writeln('1 >= 2 is ', 1 >= 2);
end.
```

Lesson 32: The Less Than Operator

The Pascal *less than operator* (<) compares two expressions to see if the first expression is less than the second expression. If the first expression is less than the second expression, the less than operator returns a value of **True**. Otherwise, the less than operator returns a value of **False** to indicate that the first expression is greater than or equal to the second expression.

Table 3.33 illustrates the use of the less than operator. The following are some examples of the less than operator's proper use:

```
n < 3
MouseRow < 0
counter < 55
```

Table 3.33 *The Pascal less than operator*

expression < expression
Where:
expression is a valid Pascal expression.

Listing 3.25 displays a short program that illustrates how the Pascal less than operator is used in an actual program.

Listing 3.25

```
{ list3-25.pas - Demonstrate the Pascal less than
operator }
program less_than_operator;

begin
      writeln('1 < 1 is ', 1 < 1);
      writeln('2 < 1 is ', 2 < 1);
      writeln('1 < 2 is ', 1 < 2);
end.
```

Lesson 33: The Less Than Or Equal To Operator

The Pascal *less than or equal to operator* (**<=**) compares two expressions to see if the first expression is less than or equal to the second expression. If the first expression is less than or equal to the second expression, the less than or equal to operator returns a value of **True**. Otherwise, the less than or equal to operator returns a value of **False** to indicate that the first expression is greater than the second expression.

Table 3.34 illustrates the use of the less than or equal to operator. The following are some examples of the less than or equal to operator's proper use:

```
count <= 5
DisplayColumn <= 78
n <= 3
```

Table 3.34 *The Pascal less than or equal to operator*

expression <= expression	
Where:	
expression	is a valid Pascal expression.

Listing 3.26 displays a short program that demonstrates how the Pascal less than or equal to operator is used in an actual program.

Listing 3.26

```
{ list3-26.pas - Demonstrate the Pascal less than or
equal to operator }
program less_than_or_equal_to_operator;

begin
      writeln('1 <= 1 is ', 1 <= 1);
      writeln('2 <= 1 is ', 2 <= 1);
      writeln('1 <= 2 is ', 1 <= 2);
end.
```

Lesson 34: Operator Precedence

How an expression with only one operator type is evaluated is pretty straightforward. For example, the expression **2 + 3 + 6** is evaluated in two separate steps. First, the expression **2 + 3** is figured and returns a result of **5**. Next, the **6** is

added to the previous result. Accordingly, the expression returns a value of **11**. Seems pretty easy, doesn't it? Pascal simply evaluates expressions with all the same operator type from left to right.

What if, for example, the expression **2 + 3 * 6** needed to be evaluated. If Pascal was to evaluate the **2 + 3** portion of the expression first, the result would be determined as follows:

```
2 + 3 * 6
    5   * 6 = 30
```

But, what if Pascal evaluates the **3 * 6** portion of the expression first. The result would be determined as follows:

```
2 + 3 * 6 = ?
2 +   18  = 20
```

It's rather obvious that the two different methods for evaluating the expression return vastly different results. To overcome these types of conflicts, the Pascal programming language uses a set of rules called *operator precedence* to evaluate expressions. Essentially, Pascal assigns a precedence level for each of its operators. When an expression is evaluated, the *subexpression* (one of the individual expressions that make up a more complex expression) of the expression with the highest precedence is evaluated first. The part of the expression with the next highest precedence is evaluated second. This method continues until the portion of the expression with the lowest precedence has been evaluated.

Table 3.35 illustrates the precedence levels Pascal assigns to its wide range of operators. (Note that two of these operators, **@** and **in**, haven't been covered yet. These operators are used with some of Pascal's more advanced data types and are covered later on in this book.) As Table 3.35 shows, some of the operators have equal levels of precedence. Whenever Pascal encounters two or more subexpressions with operators of equal precedence, they are evaluated on a strictly left to right basis.

Table 3.35 *The Pascal operator precedence levels*

Level	Operators
4	@, not
3	*, /, div, mod, and, shl, shr
2	+, -, or, xor
1	=, <>, <, <=, >, >=, IN

It is possible to override the Pascal precedence rules by simply surrounding a subexpression with parentheses. Surrounding a subexpression with parentheses tells Pascal to evaluate the subexpression first. For example, the expression **5 * 3 - 2** would be evaluated as follows using the normal Pascal precedence rules:

```
5 * 3 - 2 = ?
   15 - 2 = 13
```

However, the expression **5 * (3 - 2)** would be evaluated as follows by Pascal:

```
5 * (3 - 2) = ?
5 *     1   = 5
```

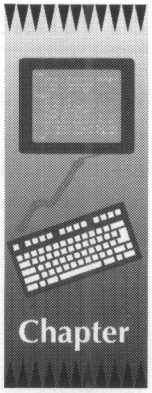
What if you had an expression with nested (one inside the other) parentheses such as **150 div ((4 - 2) * 3)**? Pascal would interpret such an expression by evaluating the inner most subexpression first. Thus, the expression **150 div ((4 - 2) * 3)** would be evaluated as follows:

```
150 div ((4 - 2) * 3) = ?
150 div (   2    * 3) = ?
150 div         6     = 25
```

Listing 3.27 displays a short program that demonstrates how Pascal would evaluate a variety of expressions.

Listing 3.27

```
{ list3-27.pas - Demonstrate Pascal precedence rules }
program precedence;

begin
      writeln('1 + 3 * 4 = ?');
      writeln('1 +   ', 3 * 4, '  = ', 1 + 3 * 4);
      writeln('150 div ((4 - 2) * 3) = ?');
      writeln('150 div (   ', 4 - 2, '    * 3) = ?');
      writeln('150 div             ', 2 * 3, '  = ',
      150 div ((4 - 2) * 3));
end.
```



4

Program Flow

As mentioned in Chapter 1, a Pascal program begins at the program's main body **begin** statement. Except when a procedure or function is called or a program flow keyword is encountered, program execution continues from the top of the program's main body to the bottom. Eventually, execution ends when the program reaches the main body's **end** statement. This chapter introduces you to the Pascal programming language's program flow keywords and shows how they are used in actual Pascal programs.

Lesson 35: While Loops

The Pascal **while** keyword tells the program to continuously execute a statement until a condition is no longer true. Table 4.1 illustrates how the **while** keyword is used to construct a **while** loop. As this table shows, the statement to be executed can be either a single program statement or a multi-statement **begin..end** block.

***Table 4.1** The **while** keyword*

```
while condition do
        statement;
or
while condition do
begin
        statement;
        .
        .
        tatement;
end;

Where:
condition       is a valid Boolean expression.
statement       is a valid program statement.
```

Listing 4.1 illustrates the use of the Pascal **while** keyword by displaying every odd number between 100 and 200.

Listing 4.1

```
{ list4-1.pas - Demonstrate the Pascal while keyword }
program while_loop1;

var
      number : integer;

begin
      number := 101;
      while number < 201 do
      begin
            writeln('number = ', number);
            number := number + 2;
      end;
end.
```

To better understand how the **while** keyword works, let's take a closer look at Listing 4.1's main program body.

```
number := 101;
```

assigns the value 101 to the **integer** variable **number**.

```
while number < 201 do
```

checks the value of **number**. If **number** is less than 201, the **while** statement is executed. If **number** is greater than or equal to 201, the **while** statement is ignored.

```
writeln('number = ', number);
```

displays **number**'s current value.

```
number := number + 2;
```

increases the value of **number** to the next odd value. After executing this statement, the program loops back to the **while** keyword.

At this point, you may be wondering what would happen if the **while** condition is initially **False**. Quite simply, the **while** statement would never be executed. For example, the following **while** loop would never be executed:

```
while False do
   i = i + 1;
```

The initial condition is **False**. Consequently, the statement **i = i + 1** will never be executed.

Lesson 36: Repeat Loops

The Pascal **repeat** keyword is very similar to the **while** keyword. The only difference between the two is that the **repeat** keyword checks for a condition after it executes its associated program statement. Therefore, a **repeat** loop is kind of a backwards **while** loop. Table 4.2 illustrates how the **repeat** keyword is used to construct a **repeat** loop. As this table illustrates, the **repeat** statement can be either a single program statement or a multi-statement **begin..end** block.

Table 4.2 *The* ***repeat*** *keyword*

```
repeat
                    statement;
until condition;
or
repeat
begin
                    statement;
                    .
                    .
                    statement;
end;
until condition;

Where:
condition           is a valid Boolean expression.
statement           is a valid program statement.
```

Listing 4.2 illustrates the use of the Pascal **repeat** keyword by displaying every odd number between 100 and 200.

Listing 4.2

```
{ list4-2.pas - Demonstrate the Pascal repeat keyword }
program repeat_loop;

var
      number : integer;

begin
      number := 101;
      repeat
      begin
            writeln('number = ', number);
            number := number + 2;
      end;
      until number > 199;
end.
```

To better understand how the **repeat** keyword works, let's take a closer look at Listing 4.2's main program body.

```
number := 101;
```

assigns the value 101 to the **integer** variable **number**.

```
repeat
```

causes the next statement to be executed.

```
writeln('number = ', number);
```

displays **number**'s current value.

```
number := number + 2;
```

increases the value of **number** to the next odd value.

```
until number > 199;
```

checks to see if the last odd value has been displayed. If the last odd value hasn't been displayed, program execution loops back to the **repeat** keyword.

You should note that a **repeat** loop is always executed at least once. As in the above example, the **repeat** statement is executed before the condition is checked; therefore, the statement is always executed at least once.

Lesson 37: For Loops

The Pascal **for** keyword is used to tell the program to execute a statement for a set number of times. Table 4.3 illustrates how the **for** keyword is used to construct a **for** loop.

Table 4.3 *The* ***for*** *keyword*

```
for identifier := expression to expression do
                    statement;
or
for identifier := expression to expression do
begin
                    statement;
                    .
                    .
                    statement;
end;
or
for identifier := expression downto expression do
                    statement;
or
for identifier := expression downto expression do
begin
                    statement;
                    .
                    .
                    statement;
end;
```

Where:

identifier	is a valid variable or typed constant identifier.
expression	is a valid Pascal expression.
statement	is a valid program statement.

As Table 4.3 illustrates, the **for** statement assigns the value of an expression to a variable. (Note that the expressions in a **for** statement must return an *ordinal* result. Ordinal numbers are be covered in detail in a later chapter, but for now just think of them as any integer.) If the **to** keyword is used in the **for** statement, program execution continues by checking to see if the variable's value is less than or equal to the value of the **for** statement's second expression. If the variable's value is less than or equal to the value of the second expression, the statement after the **do** keyword is executed. After executing the **do** statement, the variable is incremented (variable := variable + 1) and its contents are once again checked against the result of the second expression.

If the **downto** keyword is used in the **for** statement, program execution continues after the variable initialization by checking to see if the variable's value is greater than or equal to the value of the **for** statement's second expression. If the variable's value is greater than or equal to the value of the second expression, the statement following the **do** keyword is executed next. After executing the **do** statement, the variable is decremented (variable := variable - 1) and its contents are once again checked against the result of the second expression.

Listing 4.3 illustrates the use of the Pascal **for** keyword by displaying every number between 60 and 100 in ascending order.

Listing 4.3

```
{ list4-3.pas - Demonstrate an ascending for loop }
program ascending_for;

var
      cnt : integer;

begin
      for cnt := 60 to 100 do
            writeln('cnt = ', cnt);
end.
```

Listing 4.4 illustrates a descending **for** loop by displaying every number between 60 and 100 in descending order.

Listing 4.4

```
{ list4-4.pas - Demonstrate a descending for loop }
program descending_for;

var
      cnt : integer;

begin
      for cnt := 100 downto 60 do
            writeln('cnt = ', cnt);
end.
```

You should note that whenever the **for** variable's initial value exceeds the value of the second expression in a **for..to..do** combination or is smaller than the value of the **for..downto..do** combination, the statement following the **do** is never executed by the program. For example, neither the statement **for i := 1 to 0 do** or the statement **for i := 0 downto 1 do** would ever cause their associated **do** statements to be executed.

Lesson 38: The Break Procedure

The **break** procedure is used to prematurely terminate out of a **for**, **repeat**, or **while** loop. Listing 4.5 demonstrates how the Pascal **break** procedure can be used to prematurely terminate a **for** loop.

Listing 4.5

```
{ list4-5.pas - Demonstrate the break procedure }
program break_procedure;

var
    cnt : integer;

begin
    for cnt := 1 to 100 do
    begin
        writeln('cnt = ', cnt);
        if cnt = 51 then
            break;
    end;
end.
```

Note that the above program displays the numbers 1 through 50, but it does not display the numbers 51 through 100 because the **break** procedure terminates the **for** loop as soon as **cnt** reaches a value of 50.

Lesson 39: The Continue Procedure

The **continue** procedure is used in a **for**, **repeat**, or **while** loop to force program execution to start the loop's next iteration. Listing 4.6 demonstrates how the **continue** procedure can be used to prematurely force a loop to its next iteration.

Listing 4.6

```
{ list4-6.pas - Demonstrate the continue procedure }
program continue_procedure;

var
      cnt : integer;

begin
      for cnt := 1 to 100 do
    begin
      if cnt < 51 then
            continue;
      writeln('cnt = ', cnt);
    end;
end.
```

Note how the **continue** procedure in the above program forces the program to immediately continue with the next iteration if **cnt** is a value from 1 through 50. Once **cnt** reaches 51, the program displays the values 51 through 100.

Lesson 40: If Statements

Many times a program has to do different things depending on a certain condition. To meet these conditional demands, the Pascal programming language is equipped with a variety of decision-making statements. The simplest Pascal decision-making statement is the **if..then** statement. Table 4.4 illustrates how an **if..then** statement is constructed.

Table 4.4 *The Pascal* ***if..then*** *statement*

```
if expression then
        statement;
or
if expression then
begin
        statement;
        .
        .
        statement;
end;

Where:
expression      is a valid Boolean expression.
statement       is a valid Pascal statement.
```

The logic behind a Pascal **if..then** statement is almost ridiculously simple. **If** the Boolean expression following the **if** keyword is equal to **True**, **then** the program statement following the **then** keyword is executed.

Listing 4.7 illustrates how an **if** statement is used in an actual Pascal program.

Listing 4.7

```
{ list4-7.pas - Demonstrate the Pascal if..then state-
ment }
program if_then_demo;

var
      number : integer;

begin
      number := 1;
      if number = 1 then
            writeln('number is equal to 1');
      if number = 0 then
            writeln('number is equal to 0');
end.
```

Besides being able to perform an action if a condition is **True**, an **if** statement can also perform another action if a condition is **False** by using an **else** clause. Table 4.5 illustrates how an **if..then..else** statement is constructed.

Table 4.5 *The Pascal* ***if..then..else*** *statement*

```
if expression then
                statement
else
                statement;
or
if expression then
begin
                statement;
                .
                .
                statement;
end
else
begin
                statement;
                .
                .
                statement;
end;

Where:
expression      is a valid Boolean expression.
statement       is a valid Pascal statement.
```

As with an **if..then** statement, the logic behind an **if..then..else** statement is extremely easy to understand. **If** the condition is **True**, **then** the program state-

ment following the **then** keyword is executed, **else** the statement following the **else** keyword is executed.

Listing 4.8 illustrates how an **if..then..else** statement is used in an actual Pascal program.

Listing 4.8

```
{ list4-8.pas - Demonstrate the Pascal if..then..else
statement }
program if_then_else;

var
      number : integer;

begin
      number := 1;
      if number = 1 then
            writeln('number is equal to 1')
      else
            writeln('number isn''t equal to 1');
      number := 0;
      if number = 1 then
            writeln('number is equal to 1')
      else
            writeln('number isn''t equal to 1');
end.
```

Lesson 41: Case Statements

Although **if..then** and **if..then..else** statements are useful for performing actions depending on a condition being either **True** or **False**, many situations arise in a program that require any one of a variety of actions to be performed depending on an ordinal expression's value. To meet this requirement, the Pascal programming language provides the **case** statement. Table 4.6 illustrates how a **case** statement is constructed.

Table 4.6 *The Pascal* ***case*** *statement*

```
case expression of
        case constant : statement;
        case constant : statement;
        .
        .
        case constant : statement;
        else
                statement;
end;
```

Where:

expression	is a valid ordinal expression.
case constant	is a valid ordinal constant.
statement	is a valid Pascal program statement.

The case constants in a Pascal **case** statement can be either a single constant, a group of constants, or a range of constants. If the value of the **case** statement's expression matches any of a constant group's individual constants, the group's associated program statement are executed. Some examples of constant groups are as follows:

```
100, 101, 102
-55, 32, 8
36, 1
```

If the value of the **case** statement's expression falls anywhere within a range of constants, the range's associated program statement are executed. Some examples of constant ranges are as follows:

```
100..300
-5..5
2000...100000
```

Table 4.6 also shows that Pascal supports **else** clauses in a **case** statement. As with the **if..then** statement, **else** clauses in a **case** statement are strictly optional. If used, their associated program statement are only executed if the **case** statement's expression doesn't match any of the case constants. If an **else** clause isn't used and the **case** statement's expression doesn't match any of the case constants, the whole **case** statement is ignored and program execution is continued with the next program statement.

Listing 4.9 demonstrates how a **case** statement is used in an actual Pascal program.

Listing 4.9

```
{ list4-9.pas - Demonstrate the Pascal case statement }
program case_statement;

var
      number : Integer;

begin
      number := 3;
      case number of
            1 : writeln('The number is a 1');
            2 : writeln('The number is a 2');
            3..5 : writeln('The number is a 3, 4, or 5');
            7, 10 : writeln('The number is a 7 or 10');
            8 : writeln('The number is an 8');
            9 : writeln('The number is a 9');
            else
                  writeln('The number isn''t between 1
                  and 10');
      end;
end.
```

Lesson 42: Goto Statements

If program execution must branch to a different part of a program without regard for any condition, the Pascal programming language provides the **goto** statement for performing such an unconditional jump. Table 4.7 illustrates how a

Pascal **goto** statement is constructed. As this table shows, the **goto** statement requires a label to tell it where it should branch to. A Pascal label is any series of digits in the range of 0 to 9999. Optionally, an identifier can be used as a label.

***Table 4.7** The Pascal **goto** statement*

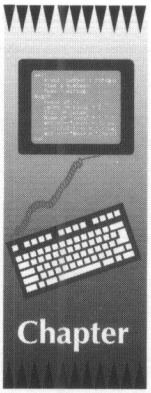
goto label; **Where:** *label* is a valid Pascal label.

Today, the use of the **goto** statement is considered very poor programming practice. Although the **goto** statement is necessary for some other languages (i.e., BASIC), you can go your whole life without finding it necessary to use a **goto** statement in a Pascal program. Perhaps its only acceptable use today is in implementing critical error handling routines. You should always strive to write your programs without using **goto**s.

Listing 4.10 presents a short program that demonstrates how a **goto** statement is used in a Pascal program.

Listing 4.10

```
{ list4-10.pas - Demonstrate the Pascal goto statement }
program goto_demo;

label 1;

begin
      writeln('This line is executed');
      goto 1;
      writeln('This line never is!');
1:
      writeln('This line is executed too');
end.
```



5

Procedures and Functions

In most programs that you write, you will find that certain routines are used repeatedly throughout a program. For example, the following short program demonstrates how an **if..then..else** statement is used again and again throughout a program to conditionally display messages.

Listing 5.1

```
{ list5-1.pas - Display messages program }
program display_messages;

var
      number : integer;

begin
      number := 1;
      if number = 1 then
            writeln('number is equal to 1')
      else
            writeln('number isn''t equal to 1');
      number := 0;
      if number = 1 then
            writeln('number is equal to 1')
      else
            writeln('number isn''t equal to 1');
end.
```

Lesson 43: Declaring Procedures and Functions

Instead of repeatedly writing the same conditional statement over and over, wouldn't it be nice to write it once and be able to use it anywhere in a program. Fortunately, Pascal provides procedures and functions for just such a purpose. Essentially, a procedure or a function is a program within a program. Not only can it have its own body of program statements, a procedure or a function can have its own variables, typed constants, and even its own procedures and functions. Table 5.1 illustrates how a procedure is declared in a Pascal program and Table 5.2 illustrates how a function is declared in a Pascal program.

Table 5.1 *A Pascal procedure declaration*

```
procedure name (parameter list);
begin
        statement;
        .
        .
        statement;
end;
```

Where:

name	is the procedure's identifier.
parameter list	is a list of arguments to be passed to the procedure.
statement	is a valid Pascal program statement.

Table 5.2 *A Pascal function definition*

```
function name(parameter list) : return type;
begin
        statement;
        .
        .
        statement;
end;
```

Where:

name	is the function's identifier.
parameter list	is a list of arguments to be passed to the procedure.
return type	is a previously defined data type.
statement	is a valid Pascal program statement.

As both tables illustrate, a procedure or a function's associated program statements are enclosed in a **begin..end** statement block just like the program's main body. Tables 5.1 and 5.2 also show that procedures and functions can have an optional parameter list. A procedure or function's parameters are used to pass values to the procedure or function. The following are some examples of parameter lists:

```
(row, col : integer; message : string)
(x, y : integer)
(name : string)
```

Note in the above examples that the parameters are defined just like they would be in a variable definition. Each data type is separated by a semicolon just like a normal program statement. However, the final parameter declaration in a parameter list doesn't require a semicolon.

To illustrate how a procedure is actually used in a Pascal program, Listing 5.2 presents a short program that performs exactly like the one in Listing 5.1. Unlike the program in Listing 5.1, this newer version uses a procedure to replace the multiple **if..then..else** statements.

Listing 5.2

```
{ list5-2.pas - Display messages program version 2}
program display_messages2;

procedure display(n : integer);
begin
      if n = 1 then
            writeln('The number is equal to 1')
      else
            writeln('The number isn''t equal to 1');
```

Listing 5.2—Continued

```
end;

var
      number : integer;

begin
      number := 1;
      display(number);
      number := 0;
      display(number);
end.
```

Although Listing 5.2 is a very simple example of how a procedure is used in a Pascal program, it does illustrate a number of important points about using a procedure or a function in a program. Let's take a line-by-line look at both the **display** procedure and the program's main body.

```
procedure display(n : integer);
```

defines a procedure named **display** that has one integer argument identified by **n**.

```
begin
```

defines the start of **display**'s statement block.

```
if n = 1 then
     writeln('The number is equal to 1')
 else
          writeln('The number isn''t equal to 1');
```

displays the message **The number is equal to 1** if argument **n** is equal to **1**. Otherwise, it displays the message **The number isn't equal to 1** if **n** isn't equal to **1**.

```
end;
```

defines the end of **display**'s statement block.

```
begin
```

defines the start of the program's main body.

```
number := 1;
```

assigns the value **1** to the **integer** variable **number**.

```
display(number);
```

calls the procedure **display**. Additionally, the value of the **integer** variable **number** is passed as **display**'s one and only argument.

```
number := 0;
```

assigns the value **0** to the **integer** variable **number**.

```
display(number);
```

calls the procedure **display**. Additionally, the value of the **integer** variable **number** is passed as **display**'s one and only argument.

```
end.
```

defines the end of the program's main body.

Lesson 44: Function Return Values

Now that we've seen how a simple procedure is written, let's take a look at how a simple function is written. As Table 5.2 illustrated, a function declaration requires that a return type be specified. For example, a function that returns an **integer** value would be declared as returning an **integer** data type.

Although declaring the function's return type is a fairly simple task, it is not so obvious how the function's return value is actually returned to the calling program. Fortunately for the Pascal programmer, a value is returned by simply assigning the function's return value to the function's identifier. Essentially, the function's identifier acts like a variable with the data type defined as the function's return type. For example, a function named **intadd** would return a value of **2** to the calling program as follows:

```
intadd := 2;
```

The above example clearly shows that returning a value to the calling program requires nothing more than a simple assignment statement. To further illustrate how a function is used in a Pascal program, Listing 5.3 presents a brief program, which calls a simple function that multiplies a passed argument by 2 and returns the result.

Listing 5.3

```
{ list5-3.pas - Demonstrates Pascal functions }
program function_calls;

function times_two(n : integer) : integer;
begin
      times_two := n * 2;
end;

begin
      writeln(times_two(4));
      writeln(times_two(16));
end.
```

In order to fully understand how the above program performs its task, let's examine the **times_two** function and the program's main body a line at a time.

```
function times_two(n : integer) : integer;
```

defines a function named **times_two** that has one **integer** argument identified by **n** and returns an **integer** value.

```
begin
```

defines the start of **times_two**'s statement block.

```
times_two := n * 2;
```

multiplies argument **n** by 2 and assigns it to the function's identifier **times_two**.

```
end;
```

defines the end of the **times_two** statement block.

```
begin
```

defines the start of the program's main body.

```
writeln(times_two(4));
```

displays the result of multiplying 4 times 2.

```
writeln(times_two(16));
```

displays the result of multiplying 16 times 2.

```
end.
```

defines the end of the program's main body.

Lesson 45: Forward Declarations

Let's suppose that you wanted to write a program with a procedure that calls another procedure. Obviously, this would be a very common occurrence in a Pascal program. Is there a special way you have to write such a program? As a matter of fact, there is. You can choose to write your program using either one of two methods. The easiest method is demonstrated below in Listing 5.4.

Listing 5.4

```
{ list5-4.pas - Demonstrate procedure calling proce-
dures }
program procedure_calling;

procedure first;
begin
      writeln('This is the first procedure.');
end;

procedure second;
begin
      first;
      writeln('This is the second procedure.');
end;

begin
      second;
end.
```

As the above program illustrates, Pascal requires that a procedure that is called by another procedure be defined before the procedure that calls it. What would happen if the called procedure isn't defined first? The simplest way to find out is to enter the program in Listing 5.5 and to compile it.

Listing 5.5

```
{ list5-5.pas - Demonstrate an incorrect declaration }
program incorrect_declaration;

procedure second;
begin
      first;
      writeln('This is the second procedure.');
end;

procedure first;
begin
      writeln('This is the first procedure.');
end;

begin
      second;
end.
```

With the exception of defining procedure **second** before procedure **first**, the above program is the same program that was presented in Listing 5.4. If you took the time to actually enter and compile the above program, you were

informed that the **first** identifier in procedure **second** is an unknown identifier. Why is **first** an unknown identifier? Simply because it has no meaning in the program yet. Without defining procedure **first** before procedure **second**, the Pascal compiler has no way of knowing just what the identifier **first** represents. As far as the compiler is concerned, **first** could be a variable, a constant, or just about anything else a Pascal identifier is used for. So obviously, the compiler has to abort the compilation process and return the rather disappointing error message.

Fortunately for the Pascal programmer, there is a way around having to define a called procedure or function before the calling procedure or function. The method used to perform this trick is called a *forward declaration.* Table 5.3 displays how a forward declaration is made. As this table show, a forward declaration is created by following a procedure or function head with the keyword **forward**. Once this has been done, any other procedure or function is able to call the forward declaration's associated procedure or function.

Table 5.3 *A forward declaration*

procedure name(parameter list) ; **forward;**

or

function name(parameter list) : return type; **forward;**

Where:

name	is the procedure's or function's identifier.
return type	is a previously defined data type.
parameter list	is a list of arguments to be passed by the procedure or function.

Listing 5.6 illustrates a short program, which demonstrates how the program in Listing 5.5 could be correctly rewritten by adding a forward declaration before procedure **second**.

Listing 5.6

```
{ list5-6.pas - Demonstrate a forward declaration }
program forward_declaration;

procedure first; forward;

procedure second;
begin
      first;
      writeln('This is the second procedure.');
end;

procedure first;
begin
      writeln('This is the first procedure.');
end;

begin
      second;
end.
```

Lesson 46: Local Variables

As it was stated earlier in this chapter, a Pascal procedure or function can have its own variables. These procedure and function variables are called *local variables.* They are called local variables because they can only be used inside of the procedure or function. The reason for this is something called *scope,* which is covered in the next lesson. Table 5.4 illustrates how variables are defined in a procedure or function.

Table 5.4 *Procedure and function variables*

```
procedure or function head;
var
        variable declaration;
        .
        .
        variable declaration;
begin
        statement;
        .
        .
        statement;
end;
```

Where:

procedure or function head	is a valid procedure or function head.
variable declaration	is a valid variable declaration.
statement	is a valid program statement.

As Table 5.4 illustrates, the variable declarations are placed right in between the procedure or function's head and its associated body. Typed constants can also be declared in a procedure or a function by using this same method. Listing 5.7 presents a short program that demonstrates how local variables are used in an actual Pascal program.

Listing 5.7

```
{ list5-7.pas - Demonstrate local variables }
program local_variables;

procedure count;
var
      i : integer;

begin
      i := 1;
      while i < 11 do
      begin
            writeln(i);
            i := i + 1;
      end;
end;

begin
      count;
      count;
end.
```

What does the above program do? It simply uses the local variable **i** to count from 1 to 10 each time the procedure **count** is called.

Listing 5.8 presents a short program that demonstrates how typed constants are used in an actual Pascal program.

Listing 5.8

```
{ list5-8.pas - Demonstrate local typed constants }
program local_typed_constants;

procedure count;
const
      i : integer = 1;

begin
      writeln(i);
      i := i + 1;
end;

begin
      count;
      count;
end.
```

The above program demonstrates a rather interesting fact about typed constants. Always remember that typed constants retain their value until the program finishes executing. Thus, the first time the procedure **count** is called it displays a value of **1** for the typed constant **i**. Furthermore, **count** displays a value of **2** the second time it is called, a value of **3** the third time it is called, and so on.

Although the above program demonstrates how a typed constant retains its value between procedure and function calls, you may be wondering what happens with a procedure's or a function's variables between function calls. Like a typed constant, do they still retain there values? Simply put, no. Like a variable in the main program, a local variable is undefined at the start of the procedure or function call. Listing 5.9 presents a short program that demonstrates how a local variable is in an undefined state at the start of procedure call. With the single exception of declaring **i** as a variable instead of a typed constant, this

program is the same as the one presented in Listing 5.8. However, this simple change clearly reflects how a local variable is considered undefined each and every time a procedure or a function is called.

Listing 5.9

```
{ list5-9.pas - Demonstrate how variables are undefined }
program undefined;

procedure count;
var
      i : integer;

begin
      writeln(i);
      i := i + 1;
end;

begin
      count;
      count;
end.
```

Lesson 47: Scope

In the previous lesson, it was mentioned that procedure and function variables are called local variables because of something called scope. In this lesson, we look at how a variable's scope affects what parts of a program can access it. As you already know, procedure and function variables are called local variables. The term local variable makes sense when you consider that only the variable's procedure or function can access them. Thus, all procedure and function variables are said to have local scope.

But what about the variables that are defined outside of a procedure or function? Any variable that is defined outside of a procedure or a function is called a *global variable*. Essentially, a global variable can be accessed by any procedure, function, or parts of the program's main body that follows its declaration. Listing 5.10 presents a brief program that demonstrates how a variable with global scope can be accessed by a procedure.

Listing 5.10

```
{ list5-10.pas - Demonstrate global scope }
program global_scope;

var
      i : integer;

procedure display_i;
begin
      writeln(i);
      i := i + 1;
end;

begin
      i := 1;
      display_i;
      writeln(i);
end.
```

Not only does the procedure in Listing 5.10 **display_i** display global variable **i**'s value, it increments **i** before returning to the program's main body. After execution is returned to the program's main body, **i**'s new value is displayed.

But wait a second, what if a procedure or a function had a local variable named **i** in addition to a global variable **i**? Not only is a situation like this apt to arise in a program, it is a fairly common occurrence in most programs. So how does a procedure or function know which **i** to choose from? Quite simply, the procedure or function always uses its local variable **i**. This concept of a procedure or function picking a variable with local scope is clearly demonstrated by the program presented in Listing 5.11.

Listing 5.11

```
{ list5-11.pas - Demonstrate global vs. local scope }
program global_vs_local_scope;

var
      i : integer;

procedure display_i;
var
      i : integer;

begin
      i := 999;
      writeln(i);
      i := i + 1;
end;

begin
      i := 1;
      display_i;
      writeln(i);
end.
```

The above program illustrates that the procedure **display_i** has no effect whatsoever on the global variable **i**. It simply displays its own local variable **i**, needlessly increments the local variable **i**, and returns to the program's main body. When execution is returned to the program's main body, the global variable **i** is displayed to demonstrate that it hasn't been changed.

Lesson 48: Arguments

In the previous lessons, we have seen that arguments (or parameters) can be passed to either a procedure or a function. Normally, an argument is **passed by value**. In stating that an argument is passed by value, it simply means that the argument's value is passed to the function. This may all seem rather obvious, but it is an important point to make. Take the short program in Listing 5.12 for example. This program passes global variable **n**'s value to the procedure **count**. Once it is passed to the procedure, the argument's value is displayed and decremented over and over until it is less than zero. Note that upon return from procedure **count**, global variable **n**'s value is displayed to prove that it hasn't been changed at all by the procedure.

Listing 5.12

```
{ list5-12.pas - Demonstrate passing by value }
program pass_by_value;

var
     n : integer;

procedure count(number : integer);
begin
     repeat
          writeln(number);
          number := number - 1;
```

Listing 5.12—Continued

```
        until number < 0;
end;

begin
        n := 10;
        count(n);
        writeln(n);
end.
```

Although passing by value is probably the most commonly used method Pascal programmers employ when passing arguments, Pascal offers another method for passing arguments called *passing by reference*. Arguments are passed by reference by simply placing the **var** keyword before the argument declaration in the procedure or function head. The following are some examples of pass by reference argument declarations:

```
var row, col : integer;
var account : real;
```

Passing an argument by reference actually passes the argument's memory location and not its value. With the argument's memory location at its disposal, the procedure or function is able to directly access and modify the passed argument. Because the procedure or function needs the argument's actual location in memory, an expression can't be passed by reference. In fact, it wouldn't make any sense to pass an expression by reference anyway. Why would you ever want to modify the value of an expression that lies outside of the procedure or function? Once its value has been passed to the procedure or function, the expression serves no useful purpose. So remember to always pass expressions by value.

To illustrate how an argument is passed by reference, Listing 5.13 presents a modified version of the program that was presented in Listing 5.12. Unlike the

previous version of the program, this modified version passes the procedure **count**'s argument **number** by reference. This results in global variable **n**'s value being modified by procedure **count**. Upon return from the procedure **count,** global variable **n** has a value of -1 and not the value of 10 that it had with the previous version of the program.

Listing 5.13

```
{ list5-13.pas - Demonstrate passing by reference }
program pass_by_reference;

var
      n : integer;

procedure count(var number : integer);
begin
      repeat
            writeln(number);
            number := number - 1;
      until number < 0;
end;

begin
      n := 10;
      count(n);
      writeln(n);
end.
```

Lesson 49: Nested Procedures and Functions

Besides being able to have their own variables, a Pascal procedure or function can also have their own procedures and functions. The easiest way to imagine how a procedure or function can have its own procedures and functions is to think of the Pascal program as nothing more than an extra large procedure. Within that big procedure you can define other procedures and functions. Table 5.5 illustrates how a procedure or a function can have their own procedures and functions.

Table 5.5 *Nested Pascal procedures and functions*

```
procedure or function head;
        procedure or function head;
        begin
                statement;
                .
                .
                statement;
        end;
begin
        statement;
        .
        .
        statement;
end;
```

Where:

procedure or function head	is a valid procedure or function head.
statement	is a valid program statement.

Listing 5.14 presents a brief program that demonstrates how a function can be nested in a procedure.

Listing 5.14

```
{ list5-14.pas - Demonstrate nested procedures and
functions }
program nested_p_and_f;

procedure display(n : integer);
var
      i : integer;

      function addone(n :  integer) : integer;
      begin
            addone := n + 1;
      end;

begin
      for i := 1 to 10 do
      begin
            writeln(n);
            n := addone(n);
      end;
end;

begin
      display(1);
end.
```

Essentially, the above program displays a number, adds one to it, and repeats the process nine more times. When studying this program, you should remember to use the analogy that a Pascal program is nothing more than a big procedure or conversely a procedure is nothing more than a miniature program. Keeping that in mind makes writing your own nested procedures and functions a snap.

To write your own nested procedures and functions correctly, you must also understand how Pascal's scope rules apply to procedures and functions. Like a procedure or a function's local variables, a nested procedure or function is local to the procedure or function it is defined in. The program in Listing 5.15 demonstrates how the Pascal scope rules work by defining two functions with the same name. Because of the scope rules, the function **addone** inside of the procedure **display** is called and not the global function **addone**.

Listing 5.15

```
{ list5-15.pas - Demonstrate procedure and function }
program p_and_f_scope;

function addone(n : integer) : integer;
begin
      addone := n + 101;
end;

procedure display(n : integer);
var
      i : integer;

      function addone(n :  integer) : integer;
      begin
            addone := n + 1;
      end;
begin
      for i := 1 to 10 do
      begin
            writeln(n);
```

Listing 5.15—Continued

```
            n := addone(n);
      end;
end;

begin
      display(1);
end.
```

Lesson 50: Recursion

Before we leave this chapter on procedures and functions, we have to explore one last feature that Pascal procedures and functions possess. This feature is called *recursion* and allows a Pascal procedure or function to call itself repeatedly. Although this may not seem to be such an important feature, recursion can greatly simplify writing some of the most important computer programming routines (i.e., quick sort, b-trees, etc.).

Listing 5.16 displays a short program that demonstrates how a Pascal procedure can recursively call itself.

Listing 5.16

```
{ list5-16.pas - Demonstrate recursion }
program recursion;

procedure count(n : integer);
begin
      writeln(n);
      n := n - 1;
      if n >= 0 then
            count(n);
end;

begin
      count(20);
end.
```

Although the above program is really quite simple, let's take a detailed look at how the procedure **count** is used to count backwards from a passed argument.

```
begin
```

defines the start of the procedure **count**'s body.

```
writeln(n);
```

displays the **integer** argument **n**'s value.

```
n := n - 1;
```

decrements **n**'s value.

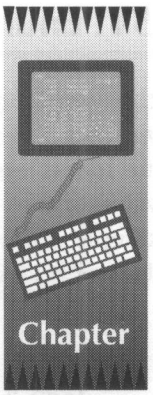
```
if n >= 0 then
    count(n);
```

checks argument **n**'s value to see if it's still greater than or equal to 0. If it is still greater than or equal to zero, **count** calls itself with **n**'s new value for its argument.

```
end;
```

defines the end of the procedure **count**'s body.

You are probably thinking at this point, "Couldn't this routine be done much easier using a simple loop?" Well yes, it could be. However, as I previously stated there are a number of important computer programming routines that are much easier to write using recursion than with more traditional programming methods. Consequently, it is essential for all Pascal programmers to understand how recursion works.



6

User-Defined Data Types

In Chapter 2, you learned what Pascal's standard data types are and how they are used. This chapter gives you a whole new way of looking at program data. That's because this chapter shows how the Pascal programmer can define his own data types. One of Pascal's main advantages over other programming languages is this ability that allows the programmer to define his own data types. Therefore, the Pascal programmer must be well versed in how user-defined data types are created.

Lesson 51: Enumerated Data Types

The first of the user-defined data types we look at in this chapter are called *enumerated data types.* Essentially, an enumerated data type is constructed from a list of unique identifiers. Each of the enumerated data type's identifiers are assigned a value of **0** to **n**. Where **n** represents the number of identifiers minus one. For example, an enumerated data type with 10 identifiers would have assigned values of 0 to 9. Table 6.1 illustrates how an enumerated data type is defined.

Table 6.1 *Defining a Pascal enumerated data type*

type	
data type identifier = (identifier list);	
Where:	
data type identifier	is a valid identifier.
identifier list	is a list of valid identifiers. If more than one identifier is specified, they are separated by commas.

Note in Table 6.1, that an enumerated data type definition follows the **type** keyword. Some examples of enumerated data type declarations are as follows:

```
suit = (clubs, spades, hearts, diamonds);
computers = (IBM, Apple, Tandy, Commodore, Acer, Dell);
```

Listing 6.1 illustrates a short program that demonstrates how an enumerated data type is used in an actual Pascal program.

Listing 6.1

```
{ list6-1.pas - Demonstrate enumerated data types }
program enum_data;

type
      computers = (IBM, Apple, Tandy, Commodore, Other);

var
      Jim, David : computers;

procedure display_brand(brand : computers);
begin
      case brand of
            IBM:
                  writeln(' has an IBM');
            Apple:
                  writeln(' has an Apple');
            Tandy:
                  writeln(' has a Tandy');
            Commodore:
                  writeln(' has a Commodore');
            else
                  writeln(' doesn''t have an IBM,
Apple, Tandy, or Commodore');
      end
end;
```

Listing 6.1—Continued

```
begin
      Jim := IBM;
      David := Other;
      write('Jim');
      display_brand(Jim);
      write('David');
      display_brand(David);
end.
```

Lesson 52: The Dec Procedure

To allow programmers to easily handle enumerated data types or any other ordinal data types (data types that can only be expressed as a series of whole numbers: integers, characters, etc.), the Pascal programming language comes equipped with a variety of ordinal related procedures and functions. The first ordinal procedure we study is the **dec** procedure. The Pascal **dec** procedure simply subtracts one from an ordinal variable's value. Table 6.2 displays how the Pascal **dec** procedure is used.

***Table 6.2** The Pascal **dec** procedure*

dec(ordinal variable); **Where:** *ordinal variable* is a valid Pascal ordinal variable.

Listing 6.2 presents a brief program that demonstrates how the Pascal **dec** procedure is used in an actual program.

Listing 6.2

```
{ list6-2.pas - Demonstrate the Pascal dec function }
program dec_demo;

type
      cards = (clubs, diamonds, spades, hearts);

procedure display_suit(card : cards);
begin
      case card of
            clubs:
                  writeln('The card is a club');
            diamonds:
                  writeln('The card is a diamond');
            spades:
                  writeln('The card is a spade');
            hearts:
                  writeln('The card is a heart');
      end;
end;

var
      c1 : cards;

begin
      c1 := hearts;
      display_suit(c1);
      dec(c1);
      display_suit(c1);
end.
```

Lesson 53: The Inc Procedure

In the last lesson, you learned that the Pascal **dec** procedure subtracts one from an ordinal variable's value. Conversely, the Pascal **inc** procedure adds one to an ordinal variable's value. Table 6.3 illustrates how the Pascal **inc** procedure is used.

***Table 6.3** The Pascal **inc** procedure*

inc(ordinal variable); **Where:** *ordinal variable* is a valid Pascal ordinal variable.

Listing 6.3 displays a short program that demonstrates how the Pascal **inc** procedure is used in an actual program.

Listing 6.3

```
{ list6-3.pas - Demonstrate the Pascal inc function }
program inc_demo;

type
      cards = (clubs, diamonds, spades, hearts);

procedure display_suit(card : cards);
begin
      case card of
            clubs:
                  writeln('The card is a club');
            diamonds:
                  writeln('The card is a diamond');
```

Listing 6.3—Continued

```
            spades:
                writeln('The card is a spade');
            hearts:
                writeln('The card is a heart');
        end;
end;

var
        c1 : cards;

begin
        c1 := clubs;
        display_suit(c1);
        inc(c1);
        display_suit(c1);
end.
```

Lesson 54: The Pred Function

The Pascal **pred** function returns the value of an ordinal expression minus one. Table 6.4 illustrates how the Pascal **pred** function is used.

***Table 6.4** The Pascal **pred** function*

pred(ordinal expression0;
Where:
ordinal expression is a valid Pascal ordinal expression.

Listing 6.4 displays a short program that shows how the **pred** function is used in an actual Pascal program.

Listing 6.4

```
{ list6-4.pas - Demonstrate the Pascal pred function }
program pred_demo;

type
      cards = (clubs, diamonds, spades, hearts);

procedure display_suit(card : cards);
begin
      case card of
            clubs:
                  writeln('The card is a club');
            diamonds:
                  writeln('The card is a diamond');
            spades:
                  writeln('The card is a spade');
            hearts:
                  writeln('The card is a heart');
      end;
end;

var
      c1 : cards;

begin
      c1 := hearts;
      display_suit(c1);
      display_suit(pred(c1));
end.
```

Lesson 55: The Succ Function

Pascal provides both a **dec** and an **inc** procedure, Pascal also offers a compliment to the **pred** function called the **succ** function. Essentially, the Pascal **succ** function returns the value of an ordinal expression plus one. Table 6.5 illustrates how the Pascal **succ** function is used.

Table 6.5 *The Pascal* ***succ*** *function*

succ(ordinal expression);	
Where:	
ordinal expression	is a valid Pascal ordinal expression.

Listing 6.5 displays a brief program that shows how the **succ** function is used in an actual Pascal program.

Listing 6.5

```
{ list6-5.pas - Demonstrate the Pascal succ function }
program succ_demo;

type
     cards = (clubs, diamonds, spades, hearts);

procedure display_suit(card : cards);
begin
     case card of
          clubs:
               writeln('The card is a club');
          diamonds:
               writeln('The card is a diamond');
```

Listing 6.5—Continued

```
            spades:
                  writeln('The card is a spade');
            hearts:
                  writeln('The card is a heart');
      end;
end;

var
      c1 : cards;

begin
      c1 := clubs;
      display_suit(c1);
      display_suit(succ(c1));
end.
```

Lesson 56: Subranges

Many times the Pascal programmer wants to limit a data type's range of values. For example, a data value representing the months of a year would only need values in the range of 1 to 12. Fortunately for the Pascal programmer, Pascal allows the programmer to define new data types by using a subrange of another previously defined ordinal data type. Therefore, the Pascal programmer can construct subrange data types from integers, characters, or enumerations. Table 6.6 illustrates how a subrange data type is defined.

Table 6.6 *Defining a subrange data type*

type
identifier = minimum value..maximum value;
Where:
identifier is the data type's identifier.
minimum value is the smallest allowable value in the subrange.
maximum value is the largest allowable value in the subrange.

The following are some examples of valid integer and character subrange data type definitions:

```
rows = 1..25;
columns = 1..80;
dice = 1..6;
month = 1..12;
day = 1..31;
numeric = '0'..'9';
control = #0..#31;
extended = #128..#255;
```

To define an enumerated subrange, you must define the enumerated data type first. For example, the following enumerated subranges could be defined from the enumerated data type **computer = (IBM, Tandy, Dell, Commodore, Apple);:**

```
ibm_and_compats = IBM..Dell;
non_ibm = Commodore..Apple;
```

Listing 6.6 displays a brief program that demonstrates how subranges are used in an actual Pascal program.

Listing 6.6

```
{ list6-6.pas - Demonstrate Pascal subrange data types }
program subranges;

type
      dice = 1..6;

var
      die1, die2 : dice;

begin
      randomize;
      repeat
            die1 := random(6) + 1;
            die2 := random(6) + 1;
            writeln('die1 = ', die1);
            writeln('die2 = ', die2);
      until die1 = die2;
end.
```

Essentially, the above program creates a subrange data type that can represent all of the legal values on a die. The program demonstrates this data type by assigning randomly generated rolls until two **dice** variables are equal. In other words, the program will keep rolling the dice until doubles come up.

Lesson 57: Sets

Many Pascal programs require handling data that doesn't seem to have a particular order to it. To deal with such unordered data, the Pascal programming lan-

guage provides a user-defined data type called a *set*. A Pascal set is made up of either all of the members of an ordinal data type or a subrange of an ordinal data type. Furthermore, a Pascal set cannot represent more than 256 distinct values. Consequently, only subranges can be used to define sets of **Word**s, **Integer**s, and **LongInt**s. Table 6.7 illustrates how a Pascal set is defined.

Table 6.7 *Defining a Pascal set*

type	
identifier = **set of** ordinal data type;	
Where:	
identifier	is the data type's identifier.
ordinal data type	is an ordinal data type or subrange.

The following are some examples of valid Pascal set definitions:

```
digits = set of '0'..'9';
logical = set of boolean;
lower_case = set of 'a'..'z';
```

Table 6.8 *Set assignments*

identifier := [element list];	
Where:	
identifier	is a variable or typed constant identifier.
element list	is a list of individual values, subranges, or both. Multiple elements are separated by a comma.

Table 6.8 shows how values are assigned to set variables and typed constants. The following are some examples of valid Pascal set assignment statements:

```
vowels := ['A', 'E', 'I', 'O', 'U'];
seta := [1, 4, 6, 7, 9, 10];
setb := [1, 4, 5...9, 100, 201];
```

To deal with sets, the Pascal programming language provides a variety of set related operators. The remainder of this chapter is devoted to studying how these Pascal set operators function.

Lesson 58: The Set Equal To Operator

Like the normal Pascal equal to operator, the *set equal to operator* (=) compares two set expressions to see if they are equal. If the two set expressions are equal, the set equal to operator returns a value of **True**. Otherwise, the set equal to operator returns value of **False** to indicate that the two expressions aren't equal. Table 6.9 illustrates the use of the set equal to operator.

Table 6.9 *The Pascal set equal to operator*

set expression = set expression	
Where:	
expression	is a valid Pascal set expression.

Listing 6.7 displays a short program that illustrates how the Pascal set equal to operator is used in an actual program.

Listing 6.7

```
{ list6-7.pas - Demonstrate the Pascal set equals
operator }
program set_equals;

type
      characters = set of char;

var
      set1, set2, set3 : characters;

begin
      set1 := ['a'..'z'];
      set2 := ['A'..'Z'];
      set3 := set1;
      writeln('set1 = set2 is ', set1 = set2);
      writeln('set1 = set3 is ', set1 = set3);
end.
```

Lesson 59: The Set Not Equal To Operator

The Pascal *set not equal to operator* (<>) compares two set expressions to see if they are unequal in value. If the two set expressions aren't equal in value, the set not equal to operator returns a value of **True**. Otherwise, the set not equal to operator returns a value of **False** to indicate that the two set expressions are equal in value. Table 6.10 illustrates the use of the set not equal to operator.

Table 6.10 *The Pascal set not equal to operator*

set expression <> set expression **Where:** *set expression* is a valid Pascal set expression.

Listing 6.8 displays a brief program that illustrates how the Pascal set not equal to operator is used in an actual program.

Listing 6.8

```
{ list6-8.pas - Demonstrate the Pascal set does not
equal operator }
program set_do_not_equal;

type
      characters = set of char;

var
      set1, set2, set3 : characters;

begin
      set1 := ['a'..'z'];
      set2 := ['A'..'Z'];
      set3 := set1;
      writeln('set1 <> set2 is ', set1 <> set2);
      writeln('set1 <> set3 is ', set1 <> set3);
end.
```

Lesson 60: The Set Less Than Or Equal To Operator

The Pascal *set less than or equal to operator* (<=) compares two set expressions to see if all of the elements of the first set expression are in the second set expression. If the first set expression is less than or equal to the second set expression, the set less than or equal to operator returns a value of **True**. Otherwise, the set less than or equal to operator returns a value of **False** to indicate that the first set expression has element(s) that aren't contained in the second set expression. Table 6.11 illustrates the use of the set less than or equal to operator.

Table 6.11 *The Pascal set less than or equal to operator*

set expression <= set expression
Where:
set expression is a valid Pascal set expression.

Listing 6.9 displays a short program that demonstrates how the Pascal set less than or equal to operator is used in an actual program.

Listing 6.9

```
{ list6-9.pas - Demonstrate the Pascal set less than
or equal to operator }
program set_less_than_or_equal;

type
      characters = set of char;

var
      set1, set2, set3 : characters;
```

Listing 6.9—Continued

```
begin
      set1 := ['a'..'z', 'A'..'Z'];
      set2 := ['A'..'Z'];
      set3 := ['a'..'z', #13];
      writeln('set2 <= set1 is ', set2 <= set1);
      writeln('set3 <= set1 is ', set3 <= set1);
end.
```

Lesson 61: The Set Greater Than Or Equal To Operator

The Pascal *set greater than or equal to operator* (**>=**) compares two set expressions to see if all of the elements of the second set expression are in the first set expression. If the first set expression has at least all of the elements contained in the second set expression, the set greater than or equal to operator returns a value of **True**. Otherwise, the set greater than or equal to operator returns a value of **False** to indicate that the first set expression doesn't have all of the elements found in the second set expression. Table 6.12 illustrates the use of the set greater than or equal to operator.

Table 6.12 *The Pascal set greater than or equal to operator*

set expression >= set expression
Where:
set expression is a valid Pascal set expression.

Listing 6.10 display a brief program that demonstrates how the Pascal set greater than or equal to operator is used in an actual program.

Listing 6.10

```
{ list6-10.pas - Demonstrate the Pascal set greater
than or equal to operator }
program set_greater_than_or_equal;

type
      characters = set of char;

var
      set1, set2, set3 : characters;

begin
      set1 := ['a'..'z', 'A'..'Z'];
      set2 := ['A'..'Z'];
      set3 := ['a'..'z', #13];
      writeln('set2 >= set1 is ', set2 <= set1);
      writeln('set3 >= set1 is ', set3 <= set1);
end.
```

Lesson 62: The Set In Operator

The Pascal **in** operator tests to see if the result of an ordinal expression is an element of a set expression. If the ordinal value is contained in the set, the **in** operator returns a value of **True**. Otherwise, the **in** operator returns a value of **False** to indicate that the ordinal value isn't an element of the set. Table 6.13 illustrates the use of the **in** operator.

Table 6.13 *The Pascal* ***in*** *operator*

ordinal expression **in** set expression	
Where:	
ordinal expression	is a valid Pascal ordinal expression.
set expression	is a valid Pascal set expression.

Listing 6.11 displays a brief program that illustrates how the Pascal **in** operator is used in an actual program.

Listing 6.11

```
{ list6-11.pas - Demonstrate the Pascal in operator }
program in_demo;

procedure check_vowels(c : char);
const
      vowels : set of char = ['A', 'E', 'I', 'O', 'U',
                              'a', 'e', 'i', 'o', 'u' ];

begin
      if (c = 'Y') or (c = 'y') then
            writeln('Y is sometimes a vowel')
      else
            if c in vowels then
                  writeln(c, ' is a vowel')
            else
                  writeln(c, ' is a consonant');
```

Listing 6.11—Continued

```
end;

begin
      check_vowels('a');
      check_vowels('z');
      check_vowels('Y');
end.
```

Lesson 63: The Set Union Operator

The Pascal *set union operator* (+) returns the result of combining the elements of one set expression with the elements of another set expression. Table 6.14 illustrates the use of the set union operator.

Table 6.14 *The Pascal set union operator*

set expression + set expression	
Where:	
set expression	is a valid Pascal set expression.

Listing 6.12 displays a short program that illustrates how the Pascal set union operator is used in an actual program.

Listing 6.12

```
{ list6-12.pas - Demonstrate the Pascal set union
operator }
program set_union_demo;

type
      digits = set of '0'..'9';

var
      set1, set2 : digits;

begin
      set1 := ['0'..'2'];
      set2 := ['4'..'9'] + set1;
      writeln('''1'' in set2 = ', '1' in set2);
      writeln('''3'' in set2 = ', '3' in set2);
      writeln('''5'' in set2 = ', '5' in set2);
end.
```

Lesson 64: The Set Difference Operator

The Pascal *set difference operator* (-) returns the result of removing the elements of a second set expression from a first set expression. Table 6.15 illustrates the use of the set difference operator.

Table 6.15 *The Pascal set difference operator*

set expression - set expression **Where:** *set expression* is a valid Pascal set expression.

Listing 6.13 displays a brief program that demonstrates how the Pascal set difference operator is used in an actual program.

Listing 6.13

```
{ list6-13.pas - Demonstrate the Pascal set difference
operator }
program set_difference_demo;

type
      digits = set of '0'..'9';

var
      set1, set2 : digits;

begin
      set1 := ['0'..'2'];
      set2 := ['0'..'9'] - set1;
      writeln('''1'' in set2 = ', '1' in set2);
      writeln('''3'' in set2 = ', '3' in set2);
      writeln('''5'' in set2 = ', '5' in set2);
end.
```

Lesson 65: The Set Intersection Operator

The Pascal *set intersection operator* (*) returns a set that is constructed from the elements that are common to two set expressions. Table 6.16 illustrates the use of the set intersection operator.

Table 6.16 *The Pascal set intersection operator*

set expression * set expression
Where:
set expression is a valid Pascal set expression.

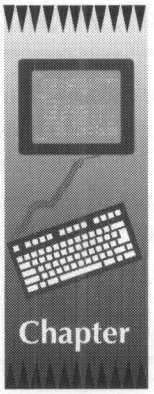
Listing 6.14 displays a short program that demonstrates how the Pascal set intersection operator is used in an actual program.

Listing 6.14

```
{ list6-14.pas - Demonstrate the Pascal set intersec-
tion operator }
program set_intersection_demo;

type
      digits = set of '0'..'9';

var
      set1, set2 : digits;

begin
      set1 := ['0'..'3'];
      set2 := ['2'..'9'] * set1;
```

Listing 6.14—Continued

```
        writeln(''''1'' in set2 = ', '1' in set2);
        writeln(''''3'' in set2 = ', '3' in set2);
        writeln(''''5'' in set2 = ', '5' in set2);
end.
```



7

Arrays

Through the first six chapters of this book, you have studied a wide variety of methods for representing data. Although all of these data types differ from each other a great deal, they all share one key characteristic in that they can only represent one piece of data at a time. This chapter demonstrates how numerous data items of the same data type can all be joined together under one identifier name by declaring the identifier to be an *array*.

Lesson 66: A Simple Array

To start off our study of arrays, lets look at the program presented in Listing 7.1. This simple program shows how a student's grades could be stored in ten integer variables. After safely tucking away the student's grades in the variables, the program figures the sum of all the grades and uses the result to figure the student's grade average.

Listing 7.1

```
{ list7-1.pas - Figure student's average no. 1 }
program stud_avg_1;


var
      g1, g2, g3, g4, g5, g6, g7, g8, g9, g10 : inte-
ger;
      total, ave : integer;


begin
      g1 := 90;
      g2 := 89;
      g3 := 100;
      g4 := 97;
      g5 := 85;
      g6 := 99;
      g7 := 96;
      g8 := 100;
      g9 := 94;
      g10 := 100;
      total := g1;
```

Listing 7.1—Continued

```
        total := total + g2;
        total := total + g3;
        total := total + g4;
        total := total + g5;
        total := total + g6;
        total := total + g7;
        total := total + g8;
        total := total + g9;
        total := total + g10;
        ave := total div 10;
        writeln('The student''s grade for the course is: ', ave);
end.
```

Although the above program gets the job done it is obviously very inefficient. The first of the program's inefficiencies is the necessity to declare each of the grade variables individually. As it was previously stated, an array uses only one identifier for all of its individual elements. Table 7.1 illustrates how an array is declared.

Table 7.1 *Declaring a Pascal array*

Var	
identifier : **array**[index type] **of** data type;	
Where:	
identifier	is a valid Pascal identifier.
index type	is an ordinal data type or ordinal subrange, except **LongInt** or **LongInt** subranges.
data type	is the array's data type.

As Table 7.1 shows, the number of elements in an array is defined by either an ordinal data type or an ordinal subrange. You will find that almost all array indexes are declared using subranges. The following are some examples of valid array declarations:

```
monthly_income : array[1..12] of real;
temperatures : array[-200..200] of integer;
```

If we wanted to declare an array for the grades in Listing 1, we could use something like the following:

```
g : array[1..10] of integer;
```

Certainly, declaring the needed variables as an array is preferable to declaring them as individual variables. However, there is still one remaining problem. How are the individual elements of an array accessed? Fortunately, the method for accessing a Pascal array element is really quite simple. Table 7.2 illustrates how an array element is accessed. As this table shows, the grades for the above example can be accessed as **g[1]**, **g[2]**, **g[3]**, **g[4]**, **g[5]**, **g[6]**, **g[7]**, **g[8]**, **g[9]**, and **g[10]**. Furthermore, operations can be performed on these individual array elements just like they would be for individually declared integer variables.

Table 7.2 *Accessing a Pascal array element*

identifier[index]	
Where:	
identifier	is a previously declared array identifier.
index	is a valid element number.

Listing 7.2 presents a modified version of the program that was presented in Listing 7.1. This version substitutes an integer array for the student's grades. Besides showing how much easier it is to declare an array than numerous individual variables, Listing 7.2 also demonstrates how the student's total grade can be figured much more efficiently with a **for** loop. To accomplish this calculation,

the program simply employs the loop counter **i** for the array element index. Thus, each of the array's individual elements are added together to form the total.

Listing 7.2

```
{ list7-2.pas - Figure student's average no. 2 }
program stud_avg_2;

var
      grades : array[1..10] of integer;
      i, total, ave : integer;

begin
      grades[1] := 90;
      grades[2] := 89;
      grades[3] := 100;
      grades[4] := 97;
      grades[5] := 85;
      grades[6] := 99;
      grades[7] := 96;
      grades[8] := 100;
      grades[9] := 94;
      grades[10] := 100;
      total := 0;
      for i := 1 to 10 do
            total := total + grades[i];
      ave := total div 10;
      writeln('The student''s grade for the course is:
            ', ave);
end.
```

Lesson 67: Typed Constant Arrays

Although the program presented in Listing 7.2 is a vast improvement over the program in Listing 7.1, the program could be even further simplified by declaring the student's grade array as a typed constant array. Table 7.3 illustrates how a typed constant array is declared. As this table displays, a typed constant array's initial values are declared by surrounding them with parenthesis and separating them with commas. The following are a few examples of valid type constant array declarations:

```
vowels : array[1..5] of char = ('a', 'e', 'i', 'o', 'u');
odd_numbers : array[1..5] of integer = (1, 3, 5, 7, 9);
```

Table 7.3 *Declaring a typed constant array*

Const	
identifier : **array**[index type] **of** data type = (initial values);	
Where:	
identifier	is a valid Pascal identifier.
index type	is an ordinal data type or an ordinal subrange, except **Longint** or **LongInt** subranges.
data type	is an array's data type.
initial values	is the array elements' initial values. Each of which is separated by a comma.

Listing 7.3 demonstrates how the program presented in Listing 7.2 could be further modified by declaring the student's grade array as a typed constant. Even a cursory examination of the program discloses that the use of a typed constant array eliminates almost all of the assignment statements found in Listing 7.2.

Listing 7.3

```
{ list7-3.pas - Figure student's average no. 3 }
program stud_avg_3;

const
      grades : array[1..10] of integer = (90, 89, 100,
97, 85, 99, 96,
                  100, 94, 100);

var
      i, total, ave : integer;

begin
      total := 0;
      for i := 1 to 10 do
            total := total + grades[i];
      ave := total div 10;
      writeln('The student''s grade for the course is:
', ave);
end.
```

Lesson 68: Multi-Dimensional Arrays

Although the short programs presented in this chapter's previous lessons have been useful for demonstrating how simple arrays are used in the Pascal programming language, they are not really true to life. All of these programs have demonstrated how a student's course average could be figure by totaling the stu-

dent's scores and then figuring the average score. The only problem with these programs is the simple fact that it is very unlikely that a class would ever have just one student. Consequently, a truly useful program would have to be written in such a way that it could figure the course averages for a number of students. Listing 7.4 presents a very simple variation of the previous programs.

Listing 7.4

```
{ list7-4.pas - Figure student's average no. 4 }
program stud_avg_4;
type
      grade_arr = array[1..10] of integer;

const
      student1 : grade_arr = (90, 89, 100, 97, 85, 99, 96,
                  100, 94, 100);
      student2 : grade_arr = (85, 75, 90, 88, 87, 93, 95,
                  97, 99, 100);

var
      i, total1, ave1, total2, ave2 : integer;

begin
      total1 := 0;
      total2 := 0;
      for i := 1 to 10 do
      begin
            total1 := total1 + student1[i];
            total2 := total2 + student2[i];
```

Listing 7.4

```
        end;
        ave1 := total1 div 10;
        ave2 := total2 div 10;
        writeln('Student no. 1''s grade for the course
is: ', ave1);
        writeln('Student no. 2''s grade for the course
is: ', ave2);
end.
```

Essentially, the above program declares a typed constant array for each of the course's students. Although the program is functionally correct, it is far from being the most efficient Pascal program we could write. A much more efficient method for representing the data in Listing 7.4 would be to define it as a multi-dimensional array. Table 7.4 illustrates how a multi-dimensional array is declared. Table 7.5 illustrates how a multi-dimensional type constant is declared. Although these illustrations both show how a two-dimensional array is declared, multi-dimensional Pascal arrays are by no means limited to only two dimensions. Indeed, three-dimensional arrays are quite common in a wide variety of programs.

Table 7.4 *Declaring a multi-dimensional array*

Var

identifier **: array[**index type, index type**] of** data type;

Where:

identifier	is a valid Pascal identifier.
index type	is an ordinal data type or an ordinal subrange, except **LongInt** or **LongInt** subranges. The array's dimensional index types are separated by commas.
data type	is the array's data type.

Table 7.5 *Declaring a multi-dimensional typed constant*

Const

identifier **: array**[index type, index type] **of data type =**
((initial values), (initial values));

Where:

identifier	is a valid Pascal identifier.
index type	is an ordinal data type or an ordinal subrange, except **LongInt** or **LongInt** subranges. The array's dimensional index types are separated by commas.
data type	is the array's data type.
initial values	is the array elements' initial values. Each of which is separated by a comma.

As declaring a multi-dimensional array is a simple variation of declaring a single-dimensional array, accessing a multi-dimensional array element is a slight variation of the method used to access a single-dimensional array element. Table 7.6 presents two methods for accessing a multi-dimensional array element. Although both methods are acceptable, the first method is what most Pascal programmers prefer.

Table 7.6 *Accessing a multi-dimensional array element*

identifier[index, index]

or

identifier[index][index]

Where:

identifier	is a previously declared array identifier.
index	is a valid element number.

Listing 7.5 presents a slightly modified version of the program presented in Listing 7.4. This new version substitutes a multi-dimensional typed constant array for the two individual student arrays.

Listing 7.5

```
{ list7-5.pas - Figure student's average no. 5 }
program stud_avg_5;
const
      students : array[1..2, 1..10] of integer =
            ( (90, 89, 100, 97, 85, 99, 96, 100, 94, 100),
              (85, 75, 90, 88, 87, 93, 95, 97, 99, 100) );

var
      i, total1, total2, ave1, ave2 : integer;

begin
      total1 := 0;
      total2 := 0;
      for i := 1 to 10 do
      begin
            total1 := total1 + students[1, i];
            total2 := total2 + students[2, i];
      end;
      ave1 := total1 div 10;
      ave2 := total2 div 10;
      writeln('Student no. 1''s grade for the course
      is: ', ave1);
      writeln('Student no. 2''s grade for the course
      is: ', ave2);
end.
```

Although the above program is a step in the right direction, the use of four variables to figure the two students' course average is very wasteful. Listing 7.6

presents an even simpler version of the program, which uses a nested **for** loop to calculate the course averages.

Listing 7.6

```
{ list7-6.pas - Figure student's average no. 6 }
program stud_avg_6;
const
      students : array[1..2, 1..10] of integer =
            ( (90, 89, 100, 97, 85, 99, 96, 100, 94, 100),
              (85, 75, 90, 88, 87, 93, 95, 97, 99, 100) );

var
      i, j, total, ave : integer;

begin
      for i := 1 to 2 do
      begin
            total := 0;
            for j := 1 to 10 do
            begin
                  total := total + students[i, j];
            end;
            ave := total div 10;
            writeln('Student no. ', i, '''s grade for
            the course is: ', ave);
      end;
end.
```

Lesson 69: Passing Arrays to Procedures and Functions

To pass an array to a procedure or a function, you must first define a data type for the array.

The one exception to this is for **string**s. **String**s are nothing more than a **char** array. Because they are defined as part of the Pascal programming language, they are already a predefined data type and a new definition would be unnecessary.

Table 7.7 displays how an array data type is defined. As this figure shows, an array data type is defined just like any other new data type and once it has been defined can be used to indicate an identifier's data type.

Table 7.7 *Defining an array data type*

type

identifier = **array**[index type] **of** data type;

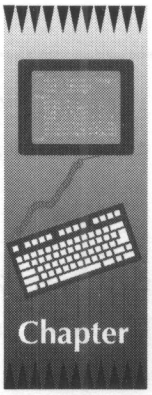
Where:

identifier	is the new data type's identifier.
index type	is an ordinal data type or an ordinal subrange, except **LongInt** or **LongInt** subranges.
data type	is the array's data type.

Listing 7.7 displays a brief program that demonstrates how an array is passed to a function in an actual Pascal program. Note that by using the **var** keyword in the **total** function's head, the array is passed by reference. Although arrays can be passed by value, passing an array by value requires Pascal to make a temporary copy of the array before each procedure call. Not only is this a time consuming process, it could easily lead to an out of memory situation. This is particularly true if the procedure or function is recursive. Consequently, it is almost always best to pass arrays by reference instead of by value.

Listing 7.7

```
{ list7-7.pas - Demonstrate how an array is passed to
a function }
program array_passing_demo;

type
      list_array = array[1..10] of integer;

function total(var la : list_array) : integer;
var
      i, t : integer;

begin
      t := 0;
      for i := 1 to 10 do
            t := t + la[i];
      total := t;
end;

const
      list : list_array = (3, 2, 4, 5, 6, 7, 8, 9, 1, 9);

begin
      writeln('Total for the array is ', total(list));
end.
```



8

Records

In the last chapter, you learned how related data of the same data type can be conveniently grouped together in an array. Although arrays are a very useful programming tool, a lot of programs work with data that are related and of different data types. To deal with related data of different types, the Pascal programming language offers a user-defined data type called *records*. Before we examine the nuts and bolts details of Pascal records, let's first take a look at a program presented in Listing 8.1. Essentially, this program builds upon the programs presented in Chapter Seven. Instead of just displaying the course averages for a couple of students though, this new version also displays the students' names. Obviously, the names and the averages are related data. Yet, they are represented by vastly different data types.

Listing 8.1

```
{ list8-1.pas - Nonrecord demonstration }
program nonrecord;

var
      name1, name2 : string;
      ave1, ave2 : integer;

begin
      name1 := 'John Smith';
      ave1 := 95;
      name2 := 'Jane Doe';
      ave2 := 98;
      writeln(name1, ''s average is a ', ave1);
      writeln(name2, ''s average is a ', ave2);
end.
```

Lesson 70: Records Basics

The first step in using a record in a Pascal program is to define the record's data type. Table 8.1 illustrates how a Pascal record is declared. As this figure shows, the record declaration is constructed from a number of field declarations. Table 8.2 illustrates how a field declaration is defined. This illustration shows that field declarations are very similar to a normal old variable definition. The following are some examples of valid Pascal record declarations:

```
mail_item = record
   name1, name2, address : string[30];
   city : string[15];
   state : string[2];
```

```
    zip1 : string[5];
    zip2 : string[4];
end;

student = record
    name : string;
    grades : array[1..10] of integer;
end;
```

Table 8.1 *Declaring a Pascal record type*

type
data type identifier = **record**
field declaration;
.
.
field declaration;
end;

Where:

data type identifier	is the new data type's identifier.
field declaration	is a valid field declaration.

Table 8.2 *Declaring a Pascal record field*

field identifier : data type;

Where:

field identifier	is the record field's identifier.
data type	is the record field's data type.

Table 8.3 displays how a record variable's field is referred to in an assignment statement or an expression. As this illustration shows, the field is referred to by

simply separating the variable's name and the field's name with a period (.). The following are a few examples of valid Pascal record variable field references:

```
item.name1 := 'John Smith';
item.address := '375 Sleepy Lane';
s1.name := 'Jane Doe';
total := s1.grades[1] + s1.grades[2];
```

Table 8.3 *Record variable field references*

variable identifier.field identifier	
Where:	
variable identifier	is the record variable's name.
field identifier	is the field name.

Listing 8.2 displays a brief program that demonstrates how the program presented in Listing 8.1 could be rewritten to take advantage of the Pascal programming language's support for record data types. Although it is only a very simple example of how Pascal record types are used in an actual program, it serves the purpose of clearly showing how related data can be joined together as a single data entry. Thus, eliminating the necessity of having to use separate variables for each of the record's fields.

Listing 8.2

```
{ list8-2.pas - Record demonstration }
program record_demo;

type
      student = record
            name : string;
            ave : integer;
      end;
```

Listing 8.2—Continued

```
var
      s1, s2 : student;

begin
      s1.name := 'John Smith';
      s1.ave := 95;
      s2.name := 'Jane Doe';
      s2.ave := 98;
      writeln(s1.name, '''s average is a ', s1.ave);
      writeln(s2.name, '''s average is a ', s2.ave);
end.
```

Lesson 71: The With Statement

Although Pascal records are an extremely valuable programming tool, records with long variable names are hard to work with because of the necessity of having to type out the variable name before each of the field names. As an example, let's suppose we were to write a mail list program that uses a record type similar to the following:

```
list_item = record
   name, address : string[30];
   city : string[15];
   state : string[2];
   zip : string[5];
end;
```

Now let's further suppose that the program uses the following statements to assign values to a **list_item** record variable named **mail_list_item1**:

```
mail_list_item1.name := 'John Smith';
mail_list_item1.address := '325 Cherry Tree Lane';
mail_list_item1.city := 'Washington';
mail_list_item1.state := 'DC';
mail_list_item1.zip := '00001';
```

Obviously, having to type **mail_list_item1** over and over and over is a very tedious task. Wouldn't it be nice if there was some type of shorthand method for writing the above assignment statements? As usual, Pascal comes to the rescue with the **with** statement. Table 8.4 illustrates how the **with** statement is used to eliminate the necessity of having to retype the variable's name over and over. The following statements show how the **with** statement can be use to rewrite the above **mail_list_item1** statements:

```
with mail_list_item1 do
begin
   name := 'John Smith';
   address := '325 Cherry Tree Lane';
   city := 'Washington';
   state := 'DC';
   zip := '00001';
end;
```

Now aren't the above statements a whole lot simpler than the previous ones?

Table 8.4 *The Pascal* ***with*** *statement*

with variable identifier **do**	
program statement;	
Where:	
variable identifier	is a previously defined record variable's identifier.
program statement	is a valid Pascal statement.

Listing 8.3 demonstrates how the program presented in Listing 8.2 could be rewritten to take advantage of the Pascal **with** statement.

Listing 8.3

```
{ list8-3.pas - With statement demonstration }
program with_demo;

type
      student = record
            name : string;
            ave : integer;
      end;

var
      s1, s2 : student;

begin
      with s1 do
      begin
            name := 'John Smith';
            ave := 95;
            writeln(name, '''s average is a ', ave);
      end;
      with s2 do
      begin
            name := 'Jane Doe';
            ave := 98;
            writeln(name, '''s average is a ', ave);
      end;
end.
```

Lesson 72: Typed Constant Records

As with arrays, Pascal records can be declared as *typed constants* to furnish an easy way for providing records with initial values. Table 8.5 displays how a typed constant record is declared. As this figure shows, a typed constant is initialized by specifying initial values for the record variable's fields. The following are some examples of valid typed constant record definitions:

```
list1 : mail = ( name : 'John Smith'; address : '338
Main St.
   Suite C'; city : 'Somewhere'; state : 'US'; zip :
'00000' );


s1 : student = ( name : 'Jane Doe'; grade : 99 );
```

Table 8.5 *Declaring a typed constant record*

Const

identifier : data type identifier =

(field : initial value; field : initial value);

Where:

identifier	is the variable's name.
data type identifier	is the variable's record type.
field	is one of the record type's field name.
initial value	is an initial value of the record's field.

The program presented in Listing 8.4 demonstrates how the program that was previously presented in Listing 8.2 could be further modified to utilize typed constant records. As you can see, the use of typed constant records in this newer version greatly simplifies how the program is written.

Listing 8.4

```
{ list8-4.pas - Typed constant record demonstration }
program type_const_record_demo;

type
      student = record
            name : string;
            ave : integer;
      end;

const
      s1 : student = (name : 'John Smith'; ave : 95);
      s2 : student = (name : 'Jane Doe'; ave : 98);

begin
      Writeln(s1.name, '''s average is a ', s1.ave);
      Writeln(s2.name, '''s average is a ', s2.ave);
end.
```

Lesson 73: Record Arrays

Wouldn't it be nice if the Pascal program could use *record arrays* and arrays for fields in a program? Fortunately, Pascal fully supports both types of arrays. Table 8.6 illustrates how an array of records is declared. Obviously, there is really no difference between this type of declaration and any other array declaration. The only sticky part with using record arrays in a program is referencing a particular record field. Table 8.7 shows how a field is referenced in an array of records. Note how the array index comes before the period (.) and not after the field name as you might expect.

Table 8.6 *Declaring an array of records*

Var	
identifier **: array**[index type] of **record type;**	
Where:	
identifier	is a valid Pascal identifier.
index type	is an ordinal data type or ordinal subramge, except **LongInt** or **LongInt** subranges.
record type	is a previously defined record type.

Table 8.7 *Record array field references*

variable identifier[index].field identifier	
Where:	
variable identifier	is the record variable's name.
index	is the record array's element number.
field identifier	is the field name.

Listing 8.5 displays a variation of this chapter's previous programs that utilize a record array to store the student data.

Listing 8.5

```
{ list8-5.pas - Demonstrate record arrays }
program record_arrays;

type
      student = record
            name : string;
            ave : integer;
      end;

const
      class : array[1..2] of student =
            ( (name : 'John Smith'; ave : 95),
              (name : 'Jane Doe'; ave : 98) );

begin
      writeln(class[1].name, '''s average is a ',
class[1].ave);
      writeln(class[2].name, '''s average is a ',
class[2].ave);
end.
```

Lesson 74: Field Arrays

As the previous lesson mentioned, Pascal supports *field arrays* as well as record arrays. To declare a field as an array, you simply declare the field like any other array declaration. Table 8.8 illustrates just how a field array is declared. The proper method for referencing a field element is illustrated in Table 8.9. As you can see from this illustration, the field element's array index is specified right after the field's name.

Table 8.8 *Declaring a field array*

field identifier : **array**[index type] **of** data type;	
Where:	
field identifier	is field's name.
index type	is an ordinal data type or ordinal subrange, except **LongInt** or **LongInt** subranges.
data type	is the array's data type.

Table 8.9 *Field array references*

variable identifier.field identifier[index]	
Where:	
variable identifier	is the record variable's name.
field identifier	is the record field name.
index	is the field array's element number.

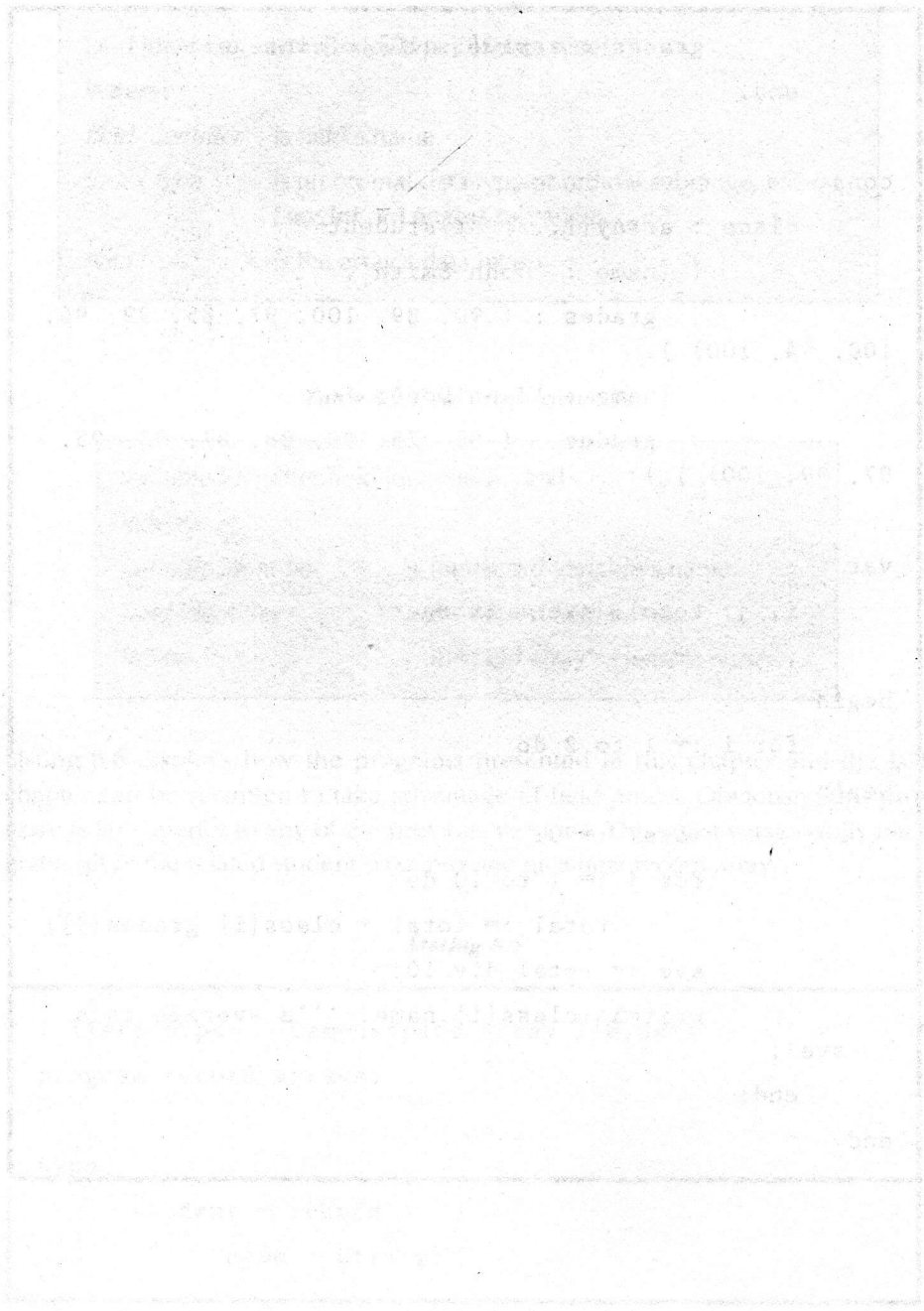
Listing 8.6 displays how the programs presented in this chapter and the last chapter can be rewritten to take advantage of field arrays. Obviously, this program is far superior to any of the previous versions. This latest version fully integrates all of the related student data into one nice neat record array.

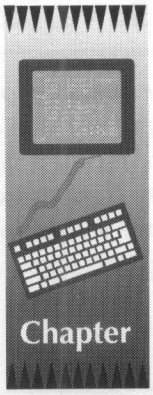
Listing 8.6

```
{ list8-6.pas - Demonstrate array fields }
program record_arrays;

type
     student = record
          name : string;
```

Listing 8.6—Continued

```
            grades : array[1..10] of integer;
      end;

const
      class : array[1..2] of student =
            ( (name : 'John Smith';
               grades : ( 90, 89, 100, 97, 85, 99, 96,
100, 94, 100) ),
              (name : 'Jane Doe';
               grades : ( 85, 75, 90, 88, 87, 93, 95,
97, 99, 100) ) );

var
      i, j, total, ave : integer;

begin
      for i := 1 to 2 do
begin
            total := 0;
            for j := 1 to 10 do
                  total := total + class[i].grades[j];
            ave := total div 10;
            writeln(class[i].name, '''s average is a
', ave);
      end;
end.
```





9

Variant Records

In the last chapter, you learned what a useful programming tool a Pascal record type can be. This chapter continues with our study of Pascal records by presenting the *variant record.* As its name implies, the variant record is able to vary its fields depending on the type of data to be stored. Although this may sound rather strange, you will soon see what a useful and powerful programming tool the variant record can be.

Lesson 75: Variant Records

To better understand the power of Pascal variant records, let's first take a look at the program presented in Listing 9.1. Essentially, this program displays a short faculty/student list for a rather small college. The chief thing to note about this program is how it needs two record types: one for a faculty member and one for a member of the student body. Obviously, a record for a faculty member needs vastly different data than a record for a student.

Listing 9.1

```
{ list9-1.pas - Display faculty/student list without
variant records}
program disp_list1;

type
      class_type = ( freshman, sophomore, junior,
      senior );
      faculty = record
            name : string;
            age : integer;
            salary : real;
            years : integer;
      end;
      student = record
            name : string;
            age : integer;
            average : real;
            class : class_type;
      end;
```

Listing 9.1—Continued

```
const
      f_list : array[1..2] of faculty =
            ( (name : 'John Smith'; age : 53; salary :
33500; years : 15),
              (name : 'Jane Doe'; age : 45; salary :
33500; years : 14) );
      s_list : array[1..2] of student =
            ( (name : 'Calvin Doe'; age : 20; average
: 3.76; class : junior),
              (name : 'Sue Smith'; age : 22; average :
4.0; class : senior) );

var
      i : integer;

begin
      for i := 1 to 2 do
            with f_list[i] do
                  writeln(name, ' ', age, ' ',
salary:8:2, ' ', years);
      for i := 1 to 2 do
            with s_list[i] do
            begin
                  write(name, ' ', age, ' ', aver-
age:4:2, ' ');
                  case class of
                        freshman :
writeln('Freshman');
```

Listing 9.1—Continued

```
sophomore : writeln('Sophomore');
                    junior : writeln('Junior');
                    senior : writeln('Senior');
                end;
            end;
end.
```

Although the above program certainly gets the job done, wouldn't it be nice if the *faculty* and *student* records could be combined into one record type? Of course, we could always use something like the following:

```
fs_rec = record
   name : string;
   age : integer;
   salary : real;
   years : integer;
   average : real;
   class : class_type;
end;
```

Whereas the above record type would work, it would obviously waste an enormous amount of space. After all, you wouldn't need to store a grade point average or class for a faculty member or a salary or number of years employed for a student. This type of data handling requirement is ideally suited for a variant record. Not only can the two record types be combined, but a variant record requires only as much memory as the largest individual record that it is constructed from. Table 9.1 illustrates how a variant record is declared.

Table 9.1 *Declaring a variant record*

```
record identifier = record
                    field declaration;
                    .
                    .
                    field declaration;
                    case tag identifier : data type of
                     case label : ( field declaration;);
                     .
                     .
                     case label : (field declaration;);
                    end;
```

Where:

record identifier	is the data type's identifier.
field declaration	is a valid field declaration. Multiple fields in the variant **case..end** structure must be separated by a semicolon.
tag identifier	is an optional tag identifier. If used, the "case" can be accessed through the tag identifier just like it was any other field.
data type	is the tag identifier's data type.
case label	is a label that identifies which field the variant records use for a particular tag.

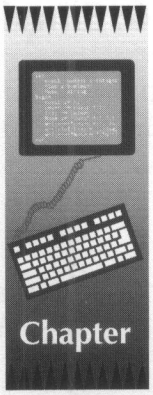
Listing 9.2 displays how the program presented in Listing 9.1 could be modified to take advantage of Pascal's variant records. The chief thing to note about this program is it's use of the tag identifier **fs** in the **fs_rec** variant record type. By setting this record field to indicate the type of data stored in the variant record, it is an extremely easy task to extract and properly display each of the variant record's appropriate values. Without **fs**, it would be impossible to know what type of data is stored in the variant record and how it is properly handled.

Listing 9.2

```
{ list9-2.pas - Display faculty/student list with
variant records}
program disp_list1;

type
      fs_type = ( faculty, student );
      class_type = ( freshman, sophomore, junior,
      senior );
      fs_rec = record
            name : string;
            age : integer;
            case fs : fs_type of
                  faculty : ( salary : real; years :
                  integer; );
                  student : ( average : real; class :
                  class_type);
      end;

   const
      list : array[1..4] of fs_rec =
            ( (name : 'John Smith'; age : 53; fs :
   faculty; salary : 33500;
                  years : 15),
                (name : 'Jane Doe'; age : 45; fs :
   faculty; salary : 33500;
                  years : 14),
 (name : 'Calvin Doe'; age : 20; fs : student; average
: 3.76;
                  class : junior),
```

Listing 9.2—Continued

```
                (name : 'Sue Smith'; age : 22; fs : stu-
dent; average : 4.0;
                 class : senior) );

var
      i : integer;

begin
      for i := 1 to 4 do
            with list[i] do
                  if fs = faculty then
                        writeln(name, ' ', age, ' ',
salary:8:2, ' ', years)
                  else
                  begin
                        write(name, ' ', age, ' ',
average:4:2, ' ');
                        case class of
                              freshman :
writeln('Freshman');
                              sophomore :
writeln('Sophomore');
                              junior :
writeln('Junior');
                              senior :
writeln('Senior');
                        end;
                  end;
end.
```



10

Pointers

In this chapter, we examine one of the Pascal programming language's most important features called *pointers*. Essentially, pointers *point* to a data value. For example, an **integer** pointer would point to an **integer** value, a **real** pointer would point to a **real** value, etc. Besides pointing to data values, a pointer can also point to a procedure or a function. Although pointers may not sound all that special, they are indeed one of Pascal's most significant features. Consequently, all Pascal programmers should be well acquainted with how pointers are declared and how they are used in actual Pascal programs.

Lesson 76: Simple Pointers

As with all other types of data, a Pascal pointer must be declared before it is used in a program. Table 10.1 displays how a Pascal pointer is declared. The following are some examples of valid pointer declarations:

```
int1ptr, int2ptr : ^integer;
nameptr : ^string;
AmountPtr : ^Real;
```

Table 10.1 *Declaring a Pascal pointer*

Var	
identifier :^data type;	
Where:	
identifier	is the pointer's identifier.
data type	is the data type the pointer is pointing to.

You should note that a pointer declaration is much like a normal variable declaration in that there is no initial value assigned to the pointer. Therefore, results are unpredictable if a pointer is used without first initializing it. A pointer can be initialized in one of three ways: assign to the pointer an address of the proper data type, assign to the pointer the value of another pointer of the same data type, or assign to the pointer the special value **nil**.

Assigning a pointer the value of an address of a data value is displayed in Table 10.2. As this table shows, either the Pascal **@** operator or **addr** function can be used to assign the address of the data value to the pointer. It really doesn't make a difference which method is used because they both perform the same task.

Table 10.2 *Assigning a variable's address to a pointer*

pointer identifier := @ variable identifier;
or
pointer identifier := **addr**(variable identifier);
Where:
pointer identifier is the pointer's identifier.
variable identifier is the identifier of the variable, whose address is assigned to the pointer. **Note:** Both variable identifier and pointer identifier must be of the same data type.

Assigning the value of one pointer to another is displayed in Table 10.3. As this table shows, assigning one pointer to another is just like assigning the value of one variable to another.

Table 10.3 *Assigning one pointer to another*

pointer identifier := pointer identifier;
Where:
pointer identifier is a previously declared pointer identifier.

Assigning the value **nil** to a pointer is displayed in Table 10.4. Essentially, **nil** is used to indicate a pointer that has no special meaning. Therefore, it's almost always a good idea to set a pointer that doesn't have a meaning to **nil**. That way, the program can check for a **nil** pointer before it carries out a meaningless operation on an unassigned pointer.

Table 10.4 *Assigning* ***nil*** *to a pointer*

pointer identifier := **nil;**
Where:
pointer identifier is a previously defined pointer identifier.

Although all of this material about pointers is interesting, you're probably wondering just how the data value that a pointer is pointing to is actually accessed. Well, it's so simple it's almost ridiculous. You simply add a ^ to the end of the pointer's identifier. This tells Pascal to treat the pointer identifier as a variable identifier for the data value that the pointer is pointing to. The following are a few examples of pointers being used to retrieve and store data values:

```
int_ptr^ := 35 div count;
int_result := int_ptr^ + 5;
```

Listing 10.1 displays a brief program that demonstrates how Pascal pointers are declared and used in an actual program.

Listing 10.1

```
{ list10-1.pas - Simple pointers demo }
program simple_pointers;

var
      intptr : ^integer;
      i1, i2 : integer;

begin
      i1 := 1;
      i2 := 2;
      intptr := @i1;
      writeln('intptr^ is ', intptr^, ' and so isn''t ', i1);
      intptr^ := i2;
      writeln('Now intptr has changed i1 to ', i1);
end.
```

Lesson 77: Array and Record Pointers

Besides supporting pointers to simple variables, Pascal also supports *array and record pointers*. Table 10.5 displays how an array element is referenced by an array pointer and Table 10.6 illustrates how a record field is referenced by a record pointer. As with a simple variable reference, a pointer reference to either an array element or a record field is performed by putting a ^ after the pointer's identifier.

Table 10.5 *Array pointer referencing*

pointer identifier^[index]	
Where:	
pointer identifier	is the array pointer's identifier.
index	is a valid element number.

Table 10.6 *Record pointer referencing*

pointer identifier^.field identifier	
Where:	
pointer identifier	is the record pointer's name.
field identifier	is the field name.

Listing 10.2 displays a brief program that shows how an array pointer is used in an actual Pascal program. Essentially, this program initializes an array by using an array pointer to store the elements' values. Once stored, the appropriate values are displayed by accessing them with normal array element referencing methods.

Listing 10.2

```
{ list10-2.pas - Array pointers }
program array_pointers;

type
      intarray = array[1..10] of integer;

var
      iaptr : ^intarray;
      ia : intarray;
      i : integer;

begin
      iaptr := @ia;
      for i := 1 to 10 do
            iaptr^[i] := i;
      for i := 1 to 10 do
            writeln('ia[', i, '] = ', ia[i]);
end.
```

Listing 10.3 displays a brief listing that demonstrates how a record pointer is used in a Pascal program. Similar to the above array pointer demonstration program, this program initializes a record by using a record pointer to assign the initial values to the record's fields. Once the appropriate values are assigned, they are displayed by using the normal Pascal record field referencing methods.

Listing 10.3

```
{ list10-3.pas - Record pointers }
program record_pointers;

type
      maillist = record
            name, address, city, state, zip : string;
      end;
var
      mlptr : ^maillist;
      item : maillist;

begin
      mlptr := addr(item);
      mlptr^.name := 'John Smith';
      mlptr^.address := 'West 57th St.';
      mlptr^.city := 'Somewhere';
      mlptr^.state := 'US';
      mlptr^.zip := '00001';
      writeln('Name:    ', item.name);
      writeln('Address: ', item.address);
      writeln('City   : ', item.city);
      writeln('State  : ', item.state);
      writeln('Zip    : ', item.zip);
end.
```

Lesson 78: Procedure and Function Pointers

Now that we have explored how pointers can be used to manipulate simple variables, array variables, and records, it's time to take a look at how they can be used with procedures and functions. Before we continue with our pointer discussion, you must understand procedure and function variables. Essentially, a procedure or a function variable is a variable that can hold the address of a procedure or function. With the address of a procedure or function safely tucked away in a procedure or function variable, they can be called appropriately by simply using the procedure or function variable's identifier in place of their name.

Although procedure and function variables don't really sound all that special at first glance, they are an extremely powerful programming tool. Let's suppose you were writing a program that required different routines to be called depending upon a variety of conditions. You could always write a convoluted decision making statement to handle the different circumstances, but it's much easier to just modify a procedure or function variable when a certain condition occurs. That way, a single procedure or function call could handle all of the circumstances that can arise in a program.

To be able to declare a procedure or function variable in a program, a data type for the variable must first be defined. Table 10.7 displays how a procedure data type is defined and Table 10.8 illustrates how a function data type is defined. Basically, the data type is defined with a procedure or function head that doesn't have an identifier. Once an appropriate data type has been defined, a procedure or function variable is declared just like any other variable. They can be declared as simple variables, array variables, record fields, etc.

Table 10.7 *Defining a procedure data type*

type

data type identifier = **procedure**(parameter list);

Where:

data type identifier	is the data type's name.
parameter list	is a list of arguments to be passed to the procedure.

Table 10.8 *Defining a function data type*

type

data type identifier = **function**(parameter list) **:** return type;

Where:

data type identifier	is the data type's name.
parameter list	is a list of arguments to be passed to the function.
return type	is the return value's data type.

Before a procedure or function variable can be used to call a procedure or a function, an initial value must be assigned to it. Table 10.9 displays how a procedure's or function's address is assigned to a variable. As this figure shows, an **@** operator must be used before the procedure or function variable's name. Without the **@** operator, Pascal would try to execute the procedure or function contained in the procedure or function variable. Obviously, an attempt to call the routine in an unassigned procedure or function variable is unacceptable.

Table 10.9 *Procedure and function variable assignments*

@ variable identifier **:=** @ procedure identifier;

or

@ variable identifier **:=** @ function identifier;

Where:

variable identifier	is the procedure or function variable's name.
procedure identifier	is the procedure's name.
function identifier	is the function's name.

Listing 10.4 displays a program that demonstrates how procedure variables could be used in an actual Pascal program to build a rather simple menu system. As you can see from this program, procedure and function variables can be quite effective for building a menu system. If a menu item needs to be changed due

to a condition in the program, the new routine's address can simply be assigned to the appropriate procedure variable. Thus, permitting the program to quickly and efficiently modify itself due to a change in current conditions.

Listing 10.4

```
{ list10-4.pas - Procedure and function variables demo }
program proc_and_func_vars;

uses crt;

procedure m1;
begin
     writeln('This is menu item 1');
end;

procedure m2;
begin
     writeln('This is menu item 2');
end;

procedure m3;
begin
     writeln('This is menu item 3');
end;

procedure m4;
begin
     writeln('This is menu item 4');
end;
procedure m5;
begin
```

Listing 10.4—Continued

```
        halt;
end;

type
        proc = procedure;

var
        menuprocs : array[1..5] of proc;
        key : integer;

begin
        @menuprocs[1] := @m1;
        @menuprocs[2] := @m2;
        @menuprocs[3] := @m3;
        @menuprocs[4] := @m4;
        @menuprocs[5] := @m5;
        while true do
        begin
                writeln('[1]...Menu Item # 1');
                writeln('[2]...Menu Item # 2');
                writeln('[3]...Menu Item # 3');
                writeln('[4]...Menu Item # 4');
                writeln('[5]...Exit The Program');
                repeat
                        key := integer(readkey);
                until (key > 48) and (key < 54);
                menuprocs[key - 48];
        end;
end.
```

Now that you know how a menu system can be built using procedure variables, let's see how the same system can be built using pointers. To assign a procedure's or a function's address to a pointer, it is first necessary to define a pointer with a data type of **pointer**. Essentially, a **pointer** pointer has no real data type and can hold a pointer for anything. Table 10.10 displays how a **pointer** pointer is declared. As this table shows, a **pointer** pointer is declared just like any other pointer.

Table 10.10 *Declaring a* ***pointer*** *pointer*

Var
pointer identifier **: pointer;**
Where:
pointer identifier is the pointer's name.

Table 10.11 displays how a procedure's or a function's address is assigned to a **pointer** pointer. Unlike procedure and function variable assignments, a **pointer** pointer assignment doesn't require the **@** operator before the pointer's identifier. The reason for this is because the address is being assigned to a generic pointer; therefore, there is no way Pascal could possibly confuse a **pointer** pointer with a function call.

Table 10.11 *Assigning a procedure or function address to a pointer*

pointer identifier **:=@** procedure identifier;	
or	
pointer identifier **:=@** function identifier;	
Where:	
pointer identifier	is the procedure or function pointer's name.
procedure identifier	is the procedure's name.
function identifier	is the function's name.

Listing 10.5 displays a modified version of the program that was presented in Listing 10.4. Instead of using procedure variables to implement the menu system, this newer version uses pointers to accomplish the same task. The chief thing to note about this program is how typecasting is used to perform the actual procedure calls. Without casting the pointer to a procedure call, Pascal would assume that an assignment statement was being constructed and would generate an unintended error.

Listing 10.5

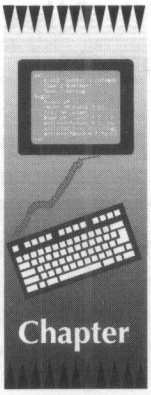
```
{ list10-5.pas - Procedure and function pointers demo }
program proc_and_func_ptrs;

uses crt;

procedure m1;
begin
      writeln('This is menu item 1');
end;

procedure m2;
begin
      writeln('This is menu item 2');
end;

procedure m3;
begin
      writeln('This is menu item 3');
end;
procedure m4;
begin
```

Listing 10.5—Continued

```
      writeln('This is menu item 4');
end;

procedure m5;
begin
      halt;
end;

type
      proc = procedure;

var
      menuprocs : array[1..5] of pointer;
      key : integer;

begin
      menuprocs[1] := @m1;
      menuprocs[2] := @m2;
      menuprocs[3] := @m3;
      menuprocs[4] := @m4;
      menuprocs[5] := @m5;
      while true do
      begin
            writeln('[1]...Menu Item # 1');
            writeln('[2]...Menu Item # 2');
            writeln('[3]...Menu Item # 3');
            writeln('[4]...Menu Item # 4');
```

Listing 10.5—Continued

```
            writeln('[5]...Exit The Program');
            repeat
                  key := integer(readkey);
            until (key > 48) and (key < 54);
            proc(menuprocs[key - 48]);
      end;
end.
```



11

Dynamic Memory Management

Now that we know everything there is to know about Pascal pointers, let's examine one of their most important uses. Throughout all of the previous chapters, we have been limited to using variables that are already declared in the program. When the program is compiled, Pascal automatically sets aside a place in the computer's memory to store the variables contents. Although all of this is well and good, often the Pascal programmer can't possibly know what a program's data requirements are until the program is actually run. To allow the Pascal programmer to write programs that can expand or contract their data space, Pascal provides a number of useful dynamic memory management routines. These routines can be used to allocate and deallocate memory for either a single data object or an entire block of memory.

Lesson 79: Allocating and Deallocating Single Data Objects

The obvious first place to begin our study of dynamic memory management is to examine how single data objects are *allocated* and *deallocated.* To allocate a data object, the Pascal programmer uses the **new** procedure. Table 11.1 illustrates how the **new** procedure is used. As this table illustrates, a pointer is passed as the **new** procedure's only argument. When the **new** procedure returns to the calling program, the pointer points to a memory location of the same size as the pointer's data type.

Table 11.1 *Allocating memory with the* ***new*** *procedure*

new(pointer); **Where:** *pointer* is a pointer to the allocated memory area.

To deallocate a previously allocated data object, the Pascal programmer uses the **dispose** procedure. Table 11.2 illustrates how the **dispose** procedure is used. Like the **new** procedure, the **dispose** procedure requires a single pointer argument. Essentially, the **dispose** procedure releases the pointer's previously allocated memory area. Once released, the deallocated memory area is available to be used by other dynamic memory management calls.

Table 11.2 *Deallocating memory with the* ***dispose*** *procedure*

dispose(pointer); **Where:** *pointer* is a pointer to a previously allocated memory area.

Listing 11.1 displays a short program, which demonstrates how the **new** and **dispose** procedures are used to dynamically allocate and deallocate memory. Essentially, the program allocates space for an integer data object, assigns a

value to the data object, displays the data object's value, and releases the data object's allocated memory area. As you can see from this program, allocating and deallocating dynamic memory is really quite simple.

Listing 11.1

```
{ list11-1.pas - Demonstrate allocating/deallocating a
single object }
program alloc_dealloc;

var
      int_ptr : ^integer;

begin
      new(int_ptr);
      int_ptr^ := 4;
      writeln('int_ptr^ = ', int_ptr^);
      dispose(int_ptr);
end.
```

Lesson 80: Allocating and Deallocating Blocks of Memory

Being able to allocate and deallocate a single data object is a very useful programming tool. It isn't very helpful if the Pascal programmer wants to dynamically allocate and deallocate memory for an array that varies in size during the life of the program. To fill this need, the Pascal programming language provides the **getmem** and **freemem** procedures. As its name implies, the **getmem** procedure gets (or allocates) a block of dynamic memory. Table 11.3 displays how the **getmem** procedure is used in a Pascal program. Like the **new** procedure, the **getmem** procedure requires a pointer argument to return a pointer to the

allocated memory. The **getmem** procedure also requires that the Pascal program specify the number of bytes to be allocated. The most convenient way to specify the number of allocation bytes is to use Pascal's **sizeof** function. Table 11.4 displays how the **sizeof** function is used. Essentially, the **sizeof** function returns the size in bytes for any previously defined data type. Consequently, the number of bytes required for a 100 element integer array could be specified with the following expression:

```
100 * sizeof(integer)
```

Table 11.3 *Allocating a memory block with* ***getmem***

getmem(pointer, size); **Where:** *pointer* is a pointer to the allocated memory block. *size* is the size of the memory block in bytes.

Table 11.4 *The Pascal* ***sizeof*** *function*

sizeof(data type); **Where:** *data type* is a previously defined data type.

As a compliment to the **getmem** procedure, the **freemem** procedure deallocates a previously allocated memory block. Table 11.5 displays how the **freemem** procedure is used in a Pascal program. As with the **getmem** procedure, the **freemem** procedure requires a pointer to the memory block and the block's size in bytes.

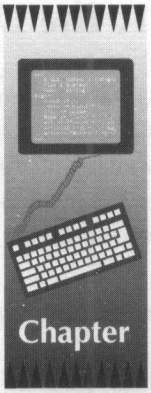
Table 11.5 *Deallocating memory with the* ***freemem*** *procedure*

freemem(pointer, size); **Where:** *pointer* is a pointer to a previously allocated memory block. *size* is a memory block's size in bytes.

Listing 11.2 displays a program that demonstrates how the Pascal **getmem** and **freemem** procedures are used to dynamically allocate and deallocate memory. One thing to note about this program is the use of the constant **max_ints** in the **int_array** data type definition. Essentially, **max_ints** represents the maximum number of elements in an **integer** array. By defining an array data type with **max_ints** as the largest element, an **int_array** pointer can be used to point to an **integer** array of any size. This method can be easily adapted for any Pascal data type by substituting the desired data type in the constant declaration's **sizeof** function.

Listing 11.2

```
{ list11-2.pas - Demonstrate allocating/deallocating
blocks }
program alloc_dealloc;

const
      max_ints = 65520 div sizeof(integer);

type
      int_array = array[1..max_ints] of integer;

var
      i : integer;
      int_ptr : ^int_array;

begin
      getmem(int_ptr, 10 * sizeof(integer));
      for i := 1 to 10 do
            int_ptr^[i] := i;
      for i := 1 to 10 do
            writeln('int_ptr^[', i, '] = ',
int_ptr^[i]);
      freemem(int_ptr, 10 * sizeof(integer));
end.
```



12

Units

As you write more and more Pascal programs, you will find that many of the procedures and functions you create for a program are the same as ones you've used in other programs. Instead of re-inventing the wheel each time you write a program, you can collect all of your most often used routines together into a Pascal library called a *unit*. Throughout this book the example programs have used a number of procedures and functions that come with Turbo Pascal. The majority of these routines are contained in Pascal's default unit called *system*. Other units included with Turbo Pascal are *crt*, *dos*, *printer*, *graph*, *graph3*, *overlay*, and *turbo3*. Besides teaching you how to use these already supplied units with your own programs, this chapter shows how you can develop your own units to simplify your future programming tasks.

Lesson 81: The Uses Statement

To use a unit in a program, the unit's name must be specified in a **uses** statement. Table 12.1 displays how a **uses** statement is constructed. As this table illustrates, the name of the unit (or units) is declared after the **uses** keyword. Once its name has been specified in a **uses** statement, all of the unit's procedures and functions are at the Pascal programmer's disposal.

Table 12.1 *The Pascal* ***uses*** *statement*

uses unit identifier;	
Where:	
unit identifier	is the Pascal unit's name. More than one unit may be specified by separating their names with commas.

Listing 12.1 displays a simple program that demonstrates how the **crt** unit is used in an actual Pascal program. Although it only clears the screen and centers a message on the top display line, it serves the purpose of demonstrating how the Pascal **uses** statement is utilized.

Listing 12.1

```
{ list12-1.pas - Demonstrate the Pascal uses statement }
program uses_demo;

uses crt;

begin
      clrscr;
      gotoxy(28, 1);
      writeln('This Message Is Centered!');
end.
```

Lesson 82: Creating a Pascal Unit

Now that we know how a unit is used with a Pascal program, let's turn our attention to the nuts and bolts details of actually writing a unit. Figure 12.2 illustrates the structure of a Pascal unit. As a program starts with a **program** statement, a unit starts with a **unit** statement. The chief purpose of the **unit** statement is to simply assign a name to the unit. Figure 12.2 also illustrates how a **unit** statement is constructed.

The second part of a Pascal unit is the **interface** section. The **interface** section is used to declare any variables, constants, data types, etc. that you want the main Pascal program to be able to use. Additionally, the **interface** section includes procedure and function prototypes for any of the unit's procedures and functions that can be called by the main Pascal program.

The third portion of a Pascal unit is the **implementation** section. The **implementation** section is used to declare any variables, constants, data types, etc. that won't be accessible to the main Pascal program. The **implementation** section also includes definitions for both private procedures and functions and any procedures and functions that were made public by specifying their prototypes in the **interface** section.

The final section of a Pascal unit is the initialization code. This code is contained in a **begin..end** block just like the program's main body. Additionally, the initialization code block uses a period (.) to signify the end of the unit instead of a semicolon (;). Essentially, any program statements contained in the initialization code will be executed before the main Pascal program is executed. For example, a serial communications unit might have an initialization section that quite literally initializes the serial interface.

Table 12.2 *The structure of a Pascal unit*

unit identifier;
interface
public variables, constants, procedure prototypes, function prototypes, etc.
implementation
private variables, constants, procedures, functions, etc.
public procedures and functions
begin
initialization code
end.
Where:
identifier is the unit's name.

Listing 12.2, **uplow.pas**, presents a short unit, which clearly demonstrates how a Pascal unit is actually constructed. The chief things to note from this listing is how the two public procedures' (**uppercase** and **lowercase**) prototypes are defined in the unit's **interface** section and how an empty **begin..end** block is specified for the unit's initialization code. If you examine the unit's code, you will quickly deduce that the unit's **uppercase** procedure converts strings to all uppercase and the unit's **lowercase** procedure converts strings to all lowercase.

Listing 12.2

```
{ uplow.pas - Demonstrate how a Pascal unit is written }
unit uplow;

interface

procedure uppercase(var s : string);
procedure lowercase(var s : string);

implementation

const
      offset = integer('a') - integer('A');

function testupper(c : char) : boolean;
begin
      if (c >= 'A') and (c <= 'Z') then
            testupper := true
      else
            testupper := false;
end;

function testlower(c : char) : boolean;
begin
      if (c >= 'a') and (c <= 'z') then
            testlower := true
      else
```

Listing 12.2—Continued

```
            testlower := false;
end;

procedure uppercase(var s : string);
var
      i : integer;

begin
      for i := 1 to length(s) do
            if testlower(s[i]) then
                  s[i] := char(integer(s[i]) - offset);
end;

procedure lowercase(var s : string);
var
      i : integer;

begin
      for i := 1 to length(s) do
            if testupper(s[i]) then
                  s[i] := char(integer(s[i]) + off-
set);
end;

begin
end.
```

Listing 12.3 presents a short program, which demonstrates how the **uplow** unit is used in an actual program. As with any of Pascal's supplied units, **uplow** is made accessible to the main Pascal program by specifying its name in a **uses** statement. With the string conversion statements available in the **uplow** unit, the demonstration program simply converts a string of lowercase characters to uppercase and a string of uppercase characters to lowercase.

Listing 12.3

```
{ list12-3.pas - uplow demonstration program }
program uplow_demo;

uses uplow;

var
      s1, s2 : string;

begin
      s1 := 'this will be converted to all uppercase';
      s2 := 'THIS WILL BE CONVERTED TO ALL LOWERCASE';
      uppercase(s1);
      lowercase(s2);
      writeln(s1);
      writeln(s2);
end.
```

Lesson 83: Identifiers with the Same Name

Sooner or later you are bound to write a program that has an identifier with the same name as one used in a unit. Obviously, two (or more) identifiers with the same name can lead to some very unexpected results if they aren't handled properly. Fortunately, Pascal provides a very simple solution to work around identifier name conflicts. Table 12.3 illustrates how a unit's identifier can be distinguished from an identifier either in the main program or another unit. As this figure shows, this task is accomplished by simply preceding the identifier with it's corresponding unit name. Just think of the unit as a big record and the identifier as one of the record's field names.

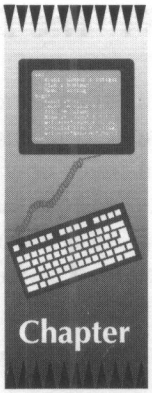
Table 12.3 *Conflicting identifier references*

unit name.identifier	
Where:	
unit name	is the name of the identifier's unit.
identifier	is the conflicting identifier.

Listing 12.4 presents a short program, which demonstrates how conflicting identifiers are referenced in a Pascal program. This program makes use of the **uplow** unit, which was presented earlier in this chapter. To provide a conflicting identifier, the program in Listing 12.4 defines a new **uppercase** procedure, which simply changes each character in a string to an *. To use the **uppercase** procedure in **uplow**, the program references the procedure as **uplow.uppercase**.

Listing 12.4

```
{ list12-4.pas - Demonstrate how conflicting identi-
fiers are referenced }
program conflict_ident;

uses uplow;

procedure uppercase(var s : string);
var
      i : integer;

begin
      for i := 1 to length(s) do
            s[i] := '*';
end;

var
      teststring : string;

begin
      teststring := 'I''m a test string!';
      writeln('teststring = ', teststring);
      uplow.uppercase(teststring);
      writeln('teststring = ', teststring);
      uppercase(teststring);
      writeln('teststring = ', teststring);
end.
```



13

Working with Strings

Although numeric data types are very important to the Pascal programmer, string data is probably the single most important data type the Pascal programmer is called upon to handle. To assist in manipulating string data, the Pascal programming language provides five very important string-related functions and procedures. This chapter will introduce the Pascal programmer to these five essential procedures and functions.

Lesson 84: The String Concatenation Function

Like the Pascal string concatenation operator (+) the Pascal string concatenation function, **concat**, combines one or more strings and returns the resulting string. Table 13.1 illustrates how the **concat** function is used. As you can see from this illustration, the **concat** function is very simple to use. Let's suppose a program had an expression of **'string one' + 'string two'**. The same expression could be rewritten as **concat('string one', 'string two')**.

Table 13.1 *The Pascal* ***concat*** *function*

concat(string expressions);	
Where:	
string expressions	are one or more string expressions separated by a comma.

Listing 13.1 presents a short program, which demonstrates how the Pascal **concat** function is used in an actual program. You will note from this listing that the program simply concatenates two strings together with both the string concatenation operator and the string concatenation function. As you will see by running the program, both methods for concatenating strings will return the same result.

Listing 13.1

```
{ list13-1.pas - Demonstrate the Pascal concat function }
program concat_demo;

var
      s1, s2, r1, r2 : string;

begin
      s1 := 'This is string 1 ';
      s2 := 'This is string 2 ';
      writeln('s1 + s2 = ', s1 + s2);
      writeln('concat(s1, s2) = ', concat(s1, s2));
end.
```

Lesson 85: The Pascal Copy Function

A lot of string manipulations require extracting a portion of one string to form another string. To perform such an operation, the Pascal programming language provides the **copy** function. Figure 13.2 shows how the **copy** function is used in a Pascal program. As this figure illustrates, the **copy** function returns a specified number of characters starting at a specified character position. If the specified character position exceeds the length of the string, then the **copy** function will return a null string. If the specified number of characters plus the specified character position exceeds the length of the string, only the remaining string characters will be returned.

Table 13.2 *The Pascal* ***copy*** *function*

copy(string, character position, number of characters);	
Where:	
string	is the source string.
character position	is the starting character position for the string to be copied.
number of characters	is the number of characters to be extracted.

Listing 13.2 presents a brief program that demonstrates how the **copy** function is used in an actual Pascal program. To demonstrate this task, the program simply extracts and displays a person's first name. As you can see from this short example, the Pascal **copy** function can be a very powerful tool for extracting a more pertinent piece of data from a large one.

Listing 13.2

```
{ list13-2.pas - Demonstrate the Pascal copy function }
program copy_demo;

var
     name, first : string;

begin
     name := 'John S. Doe';
     first := copy(name, 1, 4);
     writeln('First Name: ', first);
end.
```

Lesson 86: The Pascal Delete Procedure

Quite often a portion of a string must be removed in order to form a shorter string. The Pascal programming language provides a procedure called **delete** for performing just such an operation. Table 13.3 illustrates how the **delete** procedure is used. As this table shows, the **delete** procedure simply removes a specified number of characters starting at a specified character position. If the character position is larger than the string's length, then nothing will be removed from the string. If the number of characters plus the character position exceeds the length of the string, only the actual number of remaining string characters will be removed.

Table 13.3 *The Pascal* ***delete*** *function*

delete(string, character position, number of characters);	
Where:	
string	is the string to remove the substring from.
character position	is the starting character position for the substring to be deleted.
number of characters	is the number of characters to be deleted.

Listing 13.3 presents a short program that illustrates how the **delete** procedure is used in an actual Pascal program. To demonstrate how the **delete** procedure works, the program simply removes the middle initial from a person's name.

Listing 13.3

```
{ list13-3.pas - Demonstrate the Pascal delete procedure }
program delete_demo;

var
      s : string;

begin
      s := 'John S. Doe';
      delete(s, 6, 3);
      writeln(s);
end.
```

Lesson 87: The Pascal Insert Procedure

Although the **delete** procedure is certainly handy for removing unwanted characters from a string, what if the program requires characters to be inserted into a string? To provide for this very important task of inserting one string into another, the Pascal programming language offers the **insert** procedure. Figure 13.4 illustrates how the **insert** procedure is used. As this figure shows, the **insert** procedure simply inserts a source string into a destination string starting at a specified character position. If the resulting string's length is greater than 255 characters, the string result is truncated (chopped off at the right).

Table 13.4 *The Pascal* ***insert*** *procedure*

insert(source string, destination string, character position);	
Where:	
source string	is the string to be inserted.
destination string	is the string where the source string is inserted.
character position	is the starting character position where the source is inserted.

Listing 13.4 presents a short demonstration program that illustrates how the Pascal **insert** procedure is used in an actual program. This program demonstrates the insertion task by simply inserting a middle initial into a name string.

Listing 13.4

```
{ list13-4.pas - Demonstrate the Pascal insert procedure }
program insert_demo;

var
      s : string;

begin
      s := 'John Doe';
      insert(' S.', s, 5);
      writeln(s);
end.
```

Lesson 88: The Pascal Pos Position

One of the most important string-handling routines for any program is a routine to perform string searches. To assist in implementing a string search routine, the Pascal programming language provides the **pos** function. Table 13.5 illustrates how the **pos** function is used. As this figure shows, the **pos** function searches for one string in another. If the string is found in the string to be searched, the **pos** function returns the search string's starting character position in the string to be searched. Otherwise, the **pos** function returns a value of 0 to indicate the search string wasn't found.

Table 13.5 *The Pascal* ***pos*** *function*

pos(search string, string);	
Where:	
search string	is the string to search for.
string	is the string to be searched.

Listing 13.5 presents a brief program that demonstrates how the **pos** function is used in an actual Pascal program. To demonstrate how the Pascal **pos** function works, the program simply searches for a person's middle initial in a name string and displays the result.

Listing 13.5

```
{ list13-5.pas - Demonstrate the Pascal pos function }
program pos_demo;

var
     s : string;

begin
     s := 'John S. Doe';
     writeln('''S.'' is located at character position
', pos('S.', s));
end.
```

14

Console Input/Output

So far, we have closely examined many of the Pascal programming language's wide range of features. However, we have yet to take a look at how data can be input and output from a program. These next few chapters are devoted to explaining just how data is input and output with the Pascal programming language. This chapter explains how data is input and output from the console (keyboard and video display).

Lesson 89: The Write and Writeln Procedures

As their name implies, the Pascal **write** and **writeln** procedures are used to perform data output. The only real difference between the two procedures is what happens after they finish writing the actual data. The **write** procedure does absolutely nothing after it has sent data to the output device. After sending data to the output device, the **writeln** procedure sends a new line (carriage return/line feed combination) to the output device. Table 14.1 displays how the **write** and **writeln** procedures are used in a Pascal program. As this tables displays shows, each argument for a **write** or **writeln** procedure is a data item. This data item can be either a constant or a variable. Multiple arguments can be specified by separating them with commas. When executed, the **write** or **writeln** procedures simply display the data items' values in the order they appear in the argument list.

Table 14.1 *Using the* ***write*** *and* ***writeln*** *procedures for console output*

write(argument list);	
or	
writeln(argument list);	
Where:	
argument list	is a list of one or more data items. A data item can be a constant or a variable. Multiple data items are separated by commas.

Listing 14.1 presents a short Pascal program, which demonstrates how the **write** and **writeln** procedures are used in an actual program. Essentially, the program displays a string and two integer values with the **write** procedure first and then the same data items with the **writeln** procedure. By displaying the same exact data items with the two different procedures, the program clearly shows how the **writeln** procedure differs from the **write** procedure by generating a new line after displaying its arguments.

Listing 14.1

```
{ list14-1.pas - Demonstrate the write and writeln
procedures }
program write_writeln_demo;

var
      i1, i2 : integer;


begin
      i1 := 11;
      i2 := 33;
      write('See the difference between write and
writeln');
      write(i1, i2);
      writeln;
      writeln('See the difference between write and
writeln');
      writeln(i1, i2);
end.
```

Lesson 90: The Read and Readln Procedures

As the **write** and **writeln** procedures provide the means to send data to the console, the Pascal **read** and **readln** procedures input data from the console. Similar to the differences between the **write** and **writeln** procedures, the **read** and **readln** procedures differ slightly in the way they function. The **read** procedure only reads the input until its data arguments have been filled. Any remaining data is used by the next **read** procedure. The **readln** procedure keeps read-

ing data until a new line is encountered. If any data remains after the **readln** procedure's data arguments have been filled, the remaining data is ignored. When entering data with the **read** and **readln** procedures, each data entry item must be separated by a space, tab, or carriage return.

Table 14.2 displays how the **read** and **readln** procedures are used in a Pascal program. Like the **write** and **writeln** procedures, the **read** and **readln** procedures can have multiple data item arguments. Unlike the **write** and **writeln** procedure arguments, the **read** and **readln** arguments must be variables. This requirement is rather obvious due to the fact that you can't assign a value to a constant.

Table 14.2 *Using the* ***read*** *and* ***readln*** *procedures for console input*

read(argument list);	
or	
readln(argument list);	
Where:	
argument list	is a list of one or more data items. Unlike **write** and **writeln** arguments, **read** and **readln** arguments must be variables. Multiple data items are separated by commas.

Listing 14.2 presents a brief program, which demonstrates how the **read** and **readln** procedures are used in an actual Pascal program. This program also clearly demonstrates the differences between the **read** and **readln** procedures. The first **read** statement simply retrieves two integer values. The second set of **read** statements retrieves two integer values, but the operation is performed by two separate **read** statements. Note that if you enter both values as a response to the first **read** statement the program won't request any further input. Instead, the second **read** statement uses the remaining data from the first **read** statement. Unlike the dual **read** statements, the program's **readln** statements require that the operator specifically enter the data a line at a time.

Listing 14.2

```
{ list14-2.pas - Demonstrate the read and readln pro-
cedures }
program read_readln_demo;

var
      i1, i2 : integer;

begin
      read(i1, i2);
      writeln('i1 = ', i1, ' i2 = ', i2);
      read(i1);
      writeln('i1 = ', i1);
      read(i2);
      writeln('i2 = ', i2);
      readln(i1);
      writeln('i1 = ', i1);
      readln(i2);
      writeln('i2 = ', i2);
end.
```

Lesson 91: Formatted Output

Although the **write** and **writeln** procedures are very good at displaying data in an unformatted fashion, they also possess the ability to display data in a formatted manner. For example, you can tell either procedure to display a real number in a right-justified field with a specified width and a specified number of decimal places. Table 14.3 displays how a data item is formatted with either a **write** or **writeln** procedure. As this table displays, all formatted data items require a

width specification. If the specified width is a positive number, the data item is right-justified in a field of the specified width. If the specified width is a negative number, the data item is left-justified in a field of the specified width. Should the data item be wider than the specified width, it is displayed as an unformatted data item. Table 14.3 also displays that real number data items can optionally specify a number of decimal places.

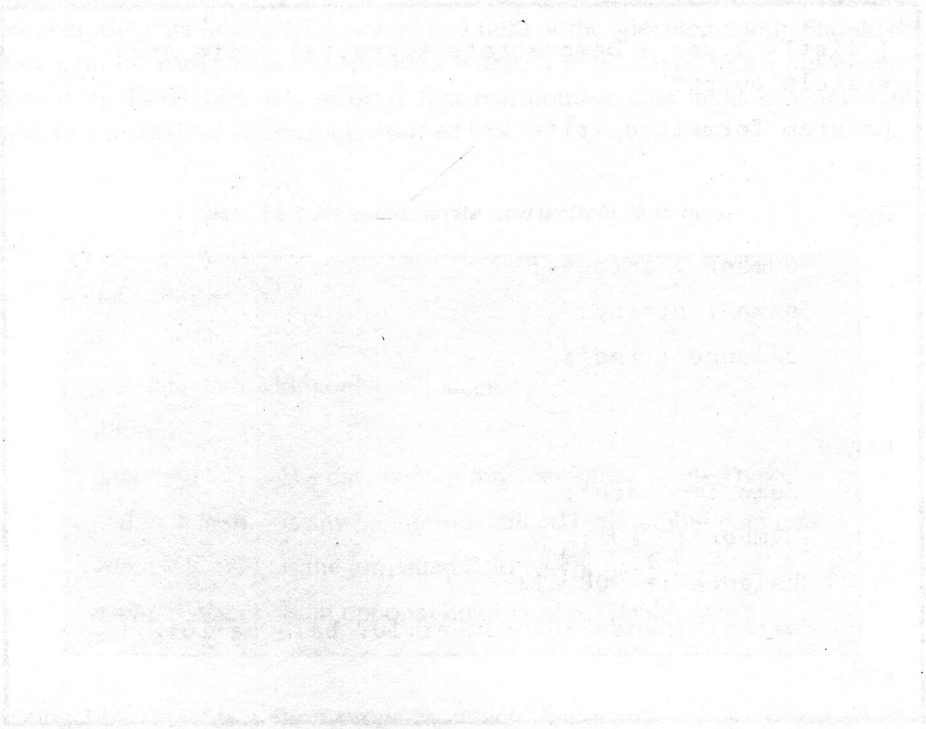
Table 14.3 *Formatted* ***write*** *and* ***writeln*** *data items*

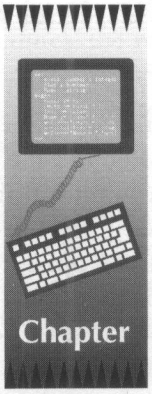
data item:width	
or	
real data item:width:decimal places	
Where:	
data item	is a data item of any previously defined type.
real data item	is any previously defined real number data type.
width	is the formatted field width.
decimal places	is an optional number of decimals places.

Listing 14.3 presents a short program, which demonstrates how formatted data items are specified in **write** and **writeln** statements. Essentially, this program displays an account's name, number, and balance as a line of formatted data output. Note how the program uses a width specifier of **-20** to left justify the account's name and a width specifier of **10:2** to display the account balance with two decimal places.

Listing 14.3

```
{ list14-3.pas - Demonstrate formatted write and
writeln output }
program formatted_write_writeln;

var
      number : integer;
      name : string;
      balance : real;

begin
      name := 'Cash';
      number := 101;
      balance := 100.31;
      writeln(name:-20, number:10, balance:10:2);
end.
```





15

Text File Input/Output

Whereas the previous chapter explained how data can be input and output from the computer console, console data input/output is transient in nature. To preserve and retrieve data to and from a more permanent type of medium, disk input/output is the preferred method. The Pascal programming language supports two basic types of disk files: text files and binary files. Like data sent to and retrieved from the console, data sent to and retrieved from text files is done in the form of ASCII strings. Consequently, a person would have little trouble reading the data in a text file by simply listing it. On the other hand, data sent to and retrieved from a binary file is done using the same formats Pascal uses to store data in the computer's internal memory. As a result, binary data files are virtually impossible for humans to read.

Lesson 92: Text Files

The obvious first step, in our study of text files, is to learn how to open a text file. To understand why a text file must first be opened, let's suppose that all of our data is stored in a file cabinet and not on a disk. To read some of the data in the cabinet, you must first open the appropriate file cabinet drawer. Like opening the drawer in the file cabinet, we must tell DOS that we want to use a disk file and to please open it for us. The first step in opening a text disk file is to declare a variable of type **text**. **Text** is a predefined data type just for working with text files. Table 15.1 displays how a **text** variable is declared. As this figure shows, a **text** variable is declared just like any of the other variable types we've covered in this book.

Table 15.1 *Declaring a* ***text*** *variable*

Var	
identifier **: text;**	
Where:	
identifier	is the **text** variable's name

With a **text** variable declared, the next step in opening the text file is to assign a file name to the **text** variable. To assign the file name to the **text** variable, the Pascal programming language provides a procedure called **assign**. Table 15.2 shows how the **assign** procedure is used in a program. As this table displays, assigning a file name to a **text** variable is accomplished by simply specifying the name of the text variable and the name of the file as the **assign** procedure's two arguments.

Table 15.2 *The Pascal* ***assign*** *procedure*

assign(file variable, file name);	
Where:	
file variable	is a previously declared file variable.
file name	is the name of the data file.

Once a file name has been assigned to the file variable, the disk file can be opened by using any one of three distinct Pascal procedures: the **rewrite** procedure, the **reset** procedure, and the **append** procedure. The **rewrite** procedure either creates a new file or opens an existing file. Using **rewrite** to open an existing file results in the loss of any data that already exists in the file. The **reset** procedure is used to open an existing file. However, the contents of the data file are preserved when opened by **reset**. The **append** procedure acts the same as the **reset** procedure except the **file pointer** is set to the end of the file. The file pointer is an internal pointer that points to the current location being accessed in a file. By using the **append** procedure, a file can be quickly opened for data to be added to the end of the file. Tables 15.3, 15.4, and 15.5 display how the **rewrite**, **reset**, and **append** procedures are used.

Table 15.3 *Opening a file with the* ***rewrite*** *procedure*

rewrite(file variable);
Where:
file variable is the variable for the file to be opened.

Table 15.4 *Opening a file with the* ***reset*** *procedure*

reset(file variable);
Where:
file variable is the variable for the file to be opened.

Table 15.5 *Opening a file with the* ***append*** *procedure*

append(file variable);
Where:
file variable is the variable for the file to be opened.

Once a file has been opened, it can be written to or read from by using the Pascal **read**, **readln**, **write**, and **writeln** procedures. Tables 15.6, 15.7, 15.8, and 15.display how these four procedures are used with text files. As you note from these tables, the only difference between using any one of these procedures with a disk file and with the console is the necessity of specifying a file variable for the procedure's first argument. By specifying a file variable as the first argument, Pascal is able to direct the data input/output to the proper file.

Table 15.6 *Reading file data with the* ***read*** *procedure*

read(file variable, data variables);	
Where:	
file variable	is a variable for the file to read the data from.
data variables	is a list of one or more data variables. Multiple data variables are separated by commas.

Table 15.7 *Reading file data with the* ***readln*** *procedure*

readln(file variable, data variables);	
Where:	
file variable	is a variable for the file to read the data from.
data variables	is a list of one or more data variables. Multiple data variables are separated by commas.

Table 15.8 *Writing file data with the* ***write*** *procedure*

write(file variable, data items);	
Where:	
file variable	is a variable for the file to write the data to.
data items	is a list of one or more constants or variables. Multiple data items are separated by commas.

Table 15.9 *Writing file data with the* ***writeln*** *procedure*

writln(file variable, data items);	
Where:	
file variable	is a variable for the file to write the data to.
data items	is a list of one or more constants or variables. Multiple data items are separated by commas.

After a data file's input/output operations have been completed, the file must be closed. An open file is closed by using the Pascal **close** procedure. Table 15.10 displays how the **close** procedure is used in a program. As this figure shows, a file is closed by simply specifying its file variable as the **close** procedure's one and only argument.

Table 15.10 *Closing a file with the* ***close*** *procedure*

close(file variable);	
Where:	
file variable	is the variable for the file to be closed.

Listing 15.1 presents a short Pascal program, which demonstrates how a Pascal text file is accessed in an actual program. This program starts by opening a text file, writing 10 lines of data to the file, and closing the file. With the data safely stored away on disk, the program continues by re-opening the text file, reading and displaying the 10 lines of data, and then reclosing the file. You should note the program's use of Pascal's **eof** function. This function returns **true** if a specified file's file pointer is located at the end of the file's data. Otherwise, the **eof** function returns **false** to indicate that the file pointer isn't pointing to the end of the data file. By using the **eof** function in a **while** loop, the program is able to easily read in all of the files data. The program simply continues to read data until **eof** returns **true**.

Listing 15.1

```
{ list15-1.pas - Text file demonstration program }
program text_file_demo;

var
      datafile : text;
      i : integer;
      s : string;

begin
      assign(datafile, 'textdemo.dat');
      rewrite(datafile);
      for i := 1 to 10 do
            writeln(datafile, 'This is data item no.
', i);
      close(datafile);
      reset(datafile);
      while not eof(datafile) do
      begin
            readln(datafile, s);
            writeln(s);
      end;
      close(datafile);
end.
```

Lesson 93: Error Trapping

Although today's disk drives are very reliable storage devices, errors do occur from time to time. Consequently, all but the simplest of data handling programs should provide at least a minimal amount of error handling. Pascal's normal input/output error handler simply generates a run-time error when an error occurs. Obviously, this crude error handling method is a little too simplistic for most programs.

To assist Pascal programmers in dealing with input/output errors, Pascal provides the **{$I-}** and **{$I+}** compiler directives. A compiler directive simply tells the Pascal compiler to switch certain features on and off. In the case of the input/output directives, the **{$I-}** compiler directive tells Pascal not to generate run-time errors when an input/output error occurs. The **{$I+}** compiler directive tells Pascal to generate run-time errors whenever an input/output error occurs.

With the Pascal program set to **{$I-}**, a call to the Pascal **ioresult** function can be used to determine if an input/output error has occurred. If **ioresult** returns a value of 0, the last input/output operation returned without an error. Otherwise, **ioresult** returns a non-zero value to indicate that the last input/output operation returned with an error.

Listing 15.2 presents a revised version of the program that was presented in Listing 15.1. As you can clearly see, this revised version of the text file demonstration program utilizes the **{$I-}** and **{$I+}** compiler directives to provide a simple error handling routine. Although this program only displays a relevant error message, the program could be further modified to provide for a much more sophisticated error handler.

Listing 15.2

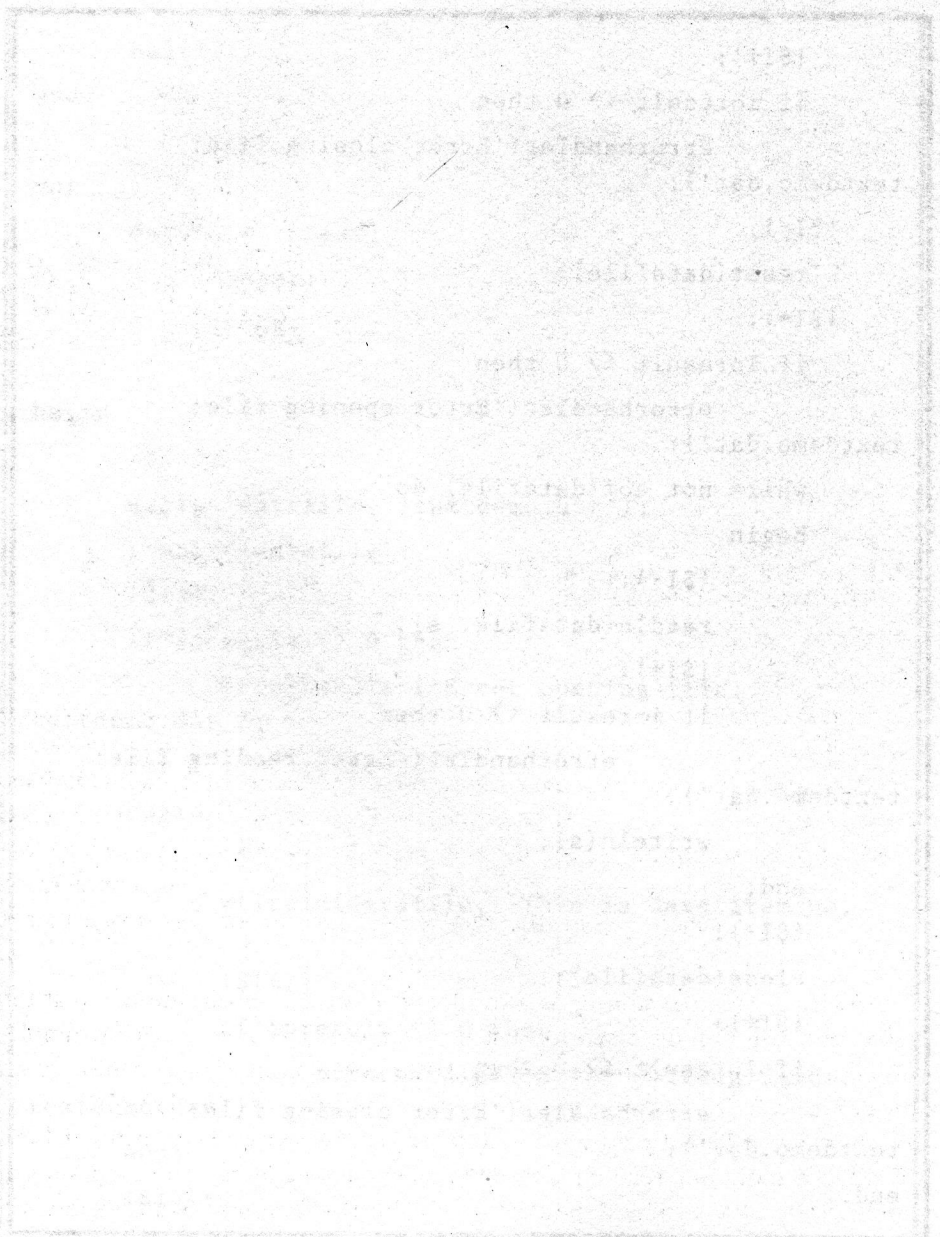
```
{ list15-2.pas - Error handling demonstration program }
program error_handler_demo;

procedure errorhandler(s : string);
begin
      writeln(s);
```

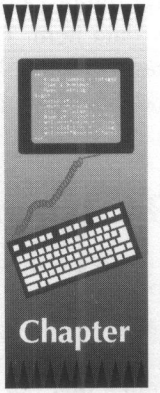
Listing 15.2—Continued

```
        halt(1);
end;

var
        datafile : text;
        i : integer;
        s : string;

begin
        {$I-};
        assign(datafile, 'textdemo.dat');
        rewrite(datafile);
        {$I+};
        if ioresult <> 0 then
                errorhandler('Error opening file:
textdemo.dat');
        for i := 1 to 10 do
        begin
                {$I-};
                writeln(datafile, 'This is data item no.
', i);
                {$I+};
                if ioresult <> 0 then
                        errorhandler('Error writing file:
textdemo.dat');
        end;
        {$I-};
        close(datafile);
```

Listing 15.2—Continued

```
      {$I+};
      if ioresult <> 0 then
            errorhandler('Error closing file:
textdemo.dat');
    {$I-};
      reset(datafile);
    {$I+};
      if ioresult <> 0 then
            errorhandler('Error opening file:
textdemo.dat');
      while not eof(datafile) do
      begin
            {$I-};
            readln(datafile, s);
            {$I+};
            if ioresult <> 0 then
                  errorhandler('Error reading file:
textdemo.dat');
            writeln(s);
      end;
      {$I-};
      close(datafile);
      {$I+};
      if ioresult <> 0 then
            errorhandler('Error closing file:
textdemo.dat');
end.
```





16

Binary File Input/Output

Although the text files presented in the previous chapter are very useful for storing string data, they are very inefficient for storing numeric data types. Consequently, the Pascal programming language also supports binary data files. With a binary data file, data is stored on disk using the same format that is used to store it in the computer's memory. Because a data item's internal binary representation almost always requires less memory than its ASCII string counterpart, storing data in a binary file greatly reduces the amount of disk space that is required to store the data.

Another benefit of storing data in binary files comes from the fact that the Pascal program is fully aware of just how large a data item is. For example, Pascal stores **integer**s as two binary bytes. An ASCII string representation of an **integer** requires from one (as in the **0**) to six (as in **-19999**) characters. Consequently, a Pascal program could never accurately extract an **integer** from a text file without first reading all of the preceding data items. The program just simply doesn't know where the data item is located in a text file. This method for accessing data in a text file is called *sequential access*. Because the Pascal program knows just how large each data item is in a binary file, it can position

the file pointer right to a desired data item and either read its contents or replace it with a new data item. This method for accessing data in a binary file is called *random access*. Obviously in all but the simplest of files, using the random access method is usually the preferred method for accessing data.

Lesson 94: Typed Binary Files

The Pascal programming language supports two types of binary files: typed binary files and untyped binary files. In this lesson, we explore how typed binary files are used. The process for opening a typed binary file is almost identical to opening a text file. As with a text file, this first step in opening a binary file is to declare a file variable. Table 16.1 displays how file variables are declared for typed binary files. As this table displays, a typed binary file variable declaration is pretty much like any other variable declaration.

Table 16.1 *Declaring a typed binary file variable*

Var	
identifier **: file of** data type;	
Where:	
identifier	is the typed binary file variable's name.
data type	is a previously defined data type.

As with opening a text file, the next step in opening a typed binary file is to assign a file name to the typed binary file's variable with the **assign** procedure. Once a file name has been assigned to a typed binary file variable, the file can be opened with either the **rewrite** or the **reset** procedures. Data is read from a file with the **read** procedure and data is written to a file with the **write** procedure. Table 16.2 displays how the **read** procedure is used to read data from a typed binary file and Table 16.3 displays how the **write** procedure is used to write data to a typed binary file. As these figures show, all data items in a **read** or **write** argument list must be of the same data type as was used in the typed binary file's variable declaration. Like text files, typed binary files are closed with the **close** procedure.

Table 16.2 *Reading typed binary file data with the* ***read*** *procedure*

read(file variable, data variables);	
Where:	
file variable	is a typed binary file variable for the file to read the data from.
data variable	is a list of one or more variables with the same data type as was used in file variable's declaration. Multiple data variables are separated by commas.

Table 16.3 *Writing typed binary file data with the* ***write*** *procedure*

write(file variable, data items);	
Where:	
file variable	is a typed binary file variable for the file to write the data to.
data items	is a list of one or more constants or variables with the same data type as was used in file variable's declaration. Multiple data variables are separated by commas.

With all of the above mentioned file-handling procedures, the Pascal programmer could write a very efficient sequential access data file. To be able to randomly access a file, the Pascal programmer needs the assistance of the **seek** procedure and the **filepos** function.

The **seek** procedure is used to move a typed binary file's file pointer to a desired location. Table 16.4 displays how the **seek** procedure is used. As this table shows, the **seek** procedure's second argument is used to specify the record number for the file pointer's new location. Note that a file's first record is record 0 and not record 1. Record 1 is the second record in a typed binary file.

Table 16.4 *The Pascal* ***seek*** *procedure*

seek(file variable, position);	
Where:	
file variable	is the typed binary file's variable.
position	is the record number to move the file pointer to.

As its name implies, the Pascal **filepos** function returns the current record number for the file's current file pointer position. The value returned by the **filepos** function is of type **longint**. Table 16.5 displays how the **filepos** function is used. As this table shows, the **filepos** function's one and only argument is the typed binary file's variable.

Table 16.5 *The Pascal* ***filepos*** *function*

filepos(file variable);	
Where:	
file variable	is the typed binary file's variable.

Listing 16.1 presents a short demonstration program, which illustrates how a typed binary file can be used to perform random access. The program starts by creating a typed binary file of type **integer** and filling the file with dummy integer values. The program continues by re-opening the file and reading and displaying the dummy integer values back in reverse order. Obviously, reading a file backwards would be impossible to do with a text file. Although this is a rather simple example of randomly accessing a data file, it clearly shows some of the power offered by random access data files.

Listing 16.1

```
{ list16-1.pas - Typed binary file demo }
program typed_bin_file;

var
      datafile : file of integer;
      i, rec : integer;

begin
      writeln('Writing demo file.....');
      assign(datafile, 'demofile.dat');
      rewrite(datafile);
      for i := 1 to 10 do
            write(datafile, i);
      close(datafile);
      write('Reading demo file backwards.....');
      reset(datafile);
      for i := 10 downto 1 do
      begin
            seek(datafile, i - 1);
            read(datafile, rec);
            write(rec, '...');
      end;
      close(datafile);
      writeln;
end.
```

Lesson 95: Untyped Binary Files

Although the typed binary files presented in the previous lesson are by far the most commonly used of the Pascal binary file types, the second method for storing data as a binary file is called the *untyped binary file*. Because an untyped binary file doesn't have a data type, data can be read from and written to the file without regard for its data type. Instead of reading and writing data in the form of constants and variables, untyped file data is stored in buffer areas.

The steps for opening an untyped binary file are very similar to opening a typed binary file. The first step is to declare an untyped binary file variable. Table 16.6 displays how an untyped binary file variable is declared. As this table shows, the Pascal programming language provides a predefined data type called **file** for declaring untyped binary file variables.

Table 16.6 *Declaring an untyped binary file variable*

Var
identifier : **file;**
Where:
identifier is the untyped binary file variable's name.

With the variable properly declared, the untyped binary file is opened by first assigning a file name to the variable with the **assign** procedure and then opening it with either the **rewrite** procedure or the **reset** procedure. Unlike how **rewrite** and **reset** are used with other Pascal file types, these two procedures can also be used to specify a record size for the untyped binary file. Tables 16.7 and 16.8 display how the **rewrite** and the **reset** procedures are used in a program. As these tables show, an untyped binary file has a record length of 128 if the default record size argument is omitted.

Table 16.7 *Opening an untyped binary file with the* ***rewrite*** *procedure*

rewrite(file variable, record size);	
Where:	
file variable	is the variable for the untyped binary file to be opened.
record size	is an optional record size for the untyped binary file. If the record size argument is omitted, a default record size of 128 is used for the untyped binary file.

Table 16.8 *Opening an untyped binary file with the* ***reset*** *procedure*

reset(file variable, record size);	
Where:	
file variable	is the variable for the untyped binary file to be opened.
record size	is an optional record size for the untyped binary file. If the record size argument is omitted, a default record size of 128 is used for the untyped binary file.

Because an untyped binary file doesn't have a data type associated with it, the Pascal **read** and **write** procedures can't be used to read from and write to an untyped binary file. Instead, Pascal offers the **blockread** and **blockwrite** procedures for dealing with untyped binary files. Table 16.9 displays how the **blockread** procedure and Table 16.10 displays how the **blockwrite** procedures are used in a Pascal program. As these figures show, the second argument is a pointer to a predeclared buffer area. This buffer area is simply an array that is big enough to hold the number of records defined in the procedures' third argument. The fourth argument for both procedures is optional and returns the actual number of records that the procedures either read or write. After all of the read/write operations are completed, an untyped binary file is closed with the Pascal **close** procedure.

Table 16.9 *Reading data with the Pascal* ***blockread*** *procedure*

blockread(file variable, buffer, number of records, number read);	
Where:	
file variable	is the variable for the untyped binary file to be read.
buffer	is a variable large enough to hold the block to be read. This variable is usually an array.
number of records	is the number of records to read.
number read	returns the actual number of records read.

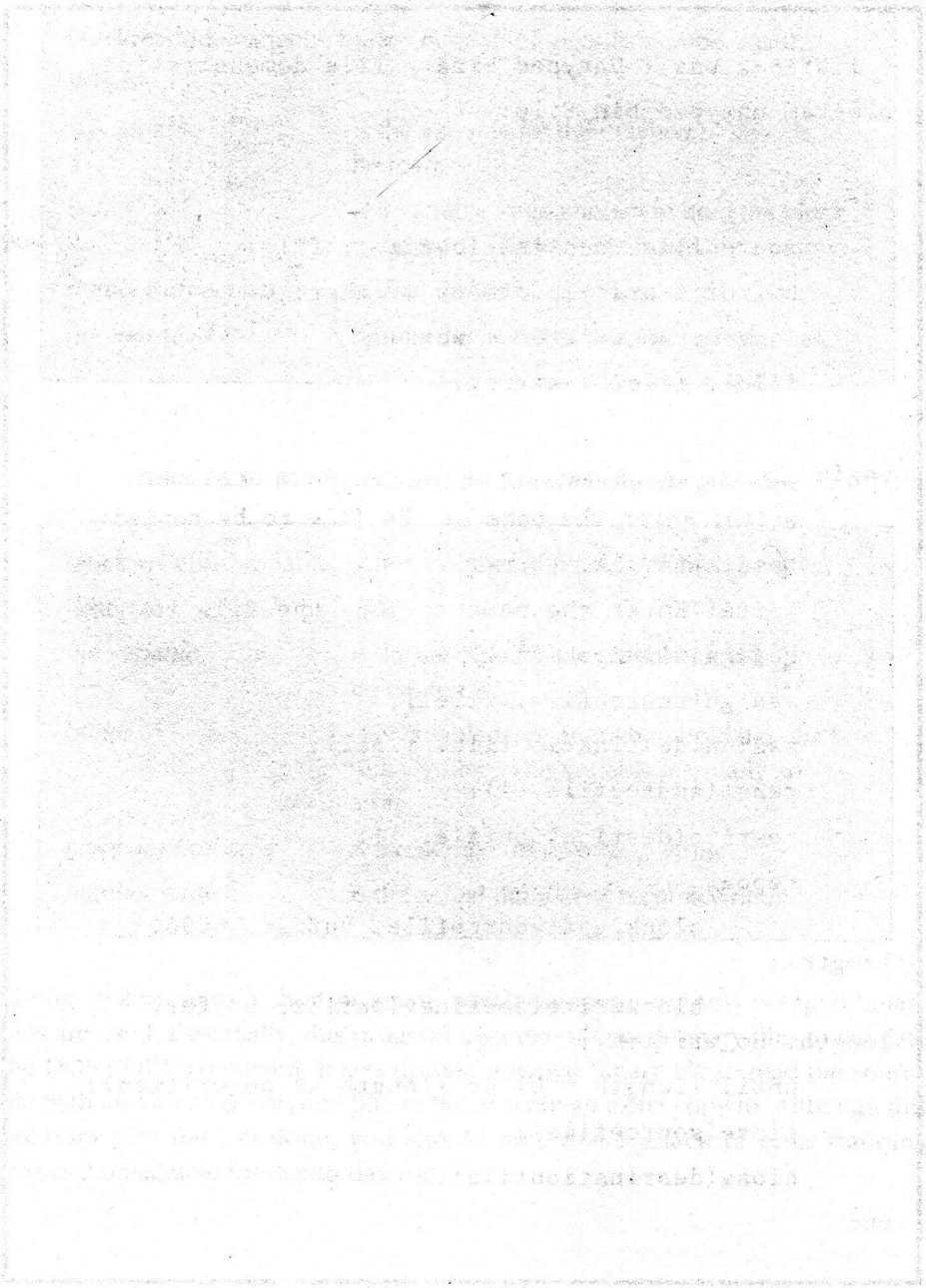
Table 16.10 *Writing data with the Pascal* ***blockwrite*** *procedure*

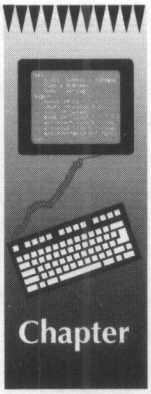
blockwrite(file variable, buffer, number of records, number written);	
Where:	
file variable	is the variable for the untyped binary file to be written to.
buffer	is a variable large enough to hold the block to be written. This variable is usually an array.
number of records	is the number of records to write.
number written	is the actual number of records written.

Listing 16.2 presents a short program, which demonstrates how untyped binary files are used. Essentially, this program uses two untyped binary files to emulate the DOS COPY command. It sets up one untyped binary file to read the source file with and another untyped binary file to write an exact copy to. Although this program gets the job done, you should note that it's lack of error trapping makes it unsuitable for day-to-day use.

Listing 16.2

```
{ list16-2.pas - Untyped binary file demonstration}
program untyped_bin_file;

var
      sourcefile, destinationfile : file;
      buffer : array[1..4096] of char;
      length, no_written : word;
      file1, file2 : string;

begin
      write('Enter the name of the file to be copied: ');
      readln(file1);
      write('Enter the name to copy the file to: ');
      readln(file2);
      assign(sourcefile, file1);
      assign(destinationfile, file2);
      reset(sourcefile, 1);
      rewrite(destinationfile, 1);
      repeat
            blockread(sourcefile, buffer, 4096,
length);
            blockwrite(destinationfile, buffer,
length, no_written);
      until (length = 0) or (length <> no_written);
      close(sourcefile);
      close(destinationfile);
end.
```





17

Object-Oriented Programming

Without a doubt, object-oriented programming is the hottest area in computer programming today. This chapter is devoted to presenting object-oriented programming as it applies to Turbo Pascal. The topics covered include: encapsulation, inheritance, and polymorphisim.

Lesson 96: Encapsulation

With traditional programming methods, programs are usually written by first designing the program's code and then creating the data structures to go along with the resulting code. This leads to a kind of second class status for the program data. With object-oriented programming, code and data are considered equal partners in the creation of a program. In an object-oriented program, the programmer defines what are known as object classes. These object classes are similar to records, but unlike records they can also have their own procedures and functions. An object class's data fields are called *instance variables* and its procedures and functions are called *methods*. This merging of data fields, procedures, and functions into a single object class is called *encapsulation*. Encapsulation is probably one of the most important concepts object-oriented programming techniques offer.

Table 17.1 displays how an object class is defined. As this table shows, an object class definition looks a great deal like a record definition. The object class's field declarations are defined using the same methods used for defining record field declarations. The object class's method declarations are nothing more than procedure and function prototypes. Note that the order of the field and method declarations is unimportant. However, most programmers declare the object class's instance variables before defining its methods.

Like any other Pascal procedure or function, an object class's methods must also have a definition. Table 17.2 displays how an object class procedure is defined and table 17.3 illustrates how an object class function is defined. As these figures show, the only difference between these types of procedure and function definitions and a regular procedure or function definition is the way the procedure or function name is constructed. All method definition names take the form of *object class.method name*. To return a value from an object class function, the return value is simply assigned to the *method name* as is displayed in Table 17.4.

Table 17.1 *Defining a Turbo Pascal object class*

```
type
        class name = object
                field declaration;
                .
                .
                field declaration;
                method declaration;
                .
                .
                method declaration;
        end;
```

Where:

class name	is the new object class's identifier.
field declaration	is a valid file declaration.
method declaration	is a **procedure** or **function** prototype.

Table 17.2 *Defining an object class procedure*

```
procedure object class.method name(parameter list);
begin
        statement;
        .
        .
        statement;
end;
```

Where:

object class	is the object class's name.
method name	is the method's name.
parameter list	is a list of arguments to be passed to the procedure.
statement	is a valid program statement.

Table 17.3 *Defining an object class function*

```
function object class.method name(parameter list) : return type;
begin
        statement;
        .
        .
        statement;
end;
```

Where:

object class	is the object class's name.
method name	is the method's name.
parameter list	is a list of argument's to be passed to the function.
return type	is the data type for the function's return value.
statement	is a valid Pascal program statement.

Table 17.4 *Return values from an object class function*

```
method name := return value;
```

Where:

method name	is the object class function's method name.
return value	is a return value of the proper data type.

Like an ordinary Pascal variable, an object (or **class instance**) must be declared before it can be used in a program. Table 17.5 displays how an object is declared. As this table shows, an object declaration is identical to any other variable declaration.

Table 17.5 *Declaring a Turbo Pascal object*

Var	
object name : object class;	
Where:	
object name	is a valid Pascal identifier.
object class	is a previously defined object class's name.

Once an object has been declared, its fields can be accessed using the same format for accessing record fields. Table 17.6 displays how an object's fields are referenced. Although there is absolutely no reason why this method for accessing an object's instance variables can not be used just about anywhere in a Pascal program, it is considered a violation of object-oriented programming techniques to access an instance variable outside of the object class's methods. The reason for this is what's known as data hiding. By not allowing a programmer to directly access instance variables outside of the object class's definitions, it insulates him from actually having to know the nitty gritty details about an object class. Instead, object class procedures and functions should be defined for setting and retrieving instance variable values.

Table 17.6 *Referencing instance variables*

object name.field name	
Where:	
object name	is the name of a previously declared class instance.
field name	is the name of one of the object class's fields.

Table 17.7 displays how an object class method is called. Other than preceding the *method name* with the *object name* and a period (.) there is no difference between a method call and a regular Pascal procedure or function call. Consequently, an object method is no harder to call than any other Pascal procedure or function.

Table 17.7 *Calling an object's method*

object name.method name(argument list)	
Where:	
object name	is the name of a previously declared object.
method name	is the name of the object class procedure or function to be called.
argument list	is a list of arguments to be passed to the procedure or function.

Referencing an object class's instance variables or methods inside of the object class's methods is done a little different from references outside of the object class. The only difference is that the object's name is not required. For that matter, it would be impossible to specify an object's name from inside of a method. At times name conflicts can arise between the object's instance variable and method names and other identifiers used in the object class's methods. To prevent these name conflicts from occurring, the conflicting instance variable or method name can be preceded by the keyword **self** and a period (.). The **self** identifier is equivalent to using the object's name outside of the object class's methods.

Listing 17.1 presents a short program, that clearly demonstrates how the object-oriented principle of encapsulation can be used in a Turbo Pascal program. Also note how methods have been defined for the program's object class to achieve data hiding. By using the technique of data hiding, the instance variables are never directly accessed from outside of the object class. Although this example of data hiding isn't overwhelming by any means, you may find data hiding is a very useful technique for reducing errors and program development time in more complex object-oriented programs.

Listing 17.1

```
{ list17-1.pas - Encapsulation demo }
program encap_demo;

type
      employee = object
            name : string;
            age : integer;
            procedure init(n : string; a : integer);
            procedure display;
            procedure setname(n : string);
            procedure setage(a : integer);
            function getname : string;
            function getage : integer;
      end;

procedure employee.init(n : string; a : integer);
begin
      name := n;
      age := a;
end;

procedure employee.display;
begin
      writeln('Employee''s name: ', name);
      writeln('Employee''s age: ', age);
end;
procedure employee.setname(n : string);
begin
```

Listing 17.1—Continued

```
      name := n;
end;

procedure employee.setage(a : integer);
begin
      age := a;
end;

function employee.getname : string;
begin
      getname := name;
end;

function employee.getage : integer;
begin
      getage := age;
end;

var
      e1, e2 : employee;

begin
      e1.init('John Smith', 33);
      e1.display;
      e2.setname('Jane Doe');
      e2.setage(28);
      writeln('Employee''s name: ', e2.getname);
      writeln('Employee''s age: ', e2.getage);
end.
```

Lesson 97: Inheritance

Although encapsulation is the cornerstone of object-oriented programming, inheritance is what makes object-oriented programming shine. Essentially, inheritance allows the programmer to define an object class (called a *subclass*) based upon a previously defined object class (called a *parent class*). Table 17.8 displays how an object subclass is defined. You will note from this illustration that the only difference between defining a subclass and a parent class is the inclusion of the parent class's name following the **object** keyword.

Besides being able to utilize its own instance variables and methods, a subclass can also use any instance variables and methods found in the parent class. In a sense, the instance variables and methods available to the subclass are a superset of the parent class's instance variables and methods. Although a subclass can utilize all portions of the parent class, the reverse is not true. The parent class has absolutely no idea that the subclass even exists; therefore, it is impossible for the parent class to take advantage of one of its subclass's instance variables or methods.

Table 17.8 *Defining a Turbo Pascal object subclass*

```
type
        class name = object(parent class)
                field declaration;
                .
                .
                field declaration;
                method declaration;
                .
                .
                method declaration;
        end;
```

Where:

class name	is the new object subclass's identifier.
parent class	is the name of the subclass's parent class.
field declaration	is a valid field declaration.
method declaration	is a **procedure** or **function** prototype.

Listing 17.2 is a variation of the program that was presented in Listing 17.1. This newer version of the program uses inheritance to create a **secretary** subclass and an **executive** subclass for the **employee** parent class. Note how these new subclasses define their own unique instance variables and methods while retaining the instance variables and methods of the parent class. Notice how the **employee** parent class is never used to declare an object with. Its only purpose is to serve as the parent class of the two new subclasses. You may think this odd at first, but it is a very common occurrence to define an object class solely for the purpose of deriving subclasses from it.

Listing 17.2

```
{ list17-2.pas - Inheritance demo }
program inherit_demo;

type
      employee = object
            name : string;
            age : integer;
            procedure init(n : string; a : integer);
            procedure display;
            procedure setname(n : string);
            procedure setage(a : integer);
            function getname : string;
            function getage : integer;
      end;

      secretary = object(employee)
            wpm : integer;
            procedure setwpm(w : integer);
            function getwpm : integer;
```

Listing 17.2—Continued

```
      end;

      executive = object(employee)
            keys : boolean;
            procedure setkeys(k : boolean);
            function getkeys : boolean;
      end;

procedure employee.init(n : string; a : integer);
begin
      name := n;
      age := a;
end;

procedure employee.display;
begin
      writeln('Employee''s name: ', name);
      writeln('Employee''s age: ', age);
end;

procedure employee.setname(n : string);
begin
      name := n;
end;

procedure employee.setage(a : integer);
```

Listing 17.2—Continued

```
begin
      age := a;
end;

function employee.getname : string;
begin
      getname := name;
end;

function employee.getage : integer;
begin
      getage := age;
end;

procedure secretary.setwpm(w : integer);
begin
      wpm := w;
end;

function secretary.getwpm : integer;
begin
      getwpm := wpm;
end;
```

Listing 17.2—Continued

```
procedure executive.setkeys(k : boolean);
begin
      keys := k;
end;

   function executive.getkeys : boolean;
   begin
      getkeys := keys;
   end;

   var
      e1 : executive;
      e2 : secretary;

   begin
      e1.init('John Smith', 33);
      e1.setkeys(true);
      e1.display;
      writeln('Executive Washroom Keys? ',
   e1.getkeys);
      e2.init('Jane Doe', 28);
      e2.setwpm(100);
      e2.display;
      writeln('Words per minute: ', e2.getwpm);
   end.
```

Lesson 98: Polymorphism

Polymorphism is the ability in an object-oriented programming language to allow subclasses to redefine methods found in their parent classes. Although polymorphism may not sound all that special, it is one of the most powerful tools at the object oriented programmer's disposal. Polymorphic methods are called *virtual methods*. Virtual methods are defined by simply following the method's prototype in both the parent class and the subclass with the **virtual** keyword. Once a method has been declared as **virtual**, it must remain **virtual** throughout all of the parent classes succeeding subclasses. Table 17.9 displays how a subclass with virtual methods is defined.

This table also illustrates how a special procedure called a **constructor** is defined. Essentially, a **constructor** is the same as a regular Pascal procedure except that the **procedure** keyword is replaced with the **constructor** keyword and it performs a few internal tasks to enable the use of virtual methods in an object class. You should note that any object class with virtual methods must have a **constructor**. Furthermore, each and every class instance must call the **constructor** before calling any of the class's other methods. Failure to call the **constructor** first most likely results in a fatal program error. Because the **constructor** must be called for each object, it is common practice to make the **constructor** an initialization routine for the object class.

Table 17.9 *Defining a Turbo Pascal polymorphic subclass*

```
type
        class name = object(parent class)
                field declaration;
                .
                .
                field declaration;
                constructor declaration;
                method declaration; virtual;
                .
                .
                method declaration; virtual;
                method declaration;
                .
                .
                method declaration
        end;
```

Where:

class name	is the new object subclass's identifier.
parent class	is the parent class's identifier.
field declaration	is a valid field declaration.
constructor declaration	is the object class's **constructor** declaration.
method declaration	is a **procedure** or **function** prototype.

Besides being able to call the subclass's virtual methods, the parent class's inherited methods can still be called. Table 17.10 displays how a virtual method's inherited method is called. As this table shows, an inherited method is called by simply preceding the method name with the parent class's name and a period (.).

Table 17.10 *Calling an inherited virtual method*

parent class.method name(**argument list**)	
Where:	
parent class	is the parent class's identifier.
method name	is the name of the virtual method.
argument list	is a list of arguments to be passed to the **procedure** or **function**.

Listing 17.3 presents a short program, which demonstrates how polymorphism is used in an actual Pascal program. This program is simply a more refined version of the programs that appeared in Listings 17.1 and 17.2. This newer version defines the two subclasses' **display** methods as virtual methods. The program also demonstrates how inherited methods can still be called by calling the inherited parent class's **display** method from in the new virtual methods. Also note how the subclasses' **constructors** serve two purposes: acting as Turbo Pascal's internal housekeeper and initializing the object.

Listing 17.3

```
{ list17-3.pas - Polymorphism demo }
program poly_demo;

type
     employee = object
          name : string;
          age : integer;
          constructor init(n : string; a : integer);
          procedure display; virtual;
          procedure setname(n : string);
```

Listing 17.3—Continued

```
            procedure setage(a : integer);
            function getname : string;
            function getage : integer;
      end;

      secretary = object(employee)
            wpm : integer;
            constructor init(n : string; a, w : integer);
            procedure display; virtual;
            procedure setwpm(w : integer);
            function getwpm : integer;
      end;

      executive = object(employee)
            keys : boolean;
            constructor init(n: string; a : integer; k
                  : boolean);
            procedure display; virtual;
            procedure setkeys(k : boolean);
            function getkeys : boolean;
      end;

constructor employee.init(n : string; a : integer);
begin
      name := n;
      age := a;
end;
```

Listing 17.3—Continued

```
procedure employee.display;
begin
      writeln('Employee''s name: ', name);
      writeln('Employee''s age: ', age);
end;

procedure employee.setname(n : string);
begin
      name := n;
end;

procedure employee.setage(a : integer);
begin
      age := a;
end;

function employee.getname : string;
begin
      getname := name;
end;

function employee.getage : integer;
begin
      getage := age;
end;
```

Listing 17.3—Continued

```
constructor secretary.init(n : string; a, w : integer);
begin
      employee.init(n, a);
      wpm := w;
end;

procedure secretary.display;
begin
      employee.display;
      writeln('Words per minute:', wpm);
end;

procedure secretary.setwpm(w : integer);
begin
      wpm := w;
end;

function secretary.getwpm : integer;
begin
      getwpm := wpm;
end;

constructor executive.init(n : string; a : integer; k
: boolean);
begin
      employee.init(n, a);
      keys := k;
end;
```

Listing 17.3—Continued

```
procedure executive.display;
begin
      employee.display;
      writeln('Executive Washroom Keys? ', keys);
end;

procedure executive.setkeys(k : boolean);
begin
      keys := k;
end;

function executive.getkeys : boolean;
begin
      getkeys := keys;
end;

var
      e1 : executive;
      e2 : secretary;

begin
      e1.init('John Smith', 33, true);
      e2.init('Jane Doe', 28, 100);
      e1.display;
      e2.display;
end.
```

Lesson 99: Dynamic Objects

As with other data types, objects can be dynamically allocated and deallocated by using the **new** and **dispose** procedures. Table 17.11 displays how objects are dynamically allocated with the **new** procedure. As this figure shows, an object's constructor can be called as the **new** procedure's second argument. Obviously, a constructor is only required for object classes that utilize virtual methods. Although the constructor can be called as the **new** procedure's second argument, the constructor can still be called in another program statement after the **new** procedure has been called.

Table 17.11 *Dynamically allocating an object with Turbo Pascal*

new(object pointer)

or

new(object pointer, constructor call);

Where:

object pointer	is a pointer to the object to be dynamically allocated.
constructor call	is an optional constructor call. Because the object hasn't yet been assigned a name, only the constructor's method name is required for the constructor call.

Table 17.12 displays how a dynamically allocated object is deallocated with the **dispose** procedure. This table also shows how a call to a special **destructor** procedure can be specified as the **dispose** procedure's second argument. A **destructor** procedure is declared by substituting the **destructor** keyword for the **procedure** keyword in the object class definition. Essentially, the destructor procedure is used to correctly deallocate dynamic memory and must be specified as the **dispose** procedure's second argument. To insure that Turbo Pascal deallocates the proper number of bytes, all dynamic objects should have a destructor. Furthermore, it is customary to specify any other clean up chores within the destructor's definition.

Table 17.12 *Dynamically deallocating an object with Turbo Pascal*

dispose(object pointer);

or

dispose(object pointer, destructor call);

Where:

object pointer	is a pointer to the object to be dynamically deallocated.
destructor call	is an optional destructor call. Note that only the destructor's method name is required for the destructor call.

Listing 17.4 presents a program, which demonstrates how objects are dynamically allocated in an actual Pascal program. Essentially, this program is a revised version of the program presented in Listing 17.3. Instead of using static objects, this revised version uses dynamically allocated objects to accomplish the same tasks. Note the use of destructors to insure that the proper deallocation of dynamic memory is accomplished.

Listing 17.4

```
{ list17-4.pas - Dynamically allocated object demo }
program dynamic_demo;

type
     employee = object
          name : string;
          age : integer;
          constructor init(n : string; a : integer);
          destructor done; virtual;
          procedure display; virtual;
```

Listing 17.4—Continued

```
            procedure setname(n : string);
            procedure setage(a : integer);
            function getname : string;
            function getage : integer;
      end;

      secretary = object(employee)
            wpm : integer;
            constructor init(n : string; a, w : integer);
            destructor done; virtual;
            procedure display; virtual;
            procedure setwpm(w : integer);
            function getwpm : integer;
      end;

      executive = object(employee)
            keys : boolean;
            constructor init(n: string; a : integer; k
                  : boolean);
            destructor done; virtual;
            procedure display; virtual;
            procedure setkeys(k : boolean);
            function getkeys : boolean;
      end;

constructor employee.init(n : string; a : integer);
```

Listing 17.4—Continued

```
begin
   name := n;
   age := a;
end;

destructor employee.done;
begin
end;

procedure employee.display;
begin
   writeln('Employee''s name: ', name);
   writeln('Employee''s age: ', age);
end;

procedure employee.setname(n : string);
begin
   name := n;
end;

procedure employee.setage(a : integer);
begin
   age := a;
end;
```

Listing 17.4—Continued

```
function employee.getname : string;
begin
      getname := name;
end;

function employee.getage : integer;
begin
      getage := age;
end;

constructor secretary.init(n : string; a, w : inte-
ger);
begin
      employee.init(n, a);
      wpm := w;
end;

destructor secretary.done;
begin
end;

procedure secretary.display;
begin
      employee.display;
      writeln('Words per minute:', wpm);
end;
```

Listing 17.4—Continued

```
procedure secretary.setwpm(w : integer);
begin
   wpm := w;
end;

function secretary.getwpm : integer;
begin
   getwpm := wpm;
end;

constructor executive.init(n : string; a : inte-
ger; k : boolean);
begin
   employee.init(n, a);
   keys := k;
end;

destructor executive.done;
begin
end;

procedure executive.display;
begin
   employee.display;
   writeln('Executive Washroom Keys? ', keys);
end;
```

Listing 17.4—Continued

```
procedure executive.setkeys(k : boolean);
begin
   keys := k;
end;

function executive.getkeys : boolean;
begin
   getkeys := keys;
end;

var
   e1 : ^executive;
   e2 : ^secretary;

begin
   new(e1, init('John Smith', 33, true));
   new(e2, init('Jane Doe', 28, 100));
   e1^.display;
   e2^.display;
   dispose(e1, done);
   dispose(e2, done);
end.
```

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

TEACH YOURSELF... PASCAL

SECOND EDITION • DISK INCLUDED

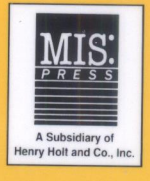
The Pascal language was originally designed as an aid for teaching computer programming. Consequently, it is an excellent language for the novice programmer. While well suited for the beginner, Pascal also provides more than enough capabilities for even the most advanced programmer. *teach yourself... Pascal* teaches the ins and outs of the Turbo Pascal programming language, by far the most dominant Pascal compiler in use. It covers all the basic features of Pascal: keywords, the structure of a program, procedures and functions, program flow, data types, arrays, records, and pointers. Once you understand and feel comfortable with these topics, the book guides you through the more advanced features: dynamic memory management, units, strings, console input/output, and file input /output. Finally, the book will show how Turbo Pascal can be used to perform object-oriented programming. Goodwin is known for his ability to explain important information in small, easy to understand steps. Each lesson is kept to just a few short pages, covering all the essentials. In this way, beginners won't feel overwhelmed by a flood of new information. By the end of Chapter 1 you will be writing your first simple program. Goodwin's expertise assures you that with a bit of patience, perseverance, and by writing programs, you will master all aspects of this language, and become an accomplished Pascal programmer.

Includes valuable information on:

- Predefined Data Types
- The Pascal Operators
- Program Flow
- Procedures and Functions
- User-Defined Data Types
- Arrays
- Records
- Pointers
- Object-Oriented Programming

MARK GOODWIN is a nationally noted author considered by many to be an expert in personal computer programming. He is an accountant, veteran programmer, and respected software reviewer. His works include: *Power of Visual Basic* (MIS:Press, 1991), *Graphical User Interfaces in C++* (MIS:Press, 1990), *User Interfaces in C++ and Object-Oriented Programming, User Interfaces in C,* and *Quick Basic Advanced Programming Tools* (all from MIS:Press, 1989), and the second edition of *teach yourself... Assembler* (MIS:Press, 1993).

ISBN 1-55828-328-5
US $29.95
CAN $38.00
90000>
9 781558 283282



Level

Beginner/Intermediate

Programming Language

IBM PC or Compatible