Microsoft® QuickPascal

Pascal by Example

Microsoft®

Microsoft® QuickPascal

PASCAL BY EXAMPLE

MICROSOFT CORPORATION

Printed and bound in the United States of America.

Document No. LN0104-500-R00-0489
Part No. 06396
10 9 8 7 6 5 4 3 2 1

Table of Contents Overview

Table of Contents

Introduction

PART 1 Pascal Basics

PART 2 Programming Topics

Appendixes

Figures

Tables

Introduction

Congratulations on your purchase of Microsoft ® QuickPascal. By now you should have read the other book in this package, *Up and Running*, and installed the software. And you're getting a feel for the many features of the product and the power that QuickPascal places at your fingertips. Now you're ready to use that power as you learn how to program in Pascal. QuickPascal makes learning easy with

- An integrated programming environment. You can write and edit code with the built-in editor, and then compile, run, and save your programs with easy-to-use menus.
- A powerful on-line help system, the QuickPascal Advisor. By pressing one key, the F1 key, you can access everything you need to know about the QuickPascal environment and about the Pascal language itself.
- An excellent debugger. You can find out exactly where a syntax error occurred, or trace through a program to find errors in logic.
- Object-oriented extensions. You can use QuickPascal to learn and use the newest concept in programming: object-oriented programming.

About This Book

This book, *Pascal by Example*, complements the on-line features of QuickPascal to teach you how to program in this language. In discussing programming terms and processes, this book refers frequently to example programs that are located on-line. Fragments of the example code are often quoted in text, thus teaching you as you read along. You can also load the whole example, so you can compile and experiment with working code.

As you read the book and run the example programs, you'll learn about these QuickPascal programming features:

- Programming basics such as variables, operators, and expressions
- Specific programming features such as strings, loops, units, procedures, data types, and input/output
- Powerful graphics capabilities and object-oriented programming

Pascal by Example makes certain assumptions about your knowledge. It does not explain programming concepts and terms in their simplest form. This book assumes you have programmed in other programming languages but are not familiar with Pascal. The following list summarizes the book's contents:

- Part 1, "Pascal Basics," covers basic Pascal language fundamentals such as procedures and data types. These chapters are designed to be read in order, from beginning to end.
- Part 2, "Programming Topics," covers practical programming topics such as using pointers and files. An advanced topics chapter discusses bitwise operators, calling assembly language routines, and similar problems.
- Part 3, "Graphics and Objects," covers QuickPascal graphics, which include graphics primitives and font support. The final chapter introduces you to the object extensions in QuickPascal. If you've never seen object-oriented programming before, this will get you started.
- Appendix A, "ASCII Character Codes and Extended Key Codes," provides valuable reference information on ASCII codes and the extended key codes returned by the **Crt** unit function **ReadKey**.
- Appendix B, "Compiler Directives," lists all QuickPascal directives.
- Appendix C, "Summary of Standard Units," briefly describes each unit provided with QuickPascal.
- Appendix D, "Quick Reference Guide," lists all QuickPascal keywords and the standard units' procedures and functions.

NOTE *Microsoft documentation uses the term "DOS" to refer to both the MS-DOS® and IBM Personal Computer DOS operating systems. The name of a specific operating system is used when it is necessary to note features that are unique to that system.*

Using the Example Programs

The QuickPascal Advisor includes all of the significant program examples in this book (although it doesn't include all of the code fragments). This feature allows you to load, run, and experiment with example programs as you read.

Every complete program in this book starts with a comment of this general form:

```
{ FTOC.PAS: converts temperatures }
```

The comment contains the program's name and a brief description of what it does. This program can be found in the QP Advisor as FTOC.PAS.

You must be using the QuickPascal environment to load an example. To load the example program, open the Help menu and choose the Contents command. Choose the title of this book from within the Contents screen. Then find the

desired program in the list of *Pascal by Example* programs, and copy it into the source window using the Copy and Paste commands from the Edit menu.

After you copy an example program into the QuickPascal source window, treat it as you would any Pascal source program. You can compile or edit the program, save it on disk, and so on.

Note that you can easily print out an example program, in whole or in part, from Help or from your file, by choosing the Print command from the File menu.

Key to Document Conventions

This book uses the following document conventions:

Example	Description
COPY TEST.OBJ C:	Uppercase type represents DOS commands and file names.
BEGIN	Boldface type (whether in all uppercase or in both upper- and lowercase letters) indicates standard features of the QuickPascal language: keywords, operators, and standard procedures and functions.

`PROGRAM FtoC;`
`CONST`
`  Factor = 32.0`
	This typeface is used for example programs, program fragments, and the names of user-defined procedures, functions and variables. It also indicates user input and screen output.

ArrayName	Words in italics indicate placeholders for information that you must supply. A file name is an example of this kind of information.
[[*Option*]]	Items inside double square brackets are optional.
{*Choice1* | *Choice2*}	Braces and a vertical bar indicate a choice among two or more items. You must choose one of the items unless all of the items are also enclosed in double square brackets.
Repeating elements...	Three dots following an item indicate that more items having the same form may be entered.

`REPEAT`
`.`
`.`
`.`
`UNTIL`
	A column of three dots indicates that part of the example program has intentionally been omitted.

F1	Small capital letters denote names of keys on the keyboard. A plus (+) indicates a combination of keys. For example, SHIFT+F5 tells you to hold down the SHIFT key while pressing the F5 key.
"array pointer"	The first time a new term is defined, it is enclosed in quotation marks. Since some knowledge of programming is assumed, common terms such as memory or branch are not defined.
American National Standards Institute (ANSI)	An acronym is spelled out the first time it appears.

Other Books on Pascal Programming

This book introduces the Pascal language and some practical programming topics. It does not attempt to teach you basic computer programming or advanced Pascal programming techniques. The following books cover a variety of topics that you may find useful. They are listed only for your convenience. With the exception of its own publications, Microsoft does not endorse these books or recommend them over others on the same subject.

Cooper, Doug. *Standard Pascal: User Reference Manual*. New York, NY: W.W. Norton & Company, 1983.

A complete and concise reference manual to standard Pascal.

Cooper, Doug and Michael Clancy. *Oh! Pascal!* 2d ed. New York, NY: W. W. Norton & Company, 1985.

A beginner's introduction to standard Pascal.

Craig, John Clark. *Microsoft QuickPascal Programmer's Toolbox*. Redmond, WA: Microsoft Press. In press.

A sourcebook that beginners will find useful for learning by example. Seasoned professionals using QuickPascal as a development system will find the routines to be valuable and immediately useful extensions to the Pascal language.

Grogono, Peter. *Programming in Pascal*. rev. ed. Menlo Park, CA: Addison-Wesley, 1980.

A good teaching source, but one that also covers sophisticated topics.

Jamsa, Kris. *Microsoft QuickPascal Programming*. Redmond, WA: Microsoft Press. In press.

A comprehensive introduction to mastering QuickPascal programming for the beginning- to intermediate- level programmer complete with hands-on examples.

Jamsa, Kris. *Microsoft QuickPascal: Programmer's Quick Reference*. Redmond, WA: Microsoft Press. In press.

This alphabetical reference provides concise descriptions of all QuickPascal procedures and functions.

Kernighan, Brian W., and P.J. Plauger. *Software Tools in Pascal*. Menlo Park, CA: Addison-Wesley, 1981.

A beginning- to intermediate-level guide to software tools and programming techniques.

Ladd, Robert Scott. *Learning Object-Oriented Programming with Microsoft QuickPascal*. Redmond, WA: Microsoft Press. In press.

Provides an example-rich introduction to the philosophy and procedures of object-oriented programming for the beginning- to intermediate-level QuickPascal user.

Leestma, Sanford, and Larry Nyhoff. *Pascal: Programming and Problem Solving*. New York, NY: Macmillan Publishing Company, 1987.

A beginner's guide to Pascal programming. The book also covers problem analysis, algorithm development, algorithm translation to Pascal, and program validation.

Wirth, Niklaus, and Kathleen Jensen. *Pascal User Manual and Report*. 3d ed. New York, NY: Springer-Verlag, 1985.

The definitive source.

Getting Assistance or Reporting Problems

If you feel you have discovered a problem in the software, please report the problem using the Product Assistance Request form at the back of this book.

If you have comments or suggestions reqarding any of the books accompanying this product, please use the Documentation Feedback Card, also at the back of this book.

PART 1

Pascal Basics

PART 1

Pascal Basics

Part 1 of *Pascal by Example* introduces you to the Pascal programming language and the basic elements of Pascal programs. The chapters in this part assume that you know some programming concepts but are not familiar with Pascal. Experienced Pascal programmers may want to skim these chapters.

The information in Part 1 helps you start writing Pascal programs immediately. Chapters progress through such major topics as procedures, program flow, user-defined data types, and units. If you're new to Pascal, reading the chapters in this part sequentially will give you a thorough introduction to the fundamentals of programming in Pascal.

CHAPTERS

CHAPTER 1

1

Your First Pascal Program

You're probably eager to use QuickPascal and begin writing programs in Pascal. In this chapter, you'll compile and run your first Pascal program, FTOC.PAS, which is shown in Figure 1.1. As with all of the complete programs in this book, we've done the work of typing it in for you. You just need to use the Copy and Paste commands on the Edit menu to copy the program from the QuickPascal Advisor into a blank source window. Then you can press F5 to compile and run the program.

This chapter uses the sample program to introduce the parts of a Pascal program, some terms you should know, and input and output. Most of these terms are discussed in more detail in Chapter 2.

If you're an experienced programmer in another structured language such as C, you might want to skip this chapter and begin with Chapter 2, "Programming Basics," or Chapter 3, "Procedures and Functions."

1.1 Sample Pascal Program

FTOC.PAS is a simple program that converts temperatures from Fahrenheit to Celsius. Like all of the sample programs in this book, you'll find it in the

QuickPascal Advisor. Figure 1.1 contains the FTOC.PAS program code and illustrates the program parts.

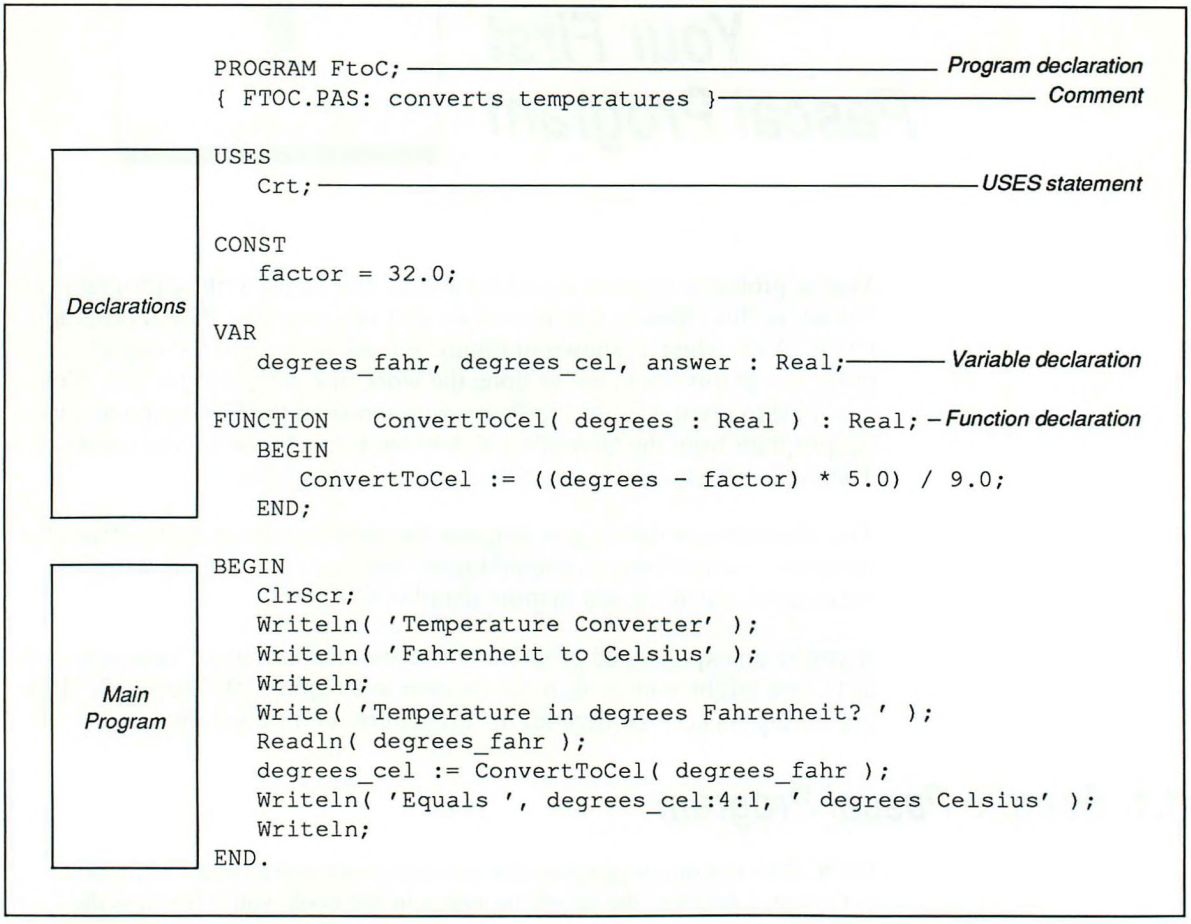

Figure 1.1 Parts of a Pascal Program

1.2 Parts of a Pascal Program

Pascal is a highly structured language. Programs have parts, and the parts have a required order. This section explains the parts of a Pascal program.

The Program Declaration

Traditionally, the first line of every Pascal program contains the program declaration. This declaration consists of the keyword **PROGRAM**, the name of the

program, and a semicolon. With Microsoft QuickPascal, such a declaration is not required, but it is still helpful as a way of labeling a program. Also, a program declaration is necessary if you want your program to compile on compilers other than those developed by Microsoft.

The USES Statement

The **USES** statement always follows the program declaration. It specifies which units are called from the program. Units are explained in detail in Chapter 7.

The **USES** statement in the FTOC.PAS program calls the **Crt** unit. This unit contains procedures and functions that allow you to control your computer screen.

Constant and Variable Declarations

Pascal requires that you declare constants, types, and variables before using them. The keywords **CONST**, **TYPE**, and **VAR** are used for this purpose. Constant declarations are made with the **CONST** keyword and include both the constant identifier (a name) and its value. For example, the constant shown in the FTOC.PAS program is named `factor` and has the value `32.0`.

The **VAR** keyword is used for variable declarations. Variable declarations include the variable identifier (a name) and its type. Multiple variables of the same type can be declared in the same statement.

Procedures and Functions

Procedure and function declarations come after the constant and variable declarations. Each procedure or function has its own **BEGIN...END** block. FTOC.PAS uses the function `converttocel` to convert the temperature from Fahrenheit to Celsius.

The Program Body

The body of the program, as shown in Figure 1.1, is enclosed by the keywords **BEGIN** and **END**. Pascal executes the instructions between **BEGIN** and **END**. The period after the **END** keyword tells Pascal that the program is over; anything that follows the period is ignored.

Comments

Comments are notes to yourself or others that make your programs easier to understand and maintain. Braces tell QuickPascal to ignore comment text. Instead of braces, comments can also be enclosed in parentheses and asterisks, as shown in this example:

```
(* This is a comment between the special symbols *)
```

The text on line 2 of FTOC.PAS is a comment.

1.3 Some Terms You Should Know

If you've never programmed before, the terms "keyword" and "identifier" may be confusing to you. These and other terms are discussed in this section. If you are an experienced programmer, you can skip this section and go on to Chapter 2, "Programming Basics."

1.3.1 Keywords

Pascal contains a set of "keywords," or words that it reserves for its own use. Whenever the Pascal compiler encounters a keyword, a specific action is performed. In the sample program FTOC.PAS, the words **PROGRAM**, **VAR**, **BEGIN**, and **END** are all keywords. A keyword always means the same thing to a program; for example, the keyword **BEGIN** always signals the beginning of a statement block, and the keyword **END** always signals the end of a statement block.

The QuickPascal environment gives you the option of displaying keywords in a special color. This makes your programs easier to read on screen. See *Up and Running* for more information on how to display in color.

Always remember that you cannot use a keyword as a variable name or for anything other than its intended purpose. Keywords are shown in all uppercase letters in this book, and in the sample programs. A list of QuickPascal keywords is shown below:

ABSOLUTE	**IF**	**RECORD**
AND	**IMPLEMENTATION**	**REPEAT**
ARRAY	**IN**	**SET**
BEGIN	**INHERITED**	**SHL**
CASE	**INLINE**	**SHR**
CONST	**INTERFACE**	**STRING**
CSTRING	**INTERRUPT**	**THEN**
DIV	**LABEL**	**TO**
DO	**MOD**	**TYPE**
DOWNTO	**NIL**	**UNIT**
ELSE	**NOT**	**UNTIL**
END	**OBJECT**	**USES**
EXTERNAL	**OF**	**VAR**
FILE	**OR**	**WHILE**
FOR	**OVERRIDE**	**WITH**
FORWARD	**PACKED**	**XOR**
FUNCTION	**PROCEDURE**	
GOTO	**PROGRAM**	

1.3.2 Identifiers

"Identifiers" are names that you use in your program. In the example program, the word `degrees_cel` is the identifier for a variable. You could call it `celsiusdegrees` or `cd` or `fred` if you wanted to; however, for the sake of good programming practice, the name should provide information about the purpose of the variable. Thus `degrees_cel` is preferable to `cd` or `fred`. QuickPascal requires that you follow three rules when creating identifiers:

1. The first character of an identifier must be a letter or underscore character (_).
2. Digits can be used within an identifier.
3. Identifier names can be of any length, but only the first 63 characters are significant.

Thus, the following are examples of valid identifiers:

```
FirstTime
first_time
index0
area233
area_555
A4_9RT4_98NNS
```

By contrast, the following are examples of invalid identifiers:

Identifier	Reason It's Invalid
`First Time`	There is a space between `First` and `Time`.
`index0$`	The `$` sign is not a valid identifier character.
`area!233`	The `!` mark is not a valid identifier character.
`555area`	The first character is a digit.

Identifiers in Pascal are not case sensitive. This means that the identifiers `First_Time`, `first_Time`, `first_time`, and `FIRST_TIME` all refer to the same identifier. If these identifiers were typed in a program, the compiler would not generate an error. Instead, the compiler would treat them as the same identifier.

Some identifiers are defined by Microsoft. These identifiers name certain data types, standard procedures and functions, units, and variables and constants defined in standard units. In this manual and throughout the sample programs, these "standard identifiers" are shown with initial capital letters.

1.3.3 Constants and Variables

"Constants" and "variables" are two different ways for your program to use data. A constant is defined at the beginning of your program and always has the same value. In the sample program FTOC.PAS, the constant `factor` is assigned the value `32.0`. After such an assignment is made (using the **CONST** keyword), QuickPascal replaces the identifier `factor` with the value `32.0` whenever the identifier is encountered. Constants can make your program more readable and understandable.

Programs operate on data. In Pascal, as in other programming languages, data are stored in variables. Variables must be declared in a Pascal program before being used. Variable declarations follow the keyword **VAR** and have two parts: a variable name (identifier) and type. Data types are discussed below. In our sample program, both `degrees_fahr` and `degrees_cel` are variables of type `Real`.

1.3.4 Data Types

Pascal classifies data according to "type." Roughly speaking, data types allow the computer to determine how much memory a variable requires, and what level of precision should be maintained for a number. Type declarations keep you from making mistakes such as storing your zip code where your salary should be.

When you declare a variable, you must declare its type as well as its name. Data naturally falls into two general categories: text and numbers. Textual data items (which may include numbers, as in a street address) are referred to as "strings." Numbers can be integers (whole numbers such as 1 and 34) or real numbers (decimal numbers such as 29.5 and 3.14159). Four simple data types are illustrated below:

Data	Type
`Mary Smith`	String
`1413 Oak Lane`	String
`76`	Integer
`21.0987`	Real
`Y`	Character

Data types are discussed in more detail in Chapter 2, "Programming Basics."

1.3.5 Operators and Expressions

Operators are used in expressions to manipulate data within your program. Pascal includes many operators, but the simplest are already familiar to you: addition, subtraction, multiplication, and division. Another class of common operators are relational operators, such as "greater than," "less than," and "equal to." You'll use operators in your programs to do tasks such as counting how many lines you've printed on a page, or seeing if the number of hours someone worked in a week exceeds 40. More sophisticated uses for operators also exist, and those are discussed in Chapter 2, "Programming Basics."

Expressions combine operators with data, as shown in the example below:

```
{ Multiply the number of hours worked by the rate of
  pay. Store the answer in the variable 'Salary'
}
salary := hours * pay_rate;
```

1.4 Input and Output

At the risk of oversimplification, you could say that most programs get some data, manipulate that data in some fashion, and display some final data to a user. So far in this chapter you've learned a bit about data and data manipulation. "Input" and "output" refer to the processes of getting and displaying data.

Input

Pascal programs frequently make use of the **Read** and **Readln** procedures for data input. **Read** is useful for getting a single keystroke—for instance, when you want the user to press a key to stop a process. **Readln** is useful when you want the user to be able to type and correct a complete line of text before the program accepts it. **Readln** doesn't do anything until the user presses ENTER, thus providing the user with the ability to edit the input line before sending it on to the program for processing.

In the example below, the first line that the user types is stored as the variable `name` and the next line is stored as `address`.

```
Readln( name );
Readln( address );
```

Output

Just as **Read** and **Readln** handle data input, **Write** and **Writeln** handle output. **Writeln** differs from **Write** in that **Writeln** generates a carriage return at the end of the string (the line of text). Thus, **Writeln** moves the cursor to a new line on your output screen.

1.5 Moving On

So far you have looked at and compiled your first program, and you have been exposed to some basic Pascal terms. The next chapter elaborates on the concepts introduced here and shows you how to create some more complex programs.

CHAPTER 2

Programming Basics

2

This chapter introduces you to some Pascal programming fundamentals, such as data types, constants, variables, operators, and expressions. If you're already an experienced programmer in another language, you may find most of this material to be familiar. In that case, you might want to skim this chapter quickly or just skip ahead to Chapter 3.

2.1 Data Types

All data in your program is either a constant or a variable; each has an associated data type. Two kinds of data types exist in QuickPascal: predefined data types and user-defined data types. Predefined data types, such as **Real** and **STRING**, are a built-in part of the language and are discussed below. User-defined data types expand your programming power considerably. They are complex enough that Chapter 5 of this book is devoted to them.

The predefined data types supported by QuickPascal and explained in the following sections are:

- Integers
- Floating-point numbers
- Characters
- Strings
- Booleans

2.1.1 Integer Types

Integers are whole numbers like the numbers you use to count with; that is, they have no fractional parts. The number 12 is an integer; 12.0 and 12.5 are not. Table 2.1 lists the five integer types that QuickPascal supports. Programs discussed later in this chapter show how these integers are used.

Table 2.1 Integer Data Types

Integer Type	Range of Values	Byte Size	Examples
ShortInt	-128 to 127	1	-7, 55, 123, 0, $F
Byte	0 to 255	1	55, 123, $F, 0
Integer	-32768 to 32767	2	-555, 30000, 0, $FF
Word	0 to 65535	2	30000, 60000, $FFFF
LongInt	-2147483648 to 2147483647	4	-100000, 100000, $FFFF

The integer types **Byte** and **Word** are called "unsigned" integers, while the other integer types are called "signed" integers. The word "unsigned" indicates that the integer includes values from zero to the upper positive limit. Signed integers include negative numbers.

You can see that the main difference between the different integer types is the range of the values that can be stored. In the **VAR** section of your program, you declare the variable identifier and the specific type.

As the examples in Table 2.1 show, QuickPascal allows you to write integers in either decimal (base 10) or hexadecimal notation (base 16). Hexadecimal numbers begin with the dollar sign ($) and use the characters 0–9 and A–F.

The INTTYPES.PAS program below demonstrates different types of integers.

```
PROGRAM Inttypes;
{ INTTYPES.PAS demonstrates integer data types. }

USES
    Crt;

VAR
    short_int_val     :   ShortInt;
    byte_val          :   Byte;
    integer_val       :   Integer;
    word_val          :   Word;
    long_int_val      :   LongInt;
```

```
BEGIN
    short_int_val := -31;
    byte_val      := 255;
    integer_val   := -21212;
    word_val      := $FFFF;
    long_int_val  := 12918656;
    ClrScr;
    Writeln( 'short_int_val = ', short_int_val );
    Writeln( 'byte_val      = ', byte_val );
    Writeln( 'integer_val   = ', integer_val );
    Writeln( 'word_val      = ', word_val );
    Writeln( 'long_int_val  = ', long_int_val );
END.
```

Here is the output from INTTYPES.PAS:

```
short_int_val   = -31
byte_val        = 255
integer_val     = -21212
word_val        = 65535
long_int_val    = 12918656
```

2.1.2 Floating-Point Types

Floating-point numbers are not integers; that is, they are written with a decimal point. Thus the number 12.5 is a floating-point number, as is .00021459862. Also, QuickPascal treats constants larger than its maximum as long-integer size floating-point numbers, even if they do not contain a decimal point. Floating-point numbers are also referred to as "real" numbers.

QuickPascal supports a group of floating-point types that vary in their precision, range of values, and storage requirements. They are shown in Table 2.2.

Table 2.2 Floating-Point Data Types

Type	Range of Value	Byte Size	Significant Digits
Single	1.5E-45 to 3.4E+38	4	7-8
Real	2.9E-39 to 1.7E+38	6	11-12
Double	5.0E-324 to 1.7E+308	8	15-16
Extended	3.4E-4951 to 1.1E+4932	10	15-16
Comp	–9.2E+18 to 9.2E+18	8	15-16

By default, QuickPascal assumes that your computer does not use a math coprocessor, resulting in fewer significant digits than if it did use a coprocessor. However, if you have a math coprocessor available (any member of the 8087 family of processors), you can increase the precision of **Comp** and **Extended** data types by 4 significant digits by including the {$N+} compiler directive in your program.

Unlike some other Pascal compilers, QuickPascal makes all real number types available for your use, whether or not your computer has a numeric coprocessor.

The **Comp** type is a little different from the other floating-point types. **Comp** is designed to count very large numbers, and so it stores only the integer part of a number between –(2^63)+1 and (2^63)–1.

2.1.3 Character Type

The character data type **Char** stores only one character. Each character occupies one byte of storage. You can represent characters using the following formats:

Characters	Representation
Readable characters (that is, the alphabet, digits, and punctuation characters)	Can be represented by using same letters, digits, and punctuation.
Control characters (ASCII characters 0 through 31)	Can be represented using the carat symbol (^) followed by the control letter. For example, the form feed is represented by ^L, since form feed is ASCII 12 and L is the 12th letter in the alphabet. Appendix A contains a chart of all ASCII codes.
All characters, including those in the extended ASCII table	Can be represented by using the number sign (#) followed by the ASCII code number. Thus, #65 is the letter A, #12 is the form feed, and so on.

2.1.4 String Types

This data type stores a string of characters such as a name or an address. QuickPascal strings can hold up to 255 characters. Many of the examples in this book use strings to display messages and store input. The STRINGS.PAS program demonstrates how to create, read, and write simple strings.

```
PROGRAM Strings;

{ STRINGS.PAS demonstrates basic string operations. }

USES
    Crt;

CONST
    str_constant = 'Type something and press Enter: ';

VAR
    prompt_one : STRING;
    prompt_two : STRING;
    input_str  : STRING;

BEGIN
    prompt_one := str_constant;
    prompt_two := 'You typed: ';

    ClrScr;
    Write( prompt_one );
    Readln( input_str );
    Write( prompt_two );
    Writeln( input_str );
END.
```

Here is typical output from STRINGS.PAS:

```
Type something and press Enter: QuickPascal!
You typed: QuickPascal!
```

2.1.4.1 Declaring Strings

Pascal strings are usually of the **STRING** type. In STRINGS.PAS, the statements

```
VAR
    prompt_one : STRING;
    prompt_two : STRING;
    input_str  : STRING;
```

declare three string variables named `prompt_one`, `prompt_two`, and `input_str`.

2.1.4.2 Initializing Strings

You can initialize string variables by assigning string literals or constants to them:

```
CONST
    str_constant = 'Type something and press Enter: ';
    .
    .
    .
    prompt_one := str_constant;  { Assign string constant }
    prompt_two := 'You typed: '; { Assign string literal }
```

Pascal strings can contain as many as 255 characters. The first character of a string is a "length byte" that indicates the number of characters in the string. Procedures such as **Writeln** look at this byte to determine the string's size.

Thus, if you assign `Hello` to a string variable, the variable actually contains six characters: a length byte that contains the number 5, followed by the 5 characters of `Hello`.

2.1.4.3 Reading and Writing Strings

You can use the **Read** and **Readln** procedures to read a string:

```
Readln( input_str );
```

and the **Write** and **Writeln** procedures to write a string:

```
Write( prompt_two );
Writeln( input_str );
```

These procedures read input from the keyboard and write output to the screen. See Chapter 8, "The Keyboard and Screen," for more information on input and output.

2.1.5 Boolean Type

A variable of type **Boolean** can be assigned a value of true or false. Variables of this type are often used as "flags" when a condition in your program becomes true (or false). Suppose, for example, that you wanted to check whether a user had finished entering data. If the variable `all_done` had been declared as type **Boolean**, you may use it like this:

```
IF (x = 0) THEN all_done := true;
```

Later you may need to check the `all_done` variable:

```
IF (all_done = True) THEN ...
```

A shorthand way of writing this statement is

```
IF (all_done) THEN ...
```

It is understood that `IF all_done` means the same as `IF (all_done = True)` and by the same understanding, `IF (NOT all_done)` means `IF (all_done = False)`.

For more information about **Boolean** types and what you can do with them, see Chapter 12, "Advanced Topics."

2.2 Constants

Fixed data, or "constants," are assigned a value which is never changed (an exception to this, called "typed constants," is discussed below). When you declare constants in Pascal, you assign them a name and value. This value, once assigned, cannot be changed in your program. The assignment of name and value to a constant is called a constant declaration.

QuickPascal supports two kinds of constants: simple and typed.

2.2.1 Simple Constants

Simple constants can be used for two purposes. First, they can store fixed values, like the number of inches in a foot. Second, they can store values you wish to use within your program, like the text of a title screen or your assumption for the rate of inflation. Since constants are all declared at the beginning of your program, it is easy to change any of those values and recompile your program. Remember, changing the value of a constant in its initial declaration gives that new value to the constant wherever it is used. In fact, any other constants that depend on the one you alter will also receive their updated values when you recompile the program.

The general syntax for declaring and initializing a constant is:

CONST
ConstantIdentifier = {*ConstantValue*|*Expression*}

For example:

```
CONST

    ft_per_mile = 5280;
```

In this example, the *ConstantIdentifier* is `ft_per_mile` and the *ConstantValue* is `5280`.

In QuickPascal, you can use an expression that contains a previously declared constant. You can also set a constant equal to an expression, as in

```
CONST

    days_per_year = 365;
    seconds_per_day = 60 * 60 * 24;
    light_speed = 186282;
    light_year = light_speed * seconds_per_day
                 * days_per_year;
```

However, if you declare a constant equal to an expression, that expression can only contain simple operations like addition, subtraction, multiplication, and division. More advanced Pascal functions like square root cannot be used.

Here are some more examples of constants:

```
CONST

    max_row = 10;
    max_col = 10;

  { Uses previously declared constants  }
    table_size = max_row * max_col;

    prompt = 'Press any key to resume...';
    byebye = 'Thank you for using the program...';

    euler_const = 0.577215664901;

  { An approximation of the Euler constant }
    euler_const2 = 228.0 / 395.0;
```

2.2.2 Typed Constants

QuickPascal lets you use another kind of constant called a "typed constant." In standard Pascal, you can't specify the data type of a constant. The compiler assigns the data a type according to how it is presented. For example, a constant called `number_of_people` with a value of 35 would be assigned an integer type by QuickPascal, which makes sense. However, if you declared `number_of_people` equal to 35.0, then the compiler would assign a real number type to `number_of_people`, which doesn't make much sense.

Fortunately, QuickPascal allows you to specifically state the data type associated with a constant, which is the typed constant. You may also see the terms "variable constant" or "static variable" used to describe such a constant. Declaring

data with a typed constant actually changes the constant into an initialized variable. That is, the value associated with the typed-constant identifier can be changed in the program, unlike regular constants whose value cannot be changed within the program.

A typed constant can be thought of as a cross between a constant and a variable. A constant's value is declared but can never be changed. A variable, on the other hand, is not declared with a predefined value, and its value can change within the program. A typed constant is like a variable whose predefined value is declared but which can be redefined within the program.

The general syntax is shown below:

CONST
ConstantIdentifier : *TypeName* = {*ConstantValue* | *Expression*}

Examples of typed constant declarations are given below. Note that the *ConstantValue* or *Expression* assigned to the typed constant is much like a value or expression assigned to a regular variable. Note that you use the :*TypeName* to specify the type of the constant, as you would in a **VAR** declaration.

```
CONST

    max_row : Word = 10;
    max_col : Word = 10;
    table_size : Word = max_row * max_col;

    prompt : STRING = 'Press any key to resume...';
    byebye : STRING = 'Thank you for using the program...';

    euler_const : Real = 0.577215664901;
    { An approximation to the Euler const }
    euler_const2 : Real = 228.0 / 395.0;
```

2.3 Simple Variables

While constants are identifiers that are declared and assigned a fixed value, variables are identifiers that are merely declared. They are assigned values in the main program or a subprogram. The general syntax for declaring a variable and identifying its type is as follows:

VAR
VariableName [[, *VariableName*...]] : *DataType*
.
.
.

Some examples of declaring variables:

```
VAR
    { Variables declared with predefined type }
    column_index : Word;
    area, circumference : Real;
    message : STRING;
    input_character : Char;
```

Once a variable is declared, QuickPascal sets aside the required amount of memory to store data associated with the variable. The amount of memory set aside is based on the variable's data type. For example, a variable declared with type `STRING[40]` gets 41 bytes of memory of which 40 bytes are available for data (the remaining byte holds the string length); a variable with type `Integer` gets 2 bytes of memory.

Once a variable is declared, you may assign it a value. Often you will use the assignment operator (:=), which should not be confused with the equality operator (=). The assigned value can be a constant, another variable, or an expression.

The VARS.PAS program shows a simple assignment of variables. The variable radius is assigned a value through user input (the **Readln** procedure). The variable area is assigned the value of an expression using the assignment operator.

```
PROGRAM Vars;
    { VARS.PAS: variable declaration and use }

VAR
    radius, area   :   Real;

BEGIN
    Writeln( 'Enter radius' );
    Readln( radius );
    Area := (Pi * (radius * radius));
    Writeln( 'Area of the circle = ', area:8:2 );
    Writeln;
END.
```

Typical output from VARS.PAS looks like this:

```
Enter radius
12
Area of the circle =   452.39
```

2.4 Pascal Operators

Operators let you manipulate data. Pascal operators are used to build expressions. This section describes some of the operators available in Microsoft QuickPascal. In addition to the ones discussed here, QuickPascal supports

some operators for advanced use, including bitwise and set operators. The bitwise operators are discussed in detail in Chapter 12, "Advanced Topics." Set operators are discussed in Chapter 5, "User-Defined Data Types."

2.4.1 Kinds of Operators

QuickPascal supports the following categories of operators:

- Arithmetic
- Relational
- String
- Address-of

2.4.1.1 Arithmetic Operators

Table 2.3 lists the arithmetic operators, which perform numeric manipulation of integer and floating-point types. While the addition (+), subtraction (–), and multiplication (*) operators work with both integers and floating-point types, the division (/) operator performs floating-point division (even if the operands are integers), and the **DIV** and **MOD** operators work with integers only.

While most of the operators take two operands, the unary plus and unary minus operators work with one operand only.

Table 2.3 Arithmetic Operators

Operator	Purpose
+	Unary plus sign
–	Unary minus sign
+	Adds two numbers
–	Subtracts two numbers
*	Multiplies two numbers
/	Divides two floating-point numbers
DIV	Divides two integer numbers
MOD	Returns the remainder of integer division

2.4.1.2 Relational Operators

Relational operators are used to compare two operands, which may be constants, variables, functions, or expressions. The operands must be of the same or compatible types. Some useful relational operators are shown in Table 2.4.

Table 2.4 Relational Operators

Operator	Purpose
=	Equal to
< >	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

2.4.1.3 String Operators

QuickPascal has one string operator, the (+) operator, for string and character concatenation. Other aspects of string manipulation are performed by predefined procedures and functions.

2.4.1.4 Address-Of Operator

The address-of operator (@) is used to return the address of variables, routines, parameters, and so forth. The topic of pointers is covered fully in Chapter 11.

2.4.2 Operator Precedence

Operators in expressions are evaluated in order of their precedence. Table 2.5 lists the QuickPascal operators according to their precedence level. Although not all of the operators shown in the table are discussed in this section, they are in the table for the sake of completeness. For information on operators such as SHL or XOR, see Chapter 12, "Advanced Topics," or use QP Advisor.

Table 2.5 Operator Precedence

Precedence Level	Operators	Operator Class
1 (highest)	**@, NOT**	Unary
2	***, /, DIV, MOD, AND, SHL, SHR**	Multiplying
3	**+, –, OR, XOR**	Adding
4 (lowest)	**=, < >, <, <=, >, >=, IN**	Relational

The following list interprets this table and gives examples of the use of operators.

- The operation defined by the operator with highest precedence and its related operand is performed first. In the example

  ```
  2 * 4 + 3
  ```

  the number `4` is placed between two operators of different precedence. Since the (*) operator has a higher precedence, multiplication of `2` and `4` proceeds first. The addition is performed afterward.

- Operations of the same precedence level are performed from left to right. In the example

  ```
  2 * 4 / 3
  ```

  the number `4` is placed between two operators of the same precedence. Since the (*) operator appears to the left of number `4`, it is applied before the (/) operator.

- Parentheses are used to group operations. Expressions enclosed in parentheses are evaluated first. The most deeply nested expression is evaluated before any other. (Nested expressions are expressions within expressions.) Parentheses alter the "effective" or "working" precedence of an operator. In the example

  ```
  2 / ((3 + 4) * 5)
  ```

  the expression in the parentheses `(3 + 4)` is evaluated first, since it is the most nested expression in parentheses. The (*) operator is applied next, since it is enclosed in parentheses, giving it a higher effect precedence. Finally, the (/) operator is applied.

2.5 Simple Pascal Expressions

Expressions are a special and very important part of any programming language. They are created to evaluate data in your program. Expressions use the operators described in Section 2.4 and operands, which are the data types in your program, as their components. QuickPascal supports simple and advanced expressions.

This chapter briefly presents the simple expressions. Each type of expression follows a syntax rule which gives its components and syntax. The simple expressions explained in this chapter are

- Arithmetic
- String

2.5.1 Arithmetic Expressions

Arithmetic expressions combine constants, numbers, variables, and functions with arithmetic operators.

The syntax of an expression is

{ *Constant* | *Expression* } |

[[*Constant* | *Expression*]] { *Operator Constant* | *Expression* }

For example

```
(length * width)
```

Expressions combine with constants or variables and relational or assignment operators to make Pascal statements:

```
area := (length * width)
```

In the example above, the expression `length * width` is evaluated, then the result is stored in the variable `area`.

Expressions are evaluated according to operator precedence. Using parentheses to force the correct evaluation of an expression is always a good idea.

An expression may contain nested expressions, which are expressions within expressions. Examples are shown below:

```
K := (1 + 6) DIV (55 - J);
Z := (2 * X) + (Y / 4);
T := (((2 * X + 2) * X - 5) * X + 1) * X - 10;
```

2.5.2 String Expressions

String expressions use the (+) operator to concatenate strings and characters. The general syntax rule is shown below:

{*Character* | *String*} + {*Character* | *String*}...

For example

```
'dog' + 'house';    { creates the word "doghouse" }
```

The (+) operator works identically to the **Concat** function. String and character concatenation cannot build a string longer than 255 characters.

CHAPTER 3

3 Procedures and Functions

Procedures and functions allow you to write well-organized Pascal programs, in which discrete tasks are done in separate, logically contained modules. Once you understand procedures and functions, you are well on your way to becoming a true Pascal programmer.

This chapter begins by discussing procedures, which are simpler and more commonly used than functions. The discussion covers many topics, such as argument passing, that are common to both procedures and functions. The chapter ends with an explanation of functions, nested procedures, and recursion.

3.1 Overview

Procedures and functions let you program using a "divide and conquer" approach. Instead of trying to solve every aspect of a large problem at once, you divide it into several small problems and solve each one separately. This strategy allows you to write clear, reliable programs that perform different tasks in distinct, logically contained modules. In Pascal, these modules are called procedures or functions.

Dividing a program into task-based modules offers several advantages:

- Makes programs easier to write and read. All of the statements related to a task are located in one place.
- Prevents unexpected side effects because you can use private ("local") variables that are not visible to the main program or other sections.
- Eliminates unnecessary repetition of code for frequently performed tasks.
- Simplifies debugging. Once you have debugged a procedure or function, you can use it with confidence in many different situations.

The distinction between procedures and functions can be summarized in a few words. A procedure performs a specific task; a function performs a specific task and also returns a value. We will return to this topic in Section 3.3, "Functions."

If you are familiar with Microsoft QuickBASIC, you will notice many similarities in Pascal. A Pascal procedure resembles a QuickBASIC **SUB** procedure, and a Pascal function is like a QuickBASIC **FUNCTION** procedure. If you know the C language, you will notice that a C function combines the qualities of Pascal procedures and functions; a C function can return a value, or return nothing.

3.2 Procedures

Procedures and functions are very similar—so similar, in fact, that everything explained in this section applies to functions as well as procedures. To avoid repeating the phrase "procedures and functions" with every sentence, this section refers only to procedures. You should read it with the understanding that these ideas also apply to functions. Section 3.3, "Functions," explains how functions differ from procedures.

A "procedure" is a collection of declarations and statements that performs a certain task. You have already seen a few of the QuickPascal standard procedures, such as **Writeln**, which writes a line. This section explains procedures using several programs. The first example, CENTER.PAS, contains a procedure that centers a line on the screen:

```
PROGRAM center;

{ CENTER.PAS: Demonstrate simple procedure }

USES
    Crt;

VAR
    row   : Byte;
    title : STRING;

PROCEDURE center_line( message : STRING; line : Byte );
    BEGIN
        GotoXY( 40 - Length( message ) DIV 2, line );
        Writeln( message );
    END;

BEGIN
    row   := 2;
    title := 'Each line of text is centered.';
    ClrScr;
    center_line( title, row );
    center_line( '--------', row+1 );
    center_line( 'Microsoft QuickPascal!', row+2 );
END.
```

The CENTER.PAS program displays these lines on the screen:

```
Each line of text is centered.
            --------
     Microsoft QuickPascal!
```

The rest of this section refers frequently to the CENTER.PAS example.

3.2.1 Calling Procedures

The CENTER.PAS program uses a procedure named `center_line` to center a line. You "call," or execute, a procedure by stating its name and by supplying any "arguments," or data items that it might require. The `center_line` procedure expects you to supply two arguments: a piece of text to print, and the screen line on which to print it. To print the message

```
Vite!
```

on the second screen line, you would call `center_line` with this statement:

```
center_line( 'Vite!', 2 );
```

You list the arguments in parentheses after the procedure name, placing a comma between each two arguments. Here, the first argument is the string `Vite!` and the second is the number `2`. Later in this chapter, you will learn how a procedure handles the arguments it receives.

3.2.2 Declaring Procedures

A procedure "declaration" contains the complete code for the procedure. Here is the procedure declaration for `center_line`:

```
PROCEDURE center_line( message : STRING; line : Byte );
    BEGIN
        GotoXY( 40 - Length( message ) DIV 2, line );
        Write( message );
    END;
```

A procedure declaration has two parts, called the "head" and the "body." A procedure head consists of the **PROCEDURE** keyword, followed by the procedure's name and a list of its arguments in parentheses. It ends with a semicolon. Here is the head of the `center_line` procedure:

```
PROCEDURE center_line( message : STRING; line : Byte );
```

The list of arguments states the name and type of each argument. This example states that `center_line` requires two arguments, one each of the data types **STRING** and **BYTE**. The first argument is named `message`, and the second argument is named `line`.

The procedure body is a statement block that contains the procedure's executable statements. Like other blocks, the procedure body is enclosed in **BEGIN** and **END**, and it ends with a semicolon. Here is the body of the `center_line` procedure:

```
BEGIN
    GotoXY( 40 - Length( message ) DIV 2, line );
    Write( message );
END;
```

The body of the `center_line` procedure contains two statements, each of which calls a standard Pascal procedure. The first calls the **GotoXY** procedure and the second calls **Write**.

Forward Declarations

Pascal requires that you declare an entity before using it. Before using a variable, for instance, you must declare its name and type. The same rule applies to a procedure. Before calling a procedure, you must declare it as explained in the previous section.

In simple programs, such as CENTER.PAS, it's easy to satisfy the "declare before calling" rule. Simply place all of your procedure declarations before the main program body. In CENTER.PAS, the `center_line` procedure declaration appears before the main program.

Occasionally, however, you may need to call a procedure before it has been declared. This can be done by providing a "forward declaration" of the procedure prior to the procedure call.

A forward declaration is identical to a procedure head, except that it contains both the keyword **FORWARD** and a semicolon after the argument list. For instance, the `center_line` procedure head looks like this

```
PROCEDURE center_line( message : STRING; line : Byte );
```

and its forward declaration looks like this

```
PROCEDURE center_line( message : STRING; line : Byte );
FORWARD;
```

The forward declaration must appear before the first reference to the procedure. Most programmers put forward references at or very near the beginning of the program.

3.2.3 Declaring Local Variables

In addition to statements, the procedure body can contain variable declarations. Variables declared in a procedure are said to be "local," meaning they can be seen only inside the procedure. Because their visibility is limited, local variables are less likely to be changed accidentally than global variables.

The LOCAL.PAS program demonstrates local variables. It prompts you to enter a number and then displays the factorial of that number. (A factorial is the product of all the integers from 1 to a number. For instance, the factorial of 4 is 24, the product of 1 * 2 * 3 * 4.)

```
PROGRAM local_variables;

{ LOCAL.PAS: Demonstrate local variables. }

VAR
    num : Byte;

PROCEDURE factor( value : Byte );
    VAR
        factorial : Real;
        count : Byte;
    BEGIN
        factorial := 1.0;
        FOR count := 1 TO value DO
        factorial := factorial * count;
        Write( 'Factorial of ', value, ' is ' );
        Writeln( factorial );
    END; { procedure factor }

BEGIN { main program }
    Write( 'Enter a number smaller than 34: ' );
    Readln( num );
    Factor( num );
END.
```

Here is typical output from LOCAL.PAS:

```
Enter a number smaller than 34: 5
Factorial of 5 is 1.20000000000000E+0002
```

The `factor` procedure in LOCAL.PAS appears below. It declares two local variables named `factorial` and `count`:

```
PROCEDURE factor( value : Byte );
    VAR
        factorial : Real;
        count : Byte;
    BEGIN
        factorial := 1.0;
        FOR count := 1 TO value DO
            factorial := factorial * count;
        Write( 'Factorial of ', value,' is ' );
        Writeln( factorial );
    END; { procedure factor }
```

Notice where local variables are declared: between the procedure's head and the **BEGIN** keyword.

Variable Scope

Unlike global variables, which are declared outside any procedure and are therefore visible everywhere in the program, local variables are declared inside a procedure and are hidden from the rest of the program. If you refer to a local variable outside the "scope," or range, where it is visible, QuickPascal issues an error message:

```
Error P0032: Unknown identifier
```

The same message appears if you refer to a variable that has never been declared. In both cases, it means the variable is not visible—and cannot be used—in the place where the reference appears.

The ability to limit a variable's visibility makes it easier to write reliable programs. If a variable is local, it can't be changed accidentally by some other part of the program. Such haphazard side effects are common in older interpreted BASIC programs, in which all variables are global.

3.2.4 Passing Arguments

The `Factor` procedure in LOCAL.PAS has another local variable that hasn't been mentioned yet. In addition to `factorial` and `count`, which it declares, the procedure uses a third variable named `value`:

```
PROCEDURE Factor( value: Byte );
    VAR
        factorial : Real;
        count : Byte;
    BEGIN
        factorial := 1.0;
        FOR count := 1 TO value DO
            factorial := factorial * count;
        Write( 'Factorial of ', value,' is ' )
        Writeln( factorial );
    END; {procedure factor }
```

The `value` argument, in the procedure head, is "passed" when you call the `factor` procedure. The argument comes from a number you type in at the keyboard. The main procedure in LOCAL.PAS stores your input in a variable named `num`:

```
Writeln( 'Enter a number smaller than 34: ' );
Readln( num );
```

The main program passes the value of `num` as an argument when it calls the `factor` procedure:

```
factor( num );
```

When you list an argument in the procedure head, it becomes a local variable in the procedure. Thus, the head of the `factor` procedure

```
PROCEDURE Factor( value : Byte );
```

creates a local variable named `value`. Inside the `factor` procedure, `value` can be treated like any other local variable.

3.2.4.1 Passing by Value

The type of argument passing in LOCAL.PAS is called "passing by value" because the procedure receives *the value of* the original variable, not the variable

itself. The BYVALUE.PAS program demonstrates this idea, which has important consequences for managing variables:

```
PROGRAM byvalue;

{ BYVALUE.PAS: Demonstrate passing by value. }

VAR
    global_var : Integer;

PROCEDURE proc( local_var : Integer );
    BEGIN
        Writeln( 'local_var = ', local_var );
        local_var := 333;
        Writeln( 'local_var = ', local_var );
    END; { procedure proc }

BEGIN { main program }
    global_var := 5;
    proc( global_var );
    Writeln( 'global_var = ', global_var );
END.
```

Here is the output from BYVALUE.PAS:

```
local_var  = 5
local_var  = 333
global_var = 5
```

The program declares a global variable named `global_var` and assigns the value `5` to `global_var`. The `proc` procedure expects you to pass one argument, which it names `local_var`. The procedure prints the value of `local_var` (initially, `5`), then changes its value to `333` and prints it again. After control returns to the main program, BYVALUE.PAS prints the value of `global_var`, which remains at `5`.

The `proc` procedure alters the value of `local_var`. But this change has no effect on the original variable, `global_var`, which is not affected by anything that happens in `proc`. The same is true even if both variables have the same name (if you name both of them `my_val`, for instance).

Passing an argument by value creates a local copy of the variable in the procedure. Because the procedure receives only a local copy, it can give the argument any name, and change its value, without affecting variables outside the procedure.

3.2.4.2 Passing by Reference

Sometimes, you may want a procedure to change the value of an argument. For instance, say you need a procedure that swaps two variables. If you pass the variables by value, their values change inside the swap procedure, but remain unchanged in the rest of the program. You need a way to tell the procedure to change the original variables, not its local copies of them.

Pascal offers a second passing method, called "passing by reference," for just such cases. The BYREF.PAS program demonstrates this method:

```
PROGRAM byref;

{ BYREF.PAS: Demonstrate passing by reference. }

VAR
    var1, var2 : Integer;

PROCEDURE swap_vars(VAR var1 : Integer; VAR var2 : Integer);
    VAR
        temp : Integer;
    BEGIN
        temp := var1;
        var1 := var2;
        var2 := temp;
    END; { procedure swap_vars }

BEGIN
    var1 := 55;
    var2 := 99;
    Writeln( 'var1 = ', var1, '  var2 = ', var2 );
    swap_vars( var1, var2 );
    Writeln( 'var1 = ', var1, '  var2 = ', var2 );
END.
```

Here is the output from BYREF.PAS:

```
var1 = 55  var2 = 99
var1 = 99  var2 = 55
```

The program declares two global variables named `var1` and `var2`, assigning them the values `55` and `99`, respectively. It prints their values, calls the `swap_vars` procedure, then prints their values again. The output proves that `swap_vars` changes the original variables.

The important difference between this program and the previous example appears in the `swap_vars` procedure head:

```
PROCEDURE swap_vars(VAR var1 : Integer; VAR var2: Integer);
```

Notice the **VAR** keyword in front of each name in the argument list. It tells the `swap_vars` procedure to treat the argument as a variable (located elsewhere) rather than as a value. Instead of creating a local copy of the passed value, the procedure acts upon the variable itself.

The `swap_vars` procedure head happens to use the same names for these arguments (`var1` and `var2`) in its argument list. But the result would be the same if `swap_vars` used different names. Because the arguments are declared with **VAR**, their names in `swap_vars` are synonyms for the original variables.

You can underscore this point by making a simple change to the previous example, BYVALUE.PAS. Load the program and add **VAR** to the `proc` procedure head:

```
PROCEDURE proc( VAR local_var : Integer );
```

After you make this change, the `proc` procedure changes the global variable `global_var`, giving this output:

```
local_var = 5
local_var = 333
global_var = 333
```

In the original BYVALUE.PAS program, the global variable `global_var` retained the value `5` even though `proc` changed the value of `local_var`. Passing `global_var` by reference gives `proc` the ability to modify `global_var`.

When you pass an argument by reference, you must pass a variable, not a value. That is, the argument must be a variable name,

```
swap_vars( global_1, global_2 ); { correct }
```

not a constant,

```
swap_vars( 55, 99 ); { error! }
```

or an expression:

```
swap_vars( 5 * 11, 93 + 6 ); { error! }
```

It's best to pass arguments by reference only when you want the procedure to change the argument. Unnecessary passing by reference creates the same problems as the overuse of global variables.

3.3 Functions

A function is a procedure that returns a value. Like most languages, Pascal has many standard functions, such as **Sqrt**, which returns a square root.

You can think of a function as a special kind of procedure. All of the concepts explained in Section 3.2, "Procedures," also apply to functions. Rather than restate everything that procedures and functions share in common, this section explains the features that make functions different from procedures.

The FUNCT.PAS program contains a simple function:

```
PROGRAM FUNCT;

{ FUNCT.PAS: Demonstrate function basics. }

VAR
    num, expo, powr : Real;

FUNCTION power( base, exponent : Real ) : Real;

    BEGIN
        IF (base > 0) THEN
            Power := Exp( exponent * Ln( base ) )
        ELSE
            Power := -1.0;
END;

BEGIN
    Write( 'Enter a number: ' );
    Readln( num );
    Write( 'Enter an exponent: ' );
    Readln( expo );
    powr := Power( num, expo );
    Writeln( num, ' ^ ', expo, ' = ', powr );
END.
```

The FUNCT.PAS program prompts you to enter two numbers, a base and an exponent. Then it calls the `power` function to raise the base to the exponent. Typical output appears below:

```
Enter a number: 2
Enter an exponent: 8

2.00000000000000E+0000 ^ 8.00000000000000E+0000 = 2.56000000000000E+0002
```

FUNCT.PAS raises `2` to the eighth power, giving a result of `256`.

3.3.1 Calling Functions

Function calls are identical to procedure calls. You state the function's name, listing in parentheses any arguments that the function requires. The only difference is in where the call can appear. A procedure call can stand alone as a statement, but a function call, because it returns a value, must appear in an assignment or expression.

The following statement from FUNCT.PAS calls the `power` function, assigning its return value to the variable `powr`:

```
powr := power( num, expo );
```

Notice the similarity to a procedure call. The `power` function takes two arguments, which are listed in parentheses after the function name.

The previous example uses the function call in an assignment. Function calls can also appear in expressions:

```
dazzle := 12 * surprise( 730, 88 ) / 2;
```

Here, the `surprise` function appears as part of the expression to the right of the assignment symbol.

3.3.2 Returning Values from Functions

A function returns a value by assigning the value to its own name. In the `power` function, for instance, this statement causes the function to return the value `-1.0`:

```
power := -1.0;
```

The return value must match the type declared in the function head. Since the `power` function returns a value of type **Real**, the above statement uses the value `-1.0` (with a decimal point).

3.3.3 Declaring Functions

Function declarations are identical to procedure declarations except that you substitute **FUNCTION** for **PROCEDURE** and declare the function's return type after the argument list. Below is the function head from FUNCT.PAS:

```
FUNCTION power( base, exponent : Real ) : Real;
```

Following the argument list, separated by a colon, is the identifier **Real**, which indicates that the `power` function returns a value of type **Real.** If `power` returned an integer value, you would replace the **Real** with **Integer**, and so on.

Again, except for the differences noted in this section, functions are identical to procedures. They can handle arguments and local variables exactly as described in Section 3.2, "Procedures."

3.4 Nested Procedures

In addition to local variables, a procedure can declare other procedures. You can "hide" one procedure declaration inside another. Like a local variable, the hidden procedure is visible only in the procedure where it is declared. This feature, which is unique to Pascal, allows you to limit the visibility of a procedure (and that procedure's local variables) in the same way you limit the visibility of local variables. Nesting applies to both procedures and functions.

The HIDEPROC.PAS program demonstrates procedure nesting:

```
PROGRAM hideproc;

{ HIDEPROC.PAS: Demonstrate procedure nesting. }

VAR
    globl : Integer;

PROCEDURE proc( p_parm : Integer );

    VAR
        p_locl: Integer;

    PROCEDURE hidden( hidn_parm : Integer );

         VAR
             hidn_locl: Integer;

         BEGIN
             Writeln( 'hidden can see: globl, p_parm, '+
                      'p_locl, hidn_parm, hidn_locl' );
         END; { hidden procedure }

     BEGIN
         Writeln( 'Proc can see: globl, p_parm, p_locl' );
         hidden( 44 ); { Pass argument to hidden }
     END; { proc }

BEGIN { main program }
    Writeln( 'Main program can see: globl' );
    proc( 99 ); { Pass argument to proc  }
END.
```

HIDEPROC.PAS produces this output:

```
Main program can see: globl
Proc can see: globl, p_parm, p_locl
Hidden can see: globl, p_parm, p_locl, hidn_parm, hidn_locl
```

The program has two procedures named `proc` and `hidden`. Because the `hidden` procedure declaration appears in the `proc` declaration, `hidden` is visible only inside `proc`.

The program's output shows how nesting affects variable visibility. At the deepest level in HIDEPROC.PAS—inside `hidden`—all of the program's variables are visible. The `hidden` procedure can see its own local variables, plus the variables local to the `proc` procedure, plus all of the global variables. At the next level—inside `proc`—visibility is more restricted. The `proc` procedure can see its own local variables and the global variables, but not the variables local to `hidden`. The main program has the most restricted visibility. It can see only global variables.

Nesting also affects the visibility of a procedure itself. The `hidden` procedure can be called only from the `proc` procedure, where it is declared. If you call `hidden` from the main program, QuickPascal issues an error, just as it would if the main program referred to one of the local variables in `proc`.

3.5 Recursion

"Recursion" is the ability of a procedure or function to call itself. The primary use of recursion is in solving certain mathematical problems that require repetitive operations.

The RECURSE.PAS program demonstrates recursion. It is a revision of the LOCAL.PAS program that demonstrated local variables. Like its predecessor, RECURSE.PAS computes a factorial. But instead of a loop, it uses a recursive function named `factor`:

```
PROGRAM recurse;

{ RECURSE.PAS: Demonstrate recursion. }

USES
    Crt;

VAR
    num : Byte;
    result : Real;

FUNCTION factor( value : Byte ) : Real;
    BEGIN
        IF (value > 1) THEN
            factor := value * factor( value - 1 )
        ELSE
            factor := 1.0;
    END; { factor }

BEGIN
    Write( 'Enter a number smaller than 34: ' );
    Readln( num );
    result := factor( num );
    Write( 'Factorial of ', num, ' is ' );
    Writeln( result );
END.
```

The output from RECURSE.PAS and LOCAL.PAS is identical:

```
Enter a number smaller than 34: 5
Factorial of 5 is 1.20000000000000E+0002
```

The only difference between normal functions and recursive functions is that a recursive function contains a statement that calls itself. Here is the recursive statement in RECURSE.PAS:

```
Factor := value * factor( value - 1 )
```

The expression on the right side of the assignment operator contains a call to the `factor` function. The first call to `factor` can trigger a second call, which can trigger a third call, and so on.

The IF statement at the beginning of the `factor` function prevents the function from calling itself endlessly. Every recursive procedure and function must include such an exit mechanism.

Most recursive procedures and functions exploit the fact that each invocation of a procedure or function creates a new set of local variables. Recursion can be very efficient in terms of programming time. You may be able to solve a complex math problem with only a few lines of code. But deeply recursive procedures can also be memory inefficient, consuming huge amounts of memory to store local variables.

CHAPTER 4

4

Controlling Program Flow

Like other high-level languages, Pascal offers a wide variety of ways to control a program's flow of execution. This chapter discusses looping statements, which perform repetitive actions, and decision-making statements, which transfer control based on logical tests. Before examining those statements in detail, this chapter briefly summarizes the operators used in logical tests.

4.1 Relational and Boolean Operators

All of the looping and branching statements in Pascal depend on the outcome of a Boolean (true or false) test. Such tests use relational and Boolean operators, which look familiar to anyone who knows BASIC or C.

Even if you have never seen a line of Pascal code, you may be able to guess that the statement

```
IF my_val = 20 THEN Writeln( 'my_val equals 20' );
```

prints the message

```
my_val equals 20
```

if the value of the variable `my_val` equals `20`. (The Pascal **IF** statement, as you'll read later in this chapter, works very much like **IF** in BASIC and C.)

The example uses the equality operator (=) to compare the variable `my_val` to the constant `20`. It produces a **True** result when `my_val` equals `20`, and a **False** result in every other case.

NOTE *__True__ and __False__ are symbolic values in Pascal. Although they are represented by actual numbers internally, you don't need to worry about what those numbers are.*

Relational operators, including the equality operator, compare two values and produce a **True** or **False** result. Table 4.1 lists all of the Pascal relational operators.

Table 4.1 Relational Operators

Operator	Description
=	Equal
< >	Not equal
<	Less than
>	Greater than
<=	Less than or equal
>=	Greater than or equal

A second group of operators allows you to perform Boolean logical operations. They are listed in Table 4.2.

Table 4.2 Boolean Operators

Operator	Description
NOT	Negation
AND	Logical **AND**
OR	Logical **OR**
XOR	Exclusive **OR**

Boolean operators (except for **NOT**) can act on one or two values, allowing more complex logical tests. For instance, the statement

```
IF ((my_val > 3) AND (my_val < 20)) THEN Writeln( 'Wahoo!');
```

tests two conditions instead of one. It prints the message

```
Wahoo!
```

if the value of `my_val` is greater than `3` and less than `20`.

Pascal provides many more operators, but these are the important ones for controlling program flow. The QP Advisor contains information about all of the QuickPascal operators.

4.2 Looping Statements

A loop performs one of the most basic computer operations: repeating an action. This section discusses the Pascal looping statements: **WHILE**, **REPEAT**, and **FOR**.

4.2.1 WHILE Loops

A **WHILE** loop is the simplest kind of loop. It repeats 0 or more times, as long as a given condition remains true. The QWHILE.PAS program contains a simple **WHILE** loop.

```
PROGRAM qwhile;

{ QWHILE.PAS: Demonstrate WHILE loop. }

VAR
    count : Integer;

BEGIN

    count := 0;

    WHILE count < 10 DO
        BEGIN
        Writeln( 'count = ', count );
        count := count + 2;
        END;

END.
```

Here is the output from QWHILE.PAS:

```
count = 0
count = 2
count = 4
count = 6
count = 8
```

A **WHILE** loop begins with the **WHILE** keyword followed by a condition. The loop repeats as long as the condition remains true. In QWHILE.PAS, the condition is

```
count < 10
```

so the loop continues as long as the value of the variable `count` is less than `10`. After the condition is the **DO** keyword followed by a "loop body," which

can be a single statement or a statement block. In QWHILE.PAS, the loop body is a statement block:

```
BEGIN
    Writeln( 'count = ', count );
    count := count + 2;
END;
```

You should enclose the body of a **WHILE** loop with **BEGIN** and **END**, even if the loop is only one statement. This convention prevents any confusion about where the loop body ends.

It's important to know that a **WHILE** loop tests its condition *before* it executes the loop body. Unlike some other kinds of loops, it's possible for a **WHILE** loop to skip everything in its loop body. If the test condition is false when a **WHILE** loop begins, the loop body does not execute at all. For instance, if `count` has the value `10` when the above loop begins, QWHILE.PAS doesn't print anything.

4.2.2 REPEAT Loops

A **REPEAT** loop is an inverted **WHILE** loop. It tests the condition *after* it executes the loop body, and the loop repeats *until* the test condition becomes true.

The QREPEAT.PAS program performs the same task as QWHILE.PAS, but it uses a **REPEAT** loop instead of a **WHILE** loop.

```
PROGRAM qrepeat;

{ QREPEAT.PAS: Demonstrate REPEAT loop. }

VAR
    count : Integer;

BEGIN

    count := 0;

    REPEAT
        Writeln( 'count = ', count );
        count := count + 2;
    UNTIL ( count > 8 );

END.
```

The output from QREPEAT.PAS and QWHILE.PAS is identical:

```
count = 0
count = 2
count = 4
count = 6
count = 8
```

The **REPEAT** loop in QREPEAT.PAS contains the same loop body as the **WHILE** loop in QWHILE.PAS:

```
REPEAT
    Writeln( 'count = ', count );
    count := count + 2;
UNTIL ( count > 8 );
```

You don't need to enclose the loop body of a **REPEAT** loop with the **BEGIN** and **END** keywords (although adding them does no harm). Since the loop body is already enclosed between two keywords (**REPEAT** and **UNTIL**), there can be no confusion about where the block begins and ends.

Remember that a **REPEAT** loop always executes the loop body at least once. If `count` has the value `10` when the loop begins, QREPEAT.PAS prints

```
count = 10
```

even though `10` is clearly greater than `8`, the cutoff value in the test condition. The value of `count` is not tested until after the loop body has executed.

Notice that **WHILE** and **REPEAT** loops use opposite logical tests. A **WHILE** loop continues *as long as* the test condition is true, but a **REPEAT** loop continues *until* the test condition becomes true (or, to put it differently, as long as the test condition is false). To illustrate, the **WHILE** loop in QWHILE.PAS continues as long as `count` is less than `10`:

```
count < 10
```

However, the **REPEAT** loop in QREPEAT.PAS continues until `count` becomes greater than `8`:

```
count > 8
```

4.2.3 FOR Loops

WHILE and **REPEAT** loops are ideal for cases in which you cannot predict how many repetitions are needed. A program that gets keyboard input, for instance, might use **REPEAT** to repeat an action until you press a certain key. Sometimes, however, you know in advance exactly how many repetitions are required.

The **FOR** loop repeats a statement, or statement block, a set number of times. The QFOR.PAS program contains a simple **FOR** loop:

```
PROGRAM qfor;

{ QFOR.PAS: Demonstrate FOR loop. }

VAR
    count : Integer;

BEGIN

    FOR count := 0 TO 10 DO
        Writeln( 'count = ', count );

END.
```

QFOR.PAS produces this output:

```
count = 0
count = 1
count = 2
count = 3
count = 4
count = 5
count = 6
count = 7
count = 8
count = 9
count = 10
```

The **FOR** loop in QFOR.PAS counts from `0` to `10` in increments of 1:

```
FOR count := 0 TO 10 DO
    Writeln( 'count = ', count );
```

In this example the control variable `count` is first set to `0`. Each repetition executes the loop body once and adds 1 to `count` until `count` reaches `10`, the terminating value.

FOR loops can count down as well as up. The **TO** keyword makes the loop count up in increments of 1, and **DOWNTO** has the opposite effect. If you substitute this loop in QFOR.PAS, the loop counts down from `10` to `0` in increments of 1:

```
FOR count := 10 DOWNTO 0 DO
    Writeln( 'count = ', count );
```

The loop body in QFOR.PAS happens to be a single statement. If the loop body is a statement block, you must enclose the block with **BEGIN** and **END** statements:

```
FOR count := 0 TO 10 DO
    BEGIN
    Writeln( 'count = ', count );
    Writeln( 'Another statement' );
    END;
```

4.3 Decision-Making Statements

Decision-making statements allow your program to perform different actions based on the outcome of a logical test. This section examines the Pascal decision-making statements: **IF** and **CASE**.

4.3.1 IF Statements

An **IF** statement consists of the **IF** keyword followed by a test expression and the **THEN** keyword. After **THEN** is a statement or statement block. The statement is executed if the test expression is true, or skipped if it is false.

The QIF.PAS program contains a simple **IF** statement:

```
PROGRAM qif;

{ QIF.PAS: Demonstrate IF statement.  }

VAR
    my_val : Integer;

BEGIN

    my_val := 3;

    IF (my_val = 3) THEN Writeln( 'my_val equals 3' );

END.
```

Here is the **IF** statement from QIF.PAS:

```
IF (my_val = 3) THEN Writeln( 'my_val equals 3' );
```

In this statement the test condition

```
(my_val = 3)
```

compares the variable `my_val` to the constant `3`. Since the comparison is true (`my_val` does equal `3`), QIF.PAS prints:

```
my_val equals 3
```

The statement following **THEN** can be a single statement or a statement block. A block must be enclosed with **BEGIN** and **END** statements:

```
IF (my_val = 3) THEN
    BEGIN
    Writeln( 'my_val equals 3' );
    Writeln( 'Another statement' );
    END;
```

4.3.2 ELSE Clauses

The **ELSE** keyword allows an **IF** statement to perform more complex branching. The QELSE.PAS program adds an **ELSE** clause to QIF.PAS:

```
PROGRAM qelse;

{ QELSE.PAS: Demonstrate ELSE clause. }


VAR
    my_val : Integer;

BEGIN

    my_val := 555;

    IF (my_val = 3) THEN
         Writeln( 'my_val equals 3' )
    ELSE
         Writeln( 'my_val does not equal 3' )

END.
```

The QELSE.PAS program contains the following **IF...ELSE** statement:

```
IF (my_val = 3) THEN
     Writeln( 'my_val equals 3' )
ELSE
     Writeln( 'my_val does not equal 3' );
```

The **ELSE** clause allows the **IF** statement to take two alternate actions. The **IF** statement prints

```
my_val equals 3
```

if `my_val` equals `3`, and it prints

```
my_val does not equal 3
```

in all other cases. Note that a semicolon does not precede the **ELSE**, because **ELSE** is considered part of the **IF** statement.

You can nest and combine **IF** statements and **ELSE** clauses as needed. Each **ELSE** is associated with the closest preceding **IF** that does not have an **ELSE** already. Consider this example:

```
IF (my_val > 9) THEN
    IF (my_val = 10) THEN
        Writeln( 'Ten' )
    ELSE
        Writeln( 'Greater than nine, but not ten ')
ELSE
    Writeln( 'Less than ten' );
```

The example can take three different actions. If `my_val` is greater then `9` and equal to `10`, it prints `Ten`. If `my_val` is greater then `9` but not equal to `10`, it prints `Greater than nine, but not ten`. If `my_val` is less than or equal to `9`, it prints `Less than ten`.

Use a **BEGIN...END** block to enclose the nested **IF** statement when the **ELSE** applies to the surrounding **IF**, as shown below:

```
IF (my_val > 9) THEN
     BEGIN
     IF (my_val = 10) THEN
           Writeln( 'Ten' );
     END
ELSE
     Writeln( 'Less than ten ');
```

This example prints `Ten` if `my_val` equals `10`, and `Less than ten` otherwise. Without the **BEGIN...END** block, the **ELSE** clause would apply to the `IF (my_val = 10)` condition, and not to `IF (my_val > 9)`.

4.3.3 CASE Statements

As the previous example demonstrates, complex **IF...ELSE** statements can be difficult to read. If all of the branches test the same value (as in the previous example), the **CASE** statement provides a cleaner solution.

The Pascal **CASE** statement is similar to **SELECT CASE** in BASIC or the **switch** statement in C. It can branch to several different alternatives based on the value of a single ordinal expression test. The QCASE.PAS program contains a simple **CASE** statement:

```
PROGRAM qcase;

{ QCASE.PAS: Demonstrate CASE statement. }

VAR
    my_val : Integer;

BEGIN

    my_val := 33;

    CASE my_val OF
        10 : Writeln( 'Ten' );
        20 : Writeln( 'Twenty' )
        ELSE
            Writeln( 'Not ten or twenty' );
        END;

END.
```

A **CASE** statement begins with the **CASE** keyword, followed by an ordinal expression and the **OF** keyword. The **CASE** statement in QCASE.PAS tests the value of `my_val`:

```
CASE my_val OF
```

Next comes a list of alternatives, each labeled with a constant followed by a colon (a "case constant"):

```
10 : Writeln( 'Ten' );
20 : Writeln( 'Twenty' );
```

The alternatives to execute can be single statements or statement blocks. Each case constant in the list acts as a target. When `my_val` equals `10`, QCASE.PAS executes the statement after the case constant

```
10:
```

When `my_val` equals `20`, control is transferred to the case constant

```
20:
```

When `my_val` doesn't match any case constant, QCASE.PAS executes the statement following **ELSE**:

```
ELSE
    Writeln( 'Not ten or twenty' );
```

The **ELSE** clause is optional. If you omit it, and the value of the expression does not match any of the **CASE** constants, none of the alternatives are executed. Instead, execution proceeds with the first statement following the **CASE** statements.

The **CASE** statement can use as many case constants as needed. The case constant can be a single constant (as shown above), a group, or a range of constants:

```
CASE my_val OF
   1600,2000 : Writeln( 'Leap century' );
   1601..1999: Writeln( 'Non-leap century' );
END;
```

The first case constant includes two values: `1600` and `2000`. The second includes a range of 399 values: `1601` through `1999`.

CHAPTER 5

User-Defined Data Types

5

Instead of limiting a program to using only predefined data types, QuickPascal allows you to create your own custom data types that are relevant to the program at hand.

An ordinal data type is a collection of values where each value (except the first) has a unique value that precedes it and (except for the last) a unique value that follows it. Examples of ordinal data types are **Boolean**, **Char**, **Integer**, and **LongInt**. With QuickPascal, you can create your own ordinal types through the use of enumerated types and subrange types.

An enumerated data type has a series of unique, ordinal values defined in it. You can think of it in terms of an **Integer**, but instead of a range of numbers, you designate a range of your own values. A subrange data type is created by specifying the first and last elements of an existing ordinal type. The type may be one of the standard Pascal ordinal types, or an enumerated type of your own creation.

In addition to enumerated and subrange types, you can also create a set type. A set holds up to 255 unique values from an existing ordinal type.

5.1 Enumerated Data Types

Enumerated data types consist of an ordered list of unique identifiers. The identifiers can be anything from cars to days of the week. An enumerated data type is defined simply by listing the identifiers that make up the type.

The syntax for an enumerated type is

TYPE *ListName* = (*Identifier* [[, *Identifier*...]])

QuickPascal assigns numbers to the identifiers in the list. The first list element is assigned 0, the second is 1, and so on, up to a maximum of 65,535 values. This is the QuickPascal internal representation. You normally refer to a value by its name.

An example of an enumerated data type is

```
TYPE
    japanese_cars = ( honda, isuzu, nissan, toyota );
```

The use of the values assigned to elements in enumerated lists is shown in this statement:

```
VAR
    rental_car : japanese_cars;
BEGIN
    rental_car := nissan;
```

This puts `nissan`, with a value of 2, into the variable `rental_car`.

There may be cases where you want the first element of a list to have a non-zero value. You can accomplish this by declaring a fake identifier as the first element:

```
TYPE
    place = (null, first, second, third, fourth);
```

This provides a more natural numeric ordering for the list elements.

QuickPascal provides two procedures and five functions to manipulate enumerated data types. The predefined routines are **First**, **Last**, **Succ**, **Pred**, **Inc**, **Dec**, and **Ord**.

5.1.1 The First Function

The **First** function returns the first element of any ordinal type. The function is passed an ordinal type, and returns a value of the same type.

The following example shows you how to use **First**:

```
PROGRAM enums;


TYPE
    my_type = -5..5;
    greek   = (alpha, beta, gamma, delta, epsilon);
    subgreek = gamma..epsilon;
```

```
BEGIN
    Writeln( 'Type ':20, 'First(Type)':20, 'Last(Type)':20);
    Writeln;
    Writeln( 'Integer':20, First( Integer ):20, Last( Integer ):20 );
    Writeln( 'LongInt':20, First( LongInt ):20, Last( LongInt ):20 );
    Writeln( 'ShortInt':20, First( ShortInt ):20, Last( ShortInt ):20 );
    Writeln( 'Word':20, First( Word ):20, Last( Word ):20 );
    Writeln( 'Char(ord)':20, Ord( First( Char ) ):20,
             Ord( Last( Char ) ):20 );
    Writeln( 'Boolean':20, First( Boolean ):20, Last( Boolean ):20 );
    Writeln( 'my_type':20, First( my_type ):20, Last( my_type ):20 );
    Writeln( 'greek(ord)':20, Ord( First( greek ) ):20,
             Ord( Last( greek ) ):20 );
    Writeln( 'subgreek(ord)':20, Ord( First( subgreek ) ):20,
             Ord( Last( subgreek ) ):20 );

END.
```

5.1.2 The Last Function

The **Last** function returns the last element of any ordinal type. The function is passed an ordinal type, and returns a value of the same type.

The previous example shows how to use **Last**.

5.1.3 The Succ Function

The **Succ** function returns the successor, or following element, of an enumerated value.

Sample Function Call	Ordinal Type	Result
Succ(second)	place	third
Succ(isuzu)	japanese_cars	nissan
Succ(15)	Integer	16
Succ('a')	Char	'b'
Succ(False)	Boolean	True

Both standard and user-defined ordinal types can be used with this function.

If range checking {**$R+**} is set, a run-time error occurs if you try to assign an element beyond the last in the list. You can avoid this problem by using the **Last** function to check if the element is last in the list.

5.1.4 The Pred Function

The **Pred** function returns the predecessor, or preceding element, of an enumerated value.

Sample Function Call	Ordinal Type	Result
`Pred(isuzu)`	`japanese_cars`	`honda`
`Pred(second)`	`place`	`first`
`Pred(12)`	`Integer`	`11`
`Pred('b')`	`Char`	`'a'`
`Pred(True)`	`Boolean`	`False`

Both standard and user-defined enumerated data types can be used with this function.

If range checking {**$R+**} is set, a run-time error occurs if you try to assign an element preceding the first in the list. You can avoid an error by using the **First** function to check if the element is first in the list.

5.1.5 The Inc Procedure

The **Inc** procedure provides a shorthand form of the **Succ** function. Upon calling the procedure, the variable it is passed is incremented by the number of elements specified.

For example, instead of using **Succ** as follows:

```
status := Succ( status );
status := Succ( status );
```

the **Inc** procedure could be used as

```
Inc( status, 2 );
```

If no increment parameter is specified, the variable is incremented by one:

```
Inc( status );
```

The **Last** function should be used to avoid unpredictable results when incrementing elements.

5.1.6 The Dec Procedure

The **Dec** procedure is an alternative to the **Pred** function. Upon calling the procedure, the variable it is passed is decremented by the number of elements specified.

For example, instead of using **Pred** as follows:

```
status := Pred( status );
status := Pred( status );
```

the **Dec** procedure could be used as

```
Dec( status, 2 );
```

If no decrement parameter is specified, the variable is decremented by one:

```
Dec( status );
```

The **First** function should be used to avoid unpredictable results when decrementing elements.

5.1.7 The Ord Function

The **Ord** function returns the ordinal number of an enumerated element. Since each enumerated element is unique, you need not mention the enumerated data type that the element belongs to. Ordinal values start at zero.

Sample Function Call	Ordinal Type	Result
`Ord(first)`	`place`	`1`
`Ord(toyota)`	`japanese_cars`	`3`
`Ord('a')`	`Char`	`97`
`Ord(False)`	`Boolean`	`0`

The **Ord** function accepts both enumeration constants and variables. Both standard and user-defined enumerated data types can be used with the function.

5.2 Subrange Types

Sometimes, only a few of the elements of an existing data type are needed. Instead of declaring a new data type, Pascal allows you to define a subrange of either a standard or enumerated data type.

For example, in a grading program, the variable `test_score` records grades between 0 and 100. Instead of creating a new data type for the variable, an **Integer** subrange is declared with a minimum value of 0 and a maximum value of 100.

You can define subranges of the following ordinal types:

- Integers
- Characters
- Enumerations

To declare a subrange type, identify the *SubrangeName* and define the first and last value in the range, connected with two dots, as shown in the syntax below:

SubrangeName = *FirstValue*..*LastValue*

The *FirstValue* and *LastValue* must be constants of the same type, with the *FirstValue* as a predecessor of the *LastValue*. That is,

Ord(**FirstValue**) <= *Ord*(**LastValue**)

Subrange types serve two useful purposes:

1. If the range of a type needs to be changed, only one change in the declaration is necessary.
2. QuickPascal automatically checks the range assigned to variables of the subrange type, with the following limitations:
   - Range checking must be turned on with the compiler directive {**$R+**}. This can be done at the beginning of the program, or turned on ({**$R+**}) and off ({**$R–**}) around a statement where a subrange variable is used. When range checking is on, QuickPascal generates a run-time error if a value outside of the subrange is assigned to a variable of the subrange type. See Appendix B, "Compiler Directives," for more information.
   - QuickPascal does range checking only in direct assignment statements. It does not check for out-of-range values in loop-control variables or **Read/Write** statements.

5.2.1 Integer Subranges

Integer subranges define a range of valid integer values. Some examples of simple integer subranges include

```
TYPE
    screen_columns = 1..80;
    die_faces      = 1..6; { values on dice faces }
    days           = 1..31; { max. 31 in a month }
    months         = 1..12; {12 months/year}
    years          = 1900..2099; {the years DOS knows}
    seconds        = 0..59;
    minutes        = 0..59;
    hours          = 0..23;
```

Constant identifiers can define the subranges. Examples include

```
CONST
    max_col            = 80;
    max_row            = 25; { or 43 for EGA screen }
    max_days_per_month = 31;
    months_per_year    = 12;
    min_year           = 1900;
    max_year           = 2099;

TYPE
    screen_columns = 1..max_col;
    screen_rows    = 1..max_row;
    days           = 1..max_days_per_month;
    months         = 1..months_per_year;
    years          = min_year..max_year;
```

Adding constants enhances readability and simplifies the subrange limits. For example, if you are developing a program that employs EGA video, you need to display 43 lines per screen. Variables of the `screen_rows` type should fall within the 1..43 range. All that is required is to change the constant `max_row` to `43` at one location.

QuickPascal permits expressions in defining ranges. The following are examples of declarations that include constant expressions:

```
CONST
    sec_per_minute  = 60;
    minute_per_hour = 60;
    hour_per_day    = 24;

TYPE
    seconds = 0..sec_per_minute - 1;
    minutes = 0..minute_per_hour - 1;
    hours   = 0..hour_per_day - 1;
```

5.2.2 Character Subranges

Character subranges define a range of acceptable **Char** values. Examples include

```
TYPE
    up_case_char = 'A'..'Z';
    lo_case_char = 'a'..'z';
    digit_char   = '0'..'9';
    ctrl_char    = #0..#31;
    { range of characters after 'Z' and before 'a' }
    between_Z_a  = Succ('Z') .. Pred('a');
```

Any range of ASCII characters can be defined in a subrange. Note the use of the **Succ** and **Pred** functions in the last example to define the six ASCII characters between 'Z' and 'a'. (See the ASCII character chart in Appendix A.)

5.2.3 Enumerated Subranges

Enumerated subranges limit the range of permissible enumerated values. An example is

```
TYPE
    vehicles = (volkswagen, honda, toyota, corvette,
                porsche, ferrari, suburban, blazer,
                bronco);

    economy_cars = volkswagen..toyota;
    sports_cars  = corvette..ferrari;
```

The type `economy_cars` is a subrange with legal values of `volkswagen`, `honda`, and `toyota`. Similarly, the `sports_cars` type has only the enumerated `corvette`, `porsche`, and `ferrari` values.

5.3 Sets

Sets are structured, user-defined data types. In mathematics, a set is an unordered collection of elements. The concept of a set is the same in QuickPascal. A set holds only unique values. For example, if A, B, and C were contained in a set, and you added B to it, the set would still only contain A, B, and C, not A, B, B, and C. Sets are useful for holding a collection of attributes or determining if an element is a member of a particular group.

The syntax for declaring a set is

SetName = **SET OF** *OrdinalType*

The argument *OrdinalType* is an ordered range of values. The members of the *OrdinalType* must be all of the same type, and can be single elements or a subrange. The *OrdinalType* cannot have more than 256 possible values. This limits

sets to the predefined **Boolean**, **Char**, and **Byte** types, and restricts the use of **Word**, **Integer**, and **LongInt** types.

As an example, a program might declare a set of uppercase letters and vowels to be

```
CONST
    vowels     = ['A', 'E', 'I', 'O', 'U'];
    upper_case = SET OF 'A'..'Z' = ['A'..'Z'];
```

Once you have declared a set, the operator **IN** can be used to test for the presence or absence of a specified element. For example, in these statements

```
IF ch IN upper_case THEN...
IF ch IN vowels THEN...
```

IN returns a true result if `ch` is a member of the set, and false if it is not.

5.3.1 Declaring Set Types

Sets can be declared with a variety of types. Included are

- Predefined ordinal types of **Boolean**, **Char**, and **Byte**

```
TYPE
    boolean_set = SET OF Boolean;
    char_set    = SET OF Char;
    byte_set    = SET OF Byte;
```

- Subranges of predefined types (either directly as the first set or indirectly as the last)

```
TYPE
    bits       = 1..7;
    byte_bits  = SET OF bits;
    up_case    = SET OF 'A'..'Z';
    lo_case    = SET OF 'a'..'z';
```

- Enumerated types

```
TYPE
    transportation  = (bicycle, motorcycle, car, truck,
                       bus);
    four_wheels     = car..bus;
    trans_set       = SET OF transportation;
    four_wheel_set  = SET OF four_wheels;
```

- Variables

```
VAR
    fast_trans    : four_wheels;
    lower_letters : lo_case;
    num1, num2    : byte_bits;
```

- Constants and typed constants

```
CONST
    math_op     : SET OF Char = ['=', '-', '*', '/'];
    vowels      : SET OF Char = ['A','E','I','O','U'];
    up_chars    : SET OF Char = ['A'..'Z'];
    lo_chars    : SET OF Char = ['a'..'z'];
    cheap_trans : SET OF transportation =
                  [bicycle, motorcycle, bus];
```

Character set constants are useful in representing fixed sets of characters used in menu selections. For example, if you have a menu with the following selections:

```
Add    Change    Delete    Print    View    Store    Recall
```

where the capital letters are "hot" keys used to quickly select a menu option, the corresponding typed constant would look like this:

```
CONST
    menu_char : SET OF Char = ['A','C','D','P','V','S','R'];
```

5.3.2 Assigning Set Elements to Variables

To assign set elements to a set variable, use the square brackets:

SetVariable := [*SetElements*]

If there are no set elements present, the set variable is assigned an empty set (*SetVariable* : = [ ]). Set variables may be initialized in this way.

A set may be constructed from a list of single elements, a subrange, or a combination of both. Examples of assigning set elements to variables are

```
set1 := [1, 3, 5, 7, 9]; { single elements }
set2 := [0..7]; { subrange }
set3 := [0..7, 14, 15, 16]; { subrange & single elements }
char_list := ['A'..'Z','a'..'z','0'..'9']; { subranges }
```

5.3.3 Set Operators

Although individual elements of a set cannot be directly accessed, a variety of operators is available to test membership and manipulate the set as a whole.

These operators offer powerful and flexible methods of creating new sets with elements from existing sets. Set operators supported by QuickPascal include

- Relational operators
- **IN** operator
- Set-union operator
- Set-difference operator
- Set-intersection operator

5.3.3.1 Relational Operators

A variety of relational operators is available to test set membership. Based on the condition of an expression, the operator will return **True** or **False**.

Table 5.1 lists the relational operators that work on sets, with `A` and `B` as example sets.

Table 5.1 Relational Operators

Expression	Returns True if
`A = B`	`A` and `B` are identical.
`A <> B`	At least one element in `A` is not in `B`, or at least one element in `B` is not in `A`.
`A <= B`	All elements in `A` are in `B`.
`A >= B`	All elements in `B` are in `A`.

5.3.3.2 IN Operator

As previously discussed, the **IN** operator tests for set membership. This operator returns a Boolean result indicating whether a value is a member of the set. The tested value must be the same or of a compatible type with the tested set's base type. The syntax is

Value **IN** *Set*

For example,

```
ch IN vowels
'i' IN consonants
operator IN ['+', '-', '/', '*']
```

Figure 5.1 displays the action of set operators on sets `A` and `B`. The shaded area represents the result of the operations on the sets.

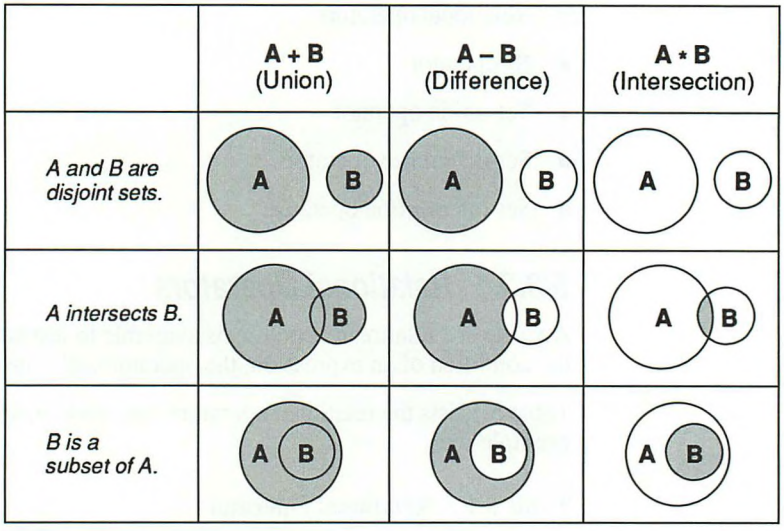

Figure 5.1 Set Operators

5.3.3.3 Set-Union Operator

The set-union operator (+) merges two sets into a third set. If either set is a subset of the other, combining the two sets results in a set that is the same as the larger set. In the example below, two sets with unique members are merged, resulting in a larger set:

```
set1 := ['A'..'Z'];
set2 := ['a'..'z'];
set3 := set1 + set2;
set3 := ['A'..'Z','a'..'z']; { same as previous assignment }
```

In the next example, the two character sets have overlapping members. The united set consists of `['A'..'Z']` with any overlapping members represented only once.

```
set1 := ['A'..'L'];
set2 := ['H'..'Z'];
set3 := set1 + set2;
set3 := ['A'..'Z']; { equivalent to previous assignment }
```

The third example shows a set being united with a subset. This results in `set3` and `set1` having the same members.

```
set1 := ['A'..'L'];
set2 := ['F'..'J'];
set3 := set1 + set2;
set3 := ['A'..'L']; { equivalent to previous assignment }
```

The union operator is also important for adding to the membership of a set. In the following example, a character set is initialized and then a **FOR** loop is used to add the characters `A,B,C,D,E,F,...,`:

```
set1 := []; { initialize }
FOR ch := 'A' TO 'Z' DO
    set1 := set1 + [ch];
```

Note the presence of the set brackets around the `ch` variable. They are required in order to make `[ch]` a single-element set.

Refer to Figure 5.1 for an illustration of the union operator.

5.3.3.4 Set-Difference Operator

The set-difference operator (–) creates a set that contains all of the members of the first set that do not appear in the second set. For example, in the statement

```
set3 := set1 - set2;
```

`set3` will contain all of the elements in `set1` that are not in `set2`.

If `set1` and `set2` have the same members, then `set3` becomes an empty set. If `set2` is a subset of `set1`, then `set3` comprises of the members of `set1` that are not common to `set2`.

In the following example, the difference between two sets with unique members is assigned to a third set. The resulting set has the same members as `set1` since the operand sets have nothing in common:

```
set1 := ['A'..'Z'];
set2 := ['a'..'z'];
set3 := set1 - set2;
set3 := ['A'..'Z']; { equivalent to previous assignment }
```

In the next example, the two character sets have overlapping members. The resulting set is made up of `['A'..'G']`:

```
set1 := ['A'..'L'];
set2 := ['H'..'Z'];
set3 := set1 - set2;
set3 := ['A'..'G']; { equivalent to previous assignment }
```

The next example shows the difference of a set with its subset. The result is that `set3` contains `['A'..'E']`, the members of the first set not found in the second one.

```
set1 := ['A'..'L'];
set2 := ['F'..'J'];
set3 := set1 - set2;
set3 := ['A'..'E', 'K', 'L']; { equivalent to previous
                                assignment }
```

The difference operator can also be used to strip single members from set variables. The last example for the union operator can be rewritten to use the difference operator in the following way. The character set is initialized to `['A'..'Z']` and a **FOR** loop is used to eliminate the `A, B, D, F, H,` characters and so on.

```
oddeven := 0;
set1 := ['A'..'Z']; { initialize }
FOR ch := 'B' TO 'Z' DO
    BEGIN
    IF (NOT Odd( Oddeven )) THEN
        set1 := set1 - [ch];
    Inc( Oddeven );
    END;
```

Refer to Figure 5.1 for an illustration of the difference operator.

5.3.3.5 Set-Intersection Operator

The set-intersection operator (*) is used to extract all of the elements that are in two similarly typed sets. For example, with

```
set3 := set1 * set2;
```

`set3` will contain all of the elements that are in both `set1` and `set2`.

In the next example, the two character sets have overlapping members. The intersection set is `['H'..'L']`:

```
set1 := ['A'..'L'];
set2 := ['H'..'Z'];
set3 := set1 * set2;
set3 := ['H'..'L'];
```

The intersection of a set with its subset returns the members of the subset. The next example shows such an operation. The result is that `set3` and `set2` have the same members:

```
set1 := ['A'..'L'];
set2 := ['F'..'J'];
set3 := set1 * set2;
set3 := ['F'..'J'];
```

In the following example, two sets with unique members are intersected. The resulting set is empty, since the operand sets have nothing in common:

```
set1 := ['A'..'Z'];
set2 := ['a'..'z'];
set3 := set1 * set2;
set3 := [ ];
```

CHAPTER 6

6

Arrays and Records

This chapter presents data types that hold organized collections of data in a definite order.

An "array" is a collection of data items of the same type. Programs use arrays in situations where a standard data format is repeated many times. For example, numbers representing the Gross National Product for all the years from 1900 to 1980 might be placed in an array.

A "record" is a collection of data items having different types. Programs use records in situations where a variety of data have a close association. For example, all of the information on a given employee—name, salary, and security clearance—might be placed in a single record.

Finally, this chapter presents the "variant record" type, which is an extension of the record type. A variant record allows you to use different data formats to access the same area of memory, enabling you to create a record that holds different kinds of information at different times.

Each of the major sections in this chapter introduces a data type, shows how to declare it, then shows how to access its components.

6.1 Arrays

An array is a collection of elements that share the same data type and a common variable name. You access an element in the array by specifying the position of the element. An integer or ordinal value indicates the position and is called an "index."

Pascal lets you declare arrays of any type. You can even declare arrays of arrays, which are, in essence, two-dimensional arrays. You can declare arrays with any number of dimensions. Multidimensional arrays have many applications and

appear often in Pascal programs. For example, a program that requires a grid will use a two-dimensional array, and a program that maps three-dimensional space will use a three-dimensional array. The simplest arrays are one-dimensional. They consist of a linear sequence of elements.

6.1.1 Declaring Arrays

In many languages (such as C), arrays must always start at a particular index number (such as 0 or 1). In Pascal, however, you can define the bounds of an array with almost any ordinal type:

ARRAY [*IndexType*] **OF** *ElementType*

To declare a multidimensional array, you declare an index type for each dimension:

ARRAY [*IndexType*, *IndexType* ...] **OF** *type*

IndexType can be any ordinal type except **LongInt** or subranges of **LongInt**. Most often, programs use subranges in array declarations. The lower and upper bounds of the subrange give the lowest and highest index, and also determine the size of the array. For example:

```
TYPE
    income_per_year = ARRAY[1977..1989] OF LongInt;
    class_size = ARRAY[1..12] OF Integer;
    grid = ARRAY[-5..5, -10..10] OF Real;
```

As you can see, even negative numbers can serve as array bounds. Because Pascal is so structured, you can use many different integer subranges and enumerated types to index arrays. For example:

```
TYPE
    pay = LongInt;
    rank = (private, sergeant, lt, captain, major, general);
    officers = lt..general ;
    letters = 'A'..'Z';
    my_arr = ARRAY[1..10] OF Real;

VAR
    low_pay : ARRAY[private .. sergeant] OF pay;
    high_pay : ARRAY[officers] OF pay;
    ascii_code : ARRAY[letters] OF Word;
    big_arr : ARRAY[letters] OF my_arr;
```

The last example above, `big_arr`, is really a two-dimensional array and is equivalent to the following declaration, which specifies the ranges of the two dimensions explicitly:

```
big_arr : ARRAY[letters, 1..10] OF Real;
```

Bear in mind the difference between the index type of an array and the element type. The item in brackets defines the index type of the array and is significant in the following ways:

- The index type determines the range and meaning of indexes. The next section describes how you use indexes to access elements.
- The index type determines the number of elements in the array. For example, an array with subrange `1..500` has 500 elements. An array with subrange `101..103` has three elements.

The elements are declared by the data type at the end of the array declaration. As mentioned above, you can use any data type, including any of the advanced data types mentioned later in this book. Each element of the array has this element type.

6.1.2 Accessing Array Elements

To refer to an element of an array, use the syntax

Name [*Index*]

in which *Name* is the name of the array variable, and *Index* has the index type used to declare the array. If the index type is a subrange, *Index* must fall into the specified range or the program produces errors. Consider the following declarations:

```
VAR
    trio : ARRAY[1..3] OF Word;
    income : ARRAY[1980..1983] OF LongInt;
```

In the example above, the elements of `trio` are referred to as

```
trio[1]
trio[2]
trio[3]
```

The elements of `income` are referred to as

```
income[1980]
income[1981]
income[1982]
income[1983]
```

Each of the elements above has the data type declared for the array—**Word** in the case of `trio` and **LongInt** in the case of `income`. You can refer to an element in any context that is valid for a simple variable of the same type. You can

alter an element, pass it to a procedure, or assign its value to another variable. The following statements are all valid:

```
trio[1] := 50;
trio[2] := trio[1] DIV 2;
Writeln( ' Result is : ', trio[1] + trio[2] );
```

The array index can be a variable as well as a constant. In fact, the power of arrays in programming comes largely from the use of variable indexes. The following code uses a loop variable to efficiently initialize a large array of random numbers:

```
VAR
    i : Integer;
    big_arr : ARRAY[1..1000] OF Word;
BEGIN
    FOR i := 1 TO 1000 DO
        big_arr[i] := Random(100);
```

To refer to an element of a multidimensional array, use the syntax

Name [*Index*, *Index* ...]

in which each *Index* is of the type and range of the corresponding index in the array declaration. For example, the following code uses a nested loop to efficiently initialize a two-dimensional array of random numbers:

```
VAR
    i, j : Integer;
    results : ARRAY[1..max_row, 1..max_col] OF Word;
BEGIN
    FOR i := 1 TO max_row DO
        FOR j := 1 TO max_col DO
            results[i, j] := Random(100);
```

The {$R+} directive causes the program to check for out-of-bound indexes at run time. Use the {$R+} directive during program development, but you may want to turn the directive off ({$R–}) once you finish writing and debugging. This lets the program run faster.

6.1.3 Declaring Constant Arrays

To declare a constant array, first define the array type, then declare the array and the initial values in a CONST statement. As shown in the examples below, follow the array type with an equal sign (=) and a list of initial values enclosed in parentheses. Separate elements with commas. Multidimensional arrays require additional levels of parentheses. For example, in a two-dimensional array, place parentheses around the values for each row.

```
CONST
    max_row    = 5;
    grid_size = 2;

TYPE
    small_int_arr = ARRAY[1..max_row] OF Integer;
    hex_digit_arr = ARRAY[1..16] OF Char;

    grid_xy = 1..grid_size;
    char_grid_type = ARRAY[grid_xy, grid_xy] OF Char;

CONST
    topic_index : small_int_arr = ( 11, 12, 13, 14, 15 );
    hex : hex_digit_arr =
        ( '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
          'A', 'B', 'C', 'D', 'E', 'F' );
    default_state : char_grid_type =
        ( ( 'a', 'j' ), ( 'W', 'T' ) );
```

6.1.4 Passing Arrays as Parameters

This section describes how to pass an entire array to a procedure or function. Note that there is no restriction on how you pass individual elements, which can be passed just like simple variables of the same type.

To pass an array as a parameter, first define the array as an independent type. Then declare the function to take a parameter of this predefined type, and pass a value of this same type to the function. This requirement is necessary for Pascal to ensure that the array is passed correctly.

The following code illustrates how to successfully pass an array:

```
TYPE
    res_arr = ARRAY[1..20] OF Integer;
VAR
    results : res_arr; { results is a res_arr type variable }

    {=====init_arr=====}
    { init_arr accepts a variable with res_arr type and initializes it }
PROCEDURE init_arr( VAR new_arr : res_arr );
    BEGIN
    .
    .
    .
    END;

BEGIN
    { call init_arr, passing it the results variable
      which has type res_arr
    }
    init_arr( results );
```

You don't need to predefine a string type in order to pass a string. However, if the Var-String Checking option (in the dialog box for Compiler directives in the Options menu) is in effect (the default), the length of the string you pass must match exactly the length of the string the procedure expects. For example, if procedure `print_str` is declared as

```
PROCEDURE print_str ( data_str : STRING[20] );
```

then you can only pass strings of type `STRING[20]`.

If Var-String Checking is turned off, then you can pass strings to parameters without regard to length. You can also pass variables of type **STRING**.

Like all variables, arrays can be passed by value or by reference. When you pass an array by value, the entire array is copied to the stack. This can cause you to run out of stack memory quickly if you work with large arrays.

6.1.5 Using the Debugger with Arrays

When you want to display an array during debugging, you can specify three different kinds of Watch expressions:

1. The entire array. You can watch all the elements of an array simply by specifying the array's name. The Watch window displays array elements as a list of comma-delimited items enclosed in parentheses. You may need to scroll through the Watch window to see all of the elements.
2. Specific array elements. You can watch individual elements of an array by specifying either a constant or variable as an index. If you use a variable, then QuickPascal displays a different element whenever the variable changes.
3. A subset of the array elements (a "subarray"). You can specify a subarray by using the following syntax:

   ArrayName[*Index*], *Number*

   For example, `a[4],5` permits you to view the subarray `a[4]` to `a[9]`. You can also use a variable as the index of the first item.

6.2 Records

A record is a collection of variables that can have different types. Each variable within the record has a name to differentiate it from other variables in the record. This name is called a "field." To access an item in a record, you give both the record name and the field.

Records are common in practical applications. For example, consider a program that maintains airplane reservations. For each reservation, there are several relevant pieces of information: customer name, flight number, date, and time. You can combine all of this information into a single record.

Although you could keep track of the different data items in separate variables, placing them together in a single record makes programs easier to write and maintain. Furthermore, as you'll see in Chapter 10, "Binary Files," records are convenient units of data to read and write to disk.

6.2.1 Declaring Records

Declare a record with the following syntax. The record can form part of a type definition or variable declaration.

RECORD
  FieldDeclarations
END

In the syntax display above, *FieldDeclarations* is a variable declaration. The variable name determines the name of the *Field* (see examples). You must separate each field from the next with a semicolon.

The following lines show some simple record type definitions:

```
TYPE
    complex = RECORD
        x_real,
        y_imag : Real;
        END;

    mail_rec = RECORD
        name   : STRING[20];
        street : STRING[25];
        city   : STRING[5];
        state  : STRING[2];
        zip    : LongInt;
        END;
```

Because of the highly structured nature of Pascal, each field can have any data type. A field can be an array or even another record. When a record appears as a field within another record, it is said to be "nested." For example, the type `mail_list_rec` shown below contains `last_deleted_rec`, which is another record, as one of its fields.

```
mail_list_rec = RECORD
    num_recs : Word;
    last_update,
    last_mailing : STRING;
    last_deleted_rec : mail_rec;
    END;
```

Arrays of records are common data types. For example, the following data structure creates a table of the grades of students in a math class:

```
TYPE
    table_rows = 1..30;
    stu_rec = RECORD
        name : STRING [20];
        test1_score,
        test2_score,
        test3_score : Word;
        grade_pt : REAL;
        END;

VAR
    table : ARRAY[table_rows] OF stu_rec;
```

6.2.2 Accessing Record Fields

To access an individual field of a record, you give both the name of the record variable and the name of the field:

RecordName . FieldName

The result is a data object of the type declared for the field. You can use the resulting item in any context that would be valid for an ordinary variable of the same type.

Consider the following record type:

```
TYPE
    mail_rec = RECORD
        name   : STRING[20];
        street : STRING[25];
        city   : STRING[5];
        state  : STRING[2];
        zip    : LongInt;
        END;
    mail_array = ARRAY [1..max_mailing] OF mail_rec;

VAR
    my_mail : mail_rec;
    mailing : mail_array;
```

The following example assigns a string to the `name` field of the `my_mail` record variable declared above.

```
my_mail.name := 'John Doe';
```

The expression `my_mail.name` is a string variable that can be assigned a value or passed to a function, just like any other string variable. For example, the following statement displays the name:

```
Writeln( 'Addressee name is ', my_mail.name );
```

The syntax gets a little more complicated when you refer to an item within the array of records. For example, the following statement initializes the `name` field of the third array element:

```
mailing[3].name := 'Hugo Bletch';
```

Consider how Pascal analyzes this expression. You can understand any complex expression by following similar logic:

1. The symbol `mailing` is declared as an array of records.
2. The expression `mailing[3]` is therefore an individual record. Specifically, it is the third record of the array.
3. The expression `mailing[3].name` refers to the `name` field in the third record of the array. The result is a variable of type `STRING[20]`.

The data path to an object grows as you increase the levels of nesting. For example, consider the definition of the `mail_rec` type above plus the following additional statements:

```
TYPE
    MailArr = ARRAY[1..100] OF mail_rec;
VAR
    mailing_list   : RECORD
        title      : STRING;
        addresses  : MailArr;
        END;
```

Given these declarations, you can access an individual name field as

```
mailing_list.addresses[5].name
```

The next section shows a technique for shortening the length of such expressions.

6.2.3 Using the WITH Statement to Access Fields

While defining the full data path in a record makes your code more readable, it also increases the length of the identifiers. The QuickPascal **WITH** statement enables you to omit the name of a record variable from a block of statements. It has the following syntax:

WITH *RecordName* **DO** *Statement*

The *Statement* following **DO** can refer to fields in *RecordName* directly. For example, you can assign a string to the `name` field of record variable `my_mail` with the following statement:

```
WITH my_mail DO
    name := 'John Doe';
```

To follow the **DO** keyword with more than one statement, use a **BEGIN...END** statement block.

In the case of nested records, the **WITH** statement may contain a record name modified by a field name. Always specify the outermost record first. For example, suppose `name` is a field of `mail_rec`, which in turn is a field of the record variable `mail_list_rec`. The following statement assigns a string to the `name` field:

```
WITH mail_list_rec.mail_rec DO
    name := 'Hugo Bletch';
```

6.2.4 Constant Records

To declare a constant record, first define the record type, then declare the record and the initial value in a **CONST** statement. The initial value of a record consists of the following syntax:

(*FieldName* : *Constant*; *FieldName* : *Constant* ...)

The initial value for each field follows the rules for its type. For example, each row of an array initializer must be enclosed in parentheses. In the case of nested records, additional levels of parentheses and fields are required, as shown below.

```
TYPE
    complex = RECORD
        x_real,
        y_imag : Real;
        END;
    square_matrix = RECORD
        mat_size    : Byte;
        imag        : complex;
        determinant : Real;
        mat_x       : ARRAY[1..3, 1..3] OF Real;
        END;
CONST
    origin  : complex = ( x_real : 0.0; y_imag : 0.0 );
    def_mat : square_matrix = (
        mat_size    : 3;
        imag        : ( x_real : 1.0; y_imag : 1.0 );
        determinant : 0.0;
        mat_x       :
            ( (1.0, 1.0, 1.0),
              (2.0, 2.0, 2.0),
              (3.0, 3.0, 3.0)
            )
        );
```

6.2.5 Assigning Records to Record Variables

Pascal supports the use of the assignment operator (:=) to assign the value of one record variable to another record. You can also assign one array to another if you declare the array type as a field within a record.

For example, if `rec_a` and `rec_b` are both arrays of the same type, you can use the following statement to assign one record to another:

```
rec_a := rec_b;
```

As a further example, assume `matrix1` and `matrix2` are both records of the same type, and that this type contains an array field `mat_x`. You can use the following statement to assign one array field to another:

```
matrix1.mat_x := matrix2.mat_x
```

6.2.6 Using the Debugger with Records

The Debug window displays the fields of a record as a list of data items enclosed in parentheses. Appending ", R" to the name of a watched record when it is put into the Debug window displays the names of the fields and their values. A colon separates the name of a field and its current value with this display format. You may need to scroll through the window to see all of the fields in the record.

6.3 Variant Records

A variant record lets you provide a variety of data formats for the same area of memory. Whenever you refer to the variant record, you indicate which format to use. This capability is useful in the following situations:

- You need flexibility within a general type of record. Suppose you want to create a record type to store data on each employee in a company. You'll want certain common fields, such as name, to apply to every employee. However, you may need to store different kinds of information on different kinds of employees.

  Variant records provide the needed flexibility. Instead of including all possible fields for all possible employees in every record (which would waste memory), you can create fields specific to each subgroup of employees.

- You want to read data in one way and write it out in another way. While this situation is comparatively rare, it offers a perfect use for variant records. For example, you can simulate the behavior of microprocessors which permit the same register to be accessed either a byte at a time or a word at a time.

6.3.1 Declaring Variant Records

The syntax for a variant record declaration is:

RECORD
FieldDeclarations
CASE [[*Tag*:]] *TagType* **OF**
CaseDeclarations
END

The *FieldDeclarations* make up the "fixed fields" of the record and are optional. The *Tag* is an optional field that you can use to indicate which of the cases (data formats) is active. Each *CaseDeclaration* has the following format:

CaseLabel: (*FieldDeclarations*)

Separate each *CaseDeclaration* with a semicolon. The *Tag* and each *CaseLabel* must be of the *TagType*, which can be any valid ordinal type (such as an integer or user-defined type).

Each case declares a different series of fields. However, each case is overlaid in the same area of the record. Pascal allocates enough memory for the case that has the largest total size. This area is called the "variant portion" of the record. One case might use this area to store a string field. Another might use it to store floating-point fields.

The following example shows a variant record that models the registers of the 8086 processor:

```
TYPE
    regtype = (reg16, reg08);
    registers86 = RECORD
        CASE Integer OF
        0 : ( ax, bx, cx, dx, si, di, bp, sp, flags : Word; );
        1 : ( al, ah, bl, bh, cl, ch, dl, dh : Byte; );
        END;
```

This type definition allows you to access the same data either as one 16-bit item, such as `ax`, or as two 8-bit items, such as `al` and `ah`.

6.3.2 Accessing Variant Record Fields

You access all of the fields in the record the same way, regardless of whether they are fixed fields or defined within the variant portion of the record. (Consequently, the names of all fields in both the fixed and variant portions must be unique.) In addition, if a tag field is declared, you can set it to indicate which case is active.

For example (assuming the declaration of `registers86` at the end of the last section), the following statements load data into `al` and `ah`, and then display `ax`. Note that `ax` contains the same data as `al` and `ah` combined.

```
VAR
    my_regs : registers86;
BEGIN
    my_regs.ah := $FF;
    my_regs.al := $10;
    Writeln ('AX now contains', my_regs.ax);
```

For another example, a simplified variant record scheme for the personnel system of a company might look like this:

```
TYPE
    clearance = (topsecret, secret, medium, low, known_spy);
    drink_type = ( martini, wine, champagne, teetotaler );
    games_type = ( tennis, squash, golf );
    title = ( secretary, engineer, exec );

    emp_rec = RECORD
        name : STRING[20];
        CASE job : title OF
        secretary : ( wpm, steno : Word );
        engineer : ( security : clearance; IQ : Byte );
        exec : ( beverage : drink_type;
                 pastime : games_type;
                 washroom_code : LongInt );
        END;
VAR
    new_emp : emp_rec;
```

The following code initializes the `new_emp` record variable for a recently hired engineer:

```
new_emp.name := 'Jane Eyre';
new_emp.job := engineer;
new_emp.security := secret;
new_emp.IQ := 120;
```

Now suppose that Jane Eyre is promoted into management. The old data relevant to engineers (security clearance and IQ) is no longer relevant to Jane in her new position. Fortunately, the variant record type lets you reuse this same area of memory to put in data relevant to managers.

First, the program makes Jane's promotion official:

```
new_emp.job := exec;
```

The program then enters Jane's new data. To accommodate Jane's rise up the corporate ladder, the program records her favorite drink, favorite pastime, and key code for the executive washroom. This information is overwritten onto the engineer data, which is no longer needed.

```
new_emp.beverage := martini;
new_emp.pastime := tennis;
new_emp.washroom_code := 12345007;
```

CHAPTER 7 Units

Units extend the usefulness of QuickPascal by giving your programs access to additional declarations and procedures. A "unit" is defined as a related collection of such declarations and procedures. Referencing a unit's name at the beginning of a program gives you access to that unit's contents just as though they were a part of standard Pascal. QuickPascal comes with several predefined (or "standard") units, and you can also write your own.

This chapter explains both how to use QuickPascal standard units and how to create your own units.

7.1 Understanding Units

If you've done much programming in another language, you're probably familiar with libraries. Libraries usually provide specialized routines not normally part of the language. Pascal units work on a similar concept but have a broader scope and application. They contain not only procedures and functions but also variables, constants, and type declarations. In addition, a unit can execute some code when the program starts and do anything a program can, like open files or get user input. These are important differences from libraries in another programming language.

Once you tell QuickPascal that your program uses a particular unit, you can access the procedures, functions, and declarations defined in that unit just as you would any standard procedure, function, or declaration.

Units effectively extend the portion of a program between the header and the main program body—the area where all the declarations appear. Instead of writing a tremendous number of declarations, procedures, and functions, you can use a unit and get the same results.

If you use some of your own procedures repeatedly, you can create a custom unit and make the procedures available to all your programs. And, rewriting large programs to use units removes some of the constraints of the 64K segment limit. Each unit gets its own segment, so creating just one unit doubles your programming space. Programs can use up to 81 units.

QuickPascal standard units broaden your programs' general capabilities; the custom units you write provide any specific functionality you need.

7.2 Using Units in a Program

You access a unit by adding the **USES** statement and the unit's name under the **PROGRAM** declaration. For example, the lines

```
USES
    Crt, Dos;
```

let you use any of the procedures defined in either the `Crt` or `Dos` units in your program, just as you would any local procedure. Any variables, constants, or types declared in the unit are available to your program. (If you write programs that need many variables or constants, declaring them in a custom unit can simplify your main program and make debugging easier.)

Here's a more complete example that shows a unit in context:

```
PROGRAM lil_unit_test;
USES
    Crt;

BEGIN
    Write( 'Press any key to end' );
    REPEAT  UNTIL KeyPressed;
END.
```

Once you put the unit in the **USES** list, you're free to use its contents just like any other Pascal procedures, functions, and declarations.

7.3 Standard QuickPascal Units

QuickPascal comes with several standard units. They enhance your control over the screen and text, DOS routines, printer use, and graphics. QuickPascal automatically inserts one other standard unit, **System**, into every program you compile. It supplies all of the standard procedures and functions.

QuickPascal standard units include the following:

Unit name	Description
Crt	Handles keyboard input, screen/window management, color selection, and cursor control.
Dos	Manipulates DOS files and directories, and performs other DOS functions.
Printer	Gives programs access to a printer.
MSGraph	Creates graphics and displays text in different sizes and type styles.
System	Contains the run-time system including the QuickPascal standard procedures and functions. This unit is automatically used by all QuickPascal programs; you do not need to explicitly declare it.

If you consistently use procedures not available in the standard units, consider creating your own unit. Like a program, a custom unit can reference other units. Both types of units—standard and custom—add power and flexibility to your programs.

7.4 Creating Your Own Units

QuickPascal standard units supply most of the procedures you commonly need. Yet you may need a special function repeatedly or want to initialize a large number of variables, constants, or data types. Or you may wish to share procedures with other programmers. In any case, a convenient solution is to write your own unit.

7.4.1 Writing a New Unit

QuickPascal units look similar to other Pascal programs, with the following differences:

- Units begin with a different keyword.
- An **INTERFACE** section indicates what the program using the unit can access.
- An **IMPLEMENTATION** section defines any procedures and functions in your unit.

Beginning a Unit

Just as any program begins with the keyword **PROGRAM**, every unit begins with a **UNIT** keyword:

```
UNIT new_unit;
```

The **UNIT** keyword tells QuickPascal to compile the file into a unit rather than an executable program. The compiled unit receives the same name as its source file, with the .QPU extension added. Unit source files usually keep the .PAS extension.

Interface Portion

Everything in the next section, the **INTERFACE** portion of the unit, becomes "public" information to any programs that use the unit. It includes any units needed by this unit; the declarations of any public variables, constants, or data types; and the calling format for any procedures or functions (including declarations of all parameters). The actual code for the procedures and functions comes later.

Consider the following sample **INTERFACE** section:

```
INTERFACE  { Public information }

USES
    Crt;
CONST
    days_per_year = 365;

PROCEDURE center_line( message : STRING );
FUNCTION cube( num : Real ) : Real;
```

These lines tell QuickPascal that this unit uses the standard **Crt** unit, establishes the constant `days_per_year`, and makes one procedure (`center_line`) and one function (`cube`) publicly available.

Implementation Portion

The second part of a unit, the **IMPLEMENTATION** portion, defines the procedures given in the **INTERFACE** portion and any data necessary to implement those procedures. This data may include any new definitions of variables, constants, types, labels, and support procedures (procedures that assist the primary procedures). The complete code for every **PROCEDURE** or **FUNCTION** declared in the **INTERFACE** section must appear in the **IMPLEMENTATION** section.

The following **IMPLEMENTATION** section includes both a new variable and a support function:

```
IMPLEMENTATION
{ Private information available only to new_unit }

VAR
    back : Word;

FUNCTION  cube( num : Real ) : Real;

VAR
    temp : Real;
BEGIN
    temp := Sqr( num );
    cube := temp * num;
END;

FUNCTION calc_start( message : STRING ) : Integer;
BEGIN
    calc_start := (80 - Length( message )) DIV 2;
END;

PROCEDURE center_line( message : STRING );
BEGIN
    GotoXY( calc_start( message ), WhereY );
    Writeln( message );
END;

{ Initialization part }
BEGIN
    Writeln( 'What background color would you like ?' );
    Write( '(0..7) :' );
    Readln( back );
    TextBackground( back );
    ClrScr;
END.
```

In QuickPascal, **VAR**, **CONST**, **PROCEDURE**, and **FUNCTION** declarations may come in any order. However, you must declare supporting procedures and functions—in this case the `calc_start` function, called by the `center_line` procedure—prior to calling them from within another procedure. Be sure to include the definition for every procedure listed in the **INTERFACE** portion of the program. Omitting any definition causes an `Unsatisfied forward reference` error.

Although the compiler does not require you to list each procedure's parameters (as it did in the **INTERFACE** portion), it is generally considered good programming practice to do so. Keep in mind that if you do repeat the parameters, they must exactly match the parameters listed in the **INTERFACE** portion.

Viewing the Entire Unit

Together, the three previous examples form a complete unit. A program that uses `new_unit` gets access to the constant `days_per_year`, the procedure `center_line`, and the function `cube`. However, the program cannot access the `calc_start` function or the `temp` variable because they appear in the **IMPLEMENTATION** section; that section remains entirely "hidden" to the program using the unit.

When you finish writing a unit, save the source code with the unit's name and the .PAS extension. The next step, compiling the unit, is discussed in the following section.

7.4.2 Compiling a Unit

You compile a unit in the same manner you compile a program. Choose the Compile File command on the Make menu. If your code contains errors that prevent the unit from compiling, QuickPascal displays an error message. When you remove the message, the cursor appears at the point QuickPascal detected the error. Correct the error and repeat the process until the unit compiles.

Alternately, you can compile the main program and a new unit in one step by choosing the Rebuild Main File command on the Make menu (or specify the /R option if you compile from the QPL command line). Again, if QuickPascal encounters any errors, the compilation halts and an error message appears. You must compile all of the custom units a program references before you can compile the main program.

Once your unit compiles properly, you can use it in any of your programs just as you would use any of the standard units. If you change a program that uses a custom unit, you need to recompile only that program (and not the unit). However, if you change the unit itself, you may need to recompile both the unit and the program. The Build Main File and Rebuild Main File commands on the Make menu simplify this task.

The Build Main File command recompiles all the source files that changed since the last Build Main File command was executed. Rebuild Main File recompiles the program and all of its associated custom units, regardless of whether anything changed. See the QP Advisor or "Compiling a Multiple-Module Program" in Chapter 3 of *Up and Running* for a step-by-step approach to compiling.

7.4.3 Tips for Programming with Units

When you program with units, avoid using identifiers with the same name in different units and circular references between units.

Identifiers with the Same Name

When two or more units contain identifiers (such as constant or procedure names) that share the same name, the identifier "belongs" to the unit most recently used. QuickPascal uses units in the order they appear in the **USES** statement.

To avoid any confusion, you can reference an identifier in a manner similar to referencing fields in records using this syntax:

UnitName. Identifier

For example, if you have a `ClrScr` procedure called SCRSTUFF.QPU, you can reference it in your code as `ScrStuff.ClrScr` in a unit, and the standard **ClrScr** can be referenced as `Crt.ClrScr`. Then the order of the units in the **USES** declaration becomes irrelevant.

Circular Referencing

QuickPascal permits two units to reference each other. UNIT_A can use UNIT_B, and UNIT_B can use UNIT_A. However, QuickPascal does not support circular referencing between three or more units. QuickPascal won't compile a program in which UNIT_A uses UNIT_B which uses UNIT_C which uses UNIT_A (an A-B-C-A loop). Careful planning can avoid problems with circular referencing.

The order of the units listed after the **USES** declaration is not important unless their procedures have the same names, or the initialization code in one unit interferes with the initialization code in another unit.

PART 2

Programming Topics

PART 2

Programming Topics

Part 2 of *Pascal by Example* assumes you either have read Part 1, "Pascal Basics," or already know Pascal's basic concepts. Part 2 covers more advanced programming concepts that give your programs additional flexibility. Topics include enhancing keyboard and screen control, saving and extracting data from files, and using pointers. The final chapter is devoted entirely to advanced topics that experienced programmers may find particularly interesting.

While Part 1 was designed to be read sequentially, this part is much more topical. You may read the chapters in any order. If you're using *Pascal by Example* to learn Pascal, however, consider reading the chapter on text files before you try working with binary files.

CHAPTERS

CHAPTER 8

8 The Keyboard and Screen

Virtually all programs receive or send out information in some way, perhaps accepting it from the keyboard or a file, perhaps printing it to the screen or a printer. QuickPascal supports all of the input and output features of standard Pascal and includes a unit specifically designed to enhance your programs' appearance and ease of use.

This chapter contains two parts. The first covers the basics of Pascal input and output and how to format output. The second part introduces the **Crt** unit and how to use it to refine control of both the keyboard and the screen.

8.1 Basic Input and Output

Pascal provides the **Read** and **Readln** procedures for input and the **Write** and **Writeln** procedures for output. These procedures can handle both single data items and lists of items separated by commas. All four use similar syntax:

Read([[*FileName*,]] *InputVariable*) [[, *InputVariable...*]]

and

Write([[*FileName*,]] *OutputVariable* [[: *Width*[[:*Decimals*]]]
[[,*OutputVariable* [[:*Width*[[:*Decimals*]]]]])

where *FileName* optionally refers to a disk file or device (such as a printer or a modem), and *Width* and *Decimals* refer to formatting information. Without *FileName*, **Read** accepts data from the standard input device (usually the keyboard) and **Write** sends data to the standard output device (usually the screen).

If either **Read** or **Write** encounters an error, the program terminates with a run-time error message. You can include the {$I–} compiler directive in your program if you want your program to continue despite **Read** or **Write** run-time errors.

If you choose to use the {$I–} compiler directive, you need to call the function **IOResult** immediately after each input or output operation. The value returned by **IOResult** indicates whether an error occurred. QuickPascal ignores any input or output attempts after an error, unless you call **IOResult**.

Sections 8.1.1 and 8.1.2 cover the basic use of the **Read** and **Write** statements. Section 8.2 goes into the formatting details (of *Width* and *Decimals*). For further information on using **Read** and **Write** with data files and other devices (by using *FileName*), read Chapter 9, "Text Files," and Chapter 10, "Binary Files."

8.1.1 Read and Readln Procedures

The **Read** and **Readln** procedures place data into one or more "input variables." Most commonly, **Read** and **Readln** receive their input from the keyboard or a data file.

The input variables must belong to one of the predefined data types (except Boolean) or a subrange type you defined, and they must be of a type that can accept the kind of data you're trying to read. For example, an input variable of type **Char** cannot accept a **Real** value. See Chapter 2, "Programming Basics," for information on assigning data types.

Readln attempts to read data into the list of input variables. If the number of data items typed on a single line exceeds the number of variables, the additional data items are ignored. However, if **Readln** receives fewer data items than the number of input variables, it expects the additional data to appear on subsequent lines. For example, the **Readln** statement in the following excerpt expects three values:

```
Writeln( 'For the investment what are: present value,' );
Writeln( 'annual return, and number of years invested?' );
Readln( present_value, rate_of_return, years_invested );
```

Suppose you run the program and in response to the prompt

```
For the investment what are: present value,
annual return, and number of years invested?
```

you type

```
3000.00 1.05
```

and press ENTER. The program will then wait for you to enter the final value. You must remember to use spaces, tabs, or carriage returns to separate the values. Using commas causes a run-time error. If you're concerned about the possibility of such errors, you can either write the program with one **Readln** statement for each item entered, or read the data as a string of text and create a procedure to separate the string into numbers.

The **Read** procedure, in contrast, reads only enough data to fill its input variables. But subsequent **Read** or **Readln** procedures may get the remaining data.

Suppose you rewrote the example above as

```
Writeln( 'For the investment what are: present value?' );
Read( present_value )
Writeln( 'the annual return?' );
Read( rate_of_return );
Writeln( 'and the number of years invested?' );
Read( years_invested );
```

and ran the program with this response

```
For the investment what are: present value?
3000.00 12.8 37
```

The program would then accept `3000.00` as `present_value` and immediately print the two remaining prompts without stopping:

```
the annual return?
and the number of years invested?
```

and set `rate_of_return` and `years_invested` equal to `12.8` and `37`, respectively. Unless your program performs some type of error checking, using such a series of **Read** statements can potentially introduce errors.

Like **Readln**, **Read** accepts all of the Pascal standard data types but inputs them differently. Character data is not delimited; **Read** assigns any kind of input to character variables, including carriage returns, spaces, and so on. Strings input with **Read** can include any combination of ASCII text except for a carriage return, which signifies the end of the string. For numeric data, **Read** ignores any leading blanks, begins with the first number or sign (+ or –), and continues until it reaches the next white-space character or carriage return.

8.1.2 Write and Writeln Procedures

The **Write** and **Writeln** procedures enable you to send variables, constants, and strings to the screen, disk files, or other output devices. The variables or constants you use in **Write** and **Writeln** may be any of the Pascal standard data types or a subrange of any of these types.

Write and **Writeln** differ only in that **Writeln** terminates the line after sending its data to the output device. Subsequent output appears on the next line.

You often use the **Write** statement to print prompts on the screen. **Writeln** displays full lines of text and is useful for leaving blank lines on the screen for aesthetic purposes.

Write is often used with **Read** for simple prompt-and-reply input:

```
Write( 'Please enter two integers separated by a space: ' );
Read( Int1, Int2 );
```

For complete lines of text and for blank lines, you might use **Writeln**:

```
Writeln( 'The Very Model of a Modern Major General' );
Writeln( '======================================' );
Writeln;
Writeln('By Shorty Bonaparte');
```

Note that the strings of text appear at the left edge of the screen.

By themselves, **Write** and **Writeln** do not provide much numerical formatting; they print numbers in a default format according to the type of variable. For example, the output from

```
real_num := 3.1415;
long_int := 1234567;
Writeln( 'A real number prints out as ', real_num,'.' );
Writeln( 'And a long integer prints as ', long_int,'.' );
```

looks like

```
A real number prints out as 3.14150000000154E+0000.
And a long integer prints as 1234567.
```

The next section explains how to change these default formats.

8.1.3 Formatted Output with Write and Writeln

The default numerical output format for real numbers is scientific notation. For integers, the default output format is a sequence of digits. And, Pascal normally prints strings as left justified. But **Write** and **Writeln** do allow you to change the default formats.

Recall that the **Write** and **Writeln** statements follow the syntax:

Write([[*FileName*,] *OutputVariable* [[: *Width*[[:*Decimals*]]]
[[,*OutputVariable* [[:*Width*[[:*Decimals*]]]]])

The two optional fields *Width* and *Decimals* control formatting. The *Width* field sets the maximum number of spaces available for printing the variable called *OutputVariable* and indicates right or left justification by its sign (positive for right justification, negative for left).

Pascal formats text and integer data in a similar fashion. These statements print out the integer `my_int` and the string `my_str` in a space six columns wide and right justified:

```
Writeln( my_int:6 );
Writeln( my_str:6 );
```

If either `my_int` or `my_str` exceeds a width of six columns, QuickPascal prints the entire integer or string with the default format.

Variables of type **Real** can also specify *Decimals*, which indicates the number of digits to appear to the right of the decimal point. You cannot specify *Decimals* for integers or strings.

The **Write** and **Writeln** procedures do not truncate strings or any type of numerical data. If *OutputVariable* is wider than *Width*, *Width* is overridden and the entire *OutputVar* is printed. For floating-point types, **Write** and **Writeln** always print the number of decimal places specified by *Decimals*, even if it means overriding the *Width* specification.

The following line right justifies a real number `realnum` and displays it with four decimal places:

```
Writeln( realnum:12:4 )
```

Provided `realnum` contains 12 or fewer digits, the last digit lies in column 12.

This example

```
int_var  := 12345;
str_var  := 'This sentence is a string.';
real_var := 9870.65434;

Writeln( '12345678901234567890123456789012345678901234567890 Ruler' );
Writeln( '         10        20        30        40' );
Writeln( int_var  : 8 );
Writeln( int_var  : 85 );
Writeln( str_var  : 35 );
Writeln( str_var  : -3 );
Writeln( real_var : -2 : 6 );
Writeln( real_var : 14 : 2 );
```

produces the following output:

```
12345678901234567890123456789012345678901234567890   Ruler
         10        20        30        40
   12345
12345
         This sentence is a string.
This sentence is a string.
9870.654340
       9870.65
```

Formatting needs vary from program to program. If you frequently format your data in a particular style, however, you may find it helpful to write a procedure that automatically follows that format.

8.1.4 DOS Redirection: Input and Output Files

A process called "DOS redirection" allows you to specify both input and output files for QuickPascal programs. An input file contains responses to all of the questions asked by the program (provided they are input with **Read** or **Readln** statements). An output file receives all of the information written by **Write** and **Writeln** statements that usually go to the screen.

8.1.4.1 Standard Redirection

Redirected programs run in a completely normal manner; they just accept their input from a file instead of the keyboard, and send their output to a file instead of the screen. You can run redirected programs only from the DOS command line. To use redirection from within the QuickPascal environment, you must first choose the DOS Shell command from the File menu. The general redirection syntax is

ProgramName [[< *InputFile*]] [[>*OutputFile*]]

You can include the *InputFile* only, the *OutputFile* only, or both. Both must be text files. If you run with just *InputFile*

ProgramName <*InputFile*

you still see the program's queries and other output on the screen. However, unless you specifically write your program to echo user input, you do not see the input data on the screen because it comes directly from the input file.

If you run *ProgramName* and include only the *OutputFile*

ProgramName >*OutputFile*

you need to know the order in which the program requires its input; all output, including prompts for input, go to *OutputFile*. DOS creates a new output file each time you run the program and specify *OutputFile*. If a file named *OutputFile* already exists, the old one is erased and a new one is created. But, if you add an additional > symbol

ProgramName >>*OutputFile*

DOS appends the new output to the end of the existing file named *OutputFile*.

For example, suppose you write a database program called `ADDRESS.EXE` that reads employee names and returns their mailing addresses. Using Quick Pascal or some other editor that saves text files, you can create a list of employee names and save it as `NAME.TXT`. Then if you enter

```
ADDRESS <NAME.TXT
```

you can print the list of employee addresses on the screen. Entering

```
ADDRESS <NAME.TXT >ADD.TXT
```

creates a new file called `ADD.TXT` that contains the employee addresses. And, if you write another input file called `MORENAME.TXT`, then the command

```
ADDRESS <MORENAME.TXT >>ADD.TXT
```

adds the additional addresses to the end of the existing `ADD.TXT` file.

8.1.4.2 The Crt Unit and DOS Redirection

The **Crt** unit overrides DOS redirection. Programs with **Crt** in their USES statements ignore input and output files.

If you want to run programs that use the **Crt** unit with input and output files, you must include the following lines in your program prior to the first input or output statement:

```
Assign( Input, '' );
Reset( Input );
Assign( Output, '' );
Rewrite( Output );
```

The **Crt** unit changes the names of the standard input and output sources; the four lines above reassign them to their original settings.

8.2 The Crt Unit

If you have read Chapter 7, "Units," you're familiar with the concept of units. The **Crt** unit is a standard unit that provides data types and procedures for keyboard input, cursor control, screen control, and windows. This section describes some of the variables and all of the procedures provided in the **Crt** unit. You can obtain more information on each of these variables and procedures through the on-line help system.

If you plan to use the **Crt** and **MSGraph** units together, be sure you read Chapter 13, "Using Graphics." QuickPascal imposes some restrictions on programs that call both the **Crt** and the **MSGraph** units.

8.2.1 Using the Crt Unit

Many of the complete sample programs presented in this manual use the **Crt** unit. To access its functions, procedures, variables, and constants in your own programs, add the statement `USES Crt;` immediately after your program declaration.

When you start a graphics program you need to tell QuickPascal how you want it to display text. You pass a particular constant to the `TextMode` procedure based on your desired text mode. The constants appear in Table 8.1.

Table 8.1 Crt Text-Mode Constants

Constant Identifier	Value	Display
BW40	0	40 by 25 monochrome text screen
BW80	2	80 by 25 monochrome text screen
Mono	7	80 by 25 monochrome text screen
CO40	1	40 by 25 color text screen
CO80	3	80 by 25 color text screen
Font8x8	256	EGA/VGA 43 lines

The **Crt** unit also establishes the color constants shown in Table 8.2. When you print text in different colors or create windows with different colored backgrounds, refer to the color by its value.

Table 8.2 Crt Color Constants

Color	Value	Color	Value
Black	0	**LightBlue**	9
Blue	1	**LightGreen**	10
Green	2	**LightCyan**	11
Cyan	3	**LightRed**	12
Red	4	**LightMagenta**	13
Magenta	5	**Yellow**	14
Brown	6	**White**	15
LightGray	7	**Blink**	128
DarkGray	8		

In addition to the constants, the **Crt** unit exports the variables listed in Table 8.3. They let your programs access status and informational settings used by Quick-Pascal, such as the previous video mode or current text attributes.

Table 8.3 Variables Provided by the Crt Unit

Variable	Data Type	Purpose
CheckBreak	**Boolean**	Indicates the state of CTRL+BREAK checking
CheckEof	**Boolean**	Enables/disables checking for end-of-file character on keyboard input
CheckSnow	**Boolean**	Enables/disables "snow checking" for the screen
DirectVideo	**Boolean**	Enables/disables direct video output
LastMode	**Word**	Saves previous video mode
TextAttr	**Byte**	Contains the current text display attribute
WindMin	**Word**	Saves the previous coordinates of the upper left corner of the active window
WindMax	**Word**	Saves the previous coordinates of the lower right corner of the active window

The variable **DirectVideo** plays an important role in speeding up screen output. Setting **DirectVideo** to **True** sends the output of **Write** and **Writeln** statements directly to your screen's memory. Unfortunately, this technique does not work on some computer configurations. Setting **DirectVideo** to **False**, on the other hand, employs your machine's Basic Input Output System (BIOS). BIOS always displays correctly but works at a slower speed; experiment to see which setting works better for your computer.

The **WindMin** and **WindMax** variables store the coordinates of the active window. Use the predefined **Hi** and **Lo** functions to read the high and low bytes that are packed in the **WindMin** or **WindMax** variables, as shown below:

```
UpperLeftX  := Lo(WindMin) + 1;
UpperLeftY  := Hi(WindMin) + 1;
LowerRightX := Lo(WindMax) + 1;
LowerRightY := Hi(WindMax) + 1;
```

Table 8.4 lists all of the **Crt** procedures and functions; by browsing through it you'll get a feel for the **Crt** unit. The "Purpose" column describes what each procedure does. On-line help gives a more complete explanation, and the sample programs show other uses of these procedures.

Table 8.4 Procedures and Functions Provided by the Crt Unit

Procedure	Purpose	Example
AssignCrt	Associates a text-file variable with the screen	`AssignCrt(DataFile);`
ClrEol	Clears to the end of the line	`ClrEol;`
ClrScr	Clears the window and places the cursor at the upper left corner	`ClrScr;`
Delay	Delays the program for a specified number of milliseconds	`Delay(1000){1 sec.};`
DelLine	Removes the line at the current cursor location	`DelLine;`
GotoXY	Moves the cursor to the specified window coordinates	`GotoXY(40,25);`
HighVideo	Displays characters in high intensity	`HighVideo;`
InsLine	Inserts an empty line at the current cursor location	`InsLine;`
KeyPressed	Returns **True** if there is a character in the keyboard buffer	`WHILE KeyPressed DO ...`
LowVideo	Displays characters in low intensity	`LowVideo;`
NormVideo	Restores screen attributes to those in effect when the program started	`NormVideo;`
NoSound	Turns the speaker off	`NoSound;`
ReadKey	Reads the next character from the keyboard buffer without showing it on the screen	`akey := ReadKey;`
Sound	Turns the speaker on	`Sound(frequency);`
TextBackground	Selects the background color for subsequent text display	`TextBackground(Black);`

Table 8.4 *(continued)*

Procedure	Purpose	Example
TextColor	Selects the foreground color for subsequent text display	`TextColor( Red );`
TextMode	Selects the display mode	`TextMode( BW80 );`
WhereX	Returns the X coordinate of the cursor location	`posx := WhereX;`
WhereY	Returns the Y coordinate of the cursor location	`posy := WhereY;`
Window	Defines a new display window	`Window( 2,2,40,10 );`

8.2.2 Character Input

You can read characters from the keyboard in a couple of different ways. In the case of the **Read** and **Readln** procedures, as each letter is typed in, it is put into the input variables and displayed on the screen. The **Crt** function **ReadKey** also reads input from the keyboard, but can read only one character at a time; it does not display the character on the screen.

Although you may want to do your primary data collection with **Read** and **Readln**, the **ReadKey** function lets you add finishing touches to your programs. For example, the following line pauses program execution until the user presses ENTER:

```
REPEAT UNTIL ReadKey = Chr( 13 );
```

ReadKey also reads function keys, cursor-control keys, control keys, and alternate keys. These special keys send a sequence of characters. The first is a null character (that is, ASCII 0). The second character is an extended key code. (See Appendix A for a table of extended key codes.)

The following excerpt illustrates a typical character processor for a keyboard-driven program. It checks for the special character code `#0` and the standard alphanumeric keys. If the first key is special, a second **CASE** statement identifies which key is pressed. Once the character processor determines the key pressed, it invokes the applicable procedure.

```
VAR
    done : Boolean;
    ch, ch2   : Char;

BEGIN
    done := False;

    REPEAT
        ch:= ReadKey;
        CASE ch OF
            #0: BEGIN
                ch2:= ReadKey;
                CASE ch2 OF
                    59:     { Got F1 key }
                        DoF1;  { F1 procedure }
                    .
                    .
                    .
                    END;{ CASE ch2 }
                END;{ #0 }
            '0'..'9',
            'A'..'Z',
            'a'..'z':
                DoAlphaDigit( ch );  { Procedure for normal keys }
            ELSE
                done :=True;
            END;
    UNTIL done;
```

The CRT1.PAS program in the QuickPascal Advisor demonstrates how to detect special keys using **ReadKey**. You can compile and run the program, which does the following:

- Moves the `'o'` character on the screen when you press the UP, DOWN, LEFT, and RIGHT arrow keys.
- Simulates a bouncing ball effect if you press the F2 function key.
- Hides the cursor.
- Exits when you press the F1 function key.

If you don't like the sound effects, set the constant `music` to `false`.

In the CRT1.PAS program, the `IF` statement

```
IF c1 = #0
```

becomes true when you press a special key. In that case, **ReadKey** returns two characters, the first of which is always ASCII `#0`. The path taken through the **WHILE** loops depends on which key you press.

The `cursor_off` procedure calls BIOS to turn off the cursor. The **Crt** procedures **Sound** and **Delay** are also used in this program. On-line help has more information about these two procedures.

8.2.3 Cursor and Screen Control

The **Crt** unit provides several procedures for screen and cursor control. Their names reflect their actions: **DelLine**, **HighVideo**, **GotoXY**, and so on. For more information about a specific procedure, use the QuickPascal Advisor help.

This section refers you to two example programs in the QP Advisor. The program CRT2.PAS uses **Crt** procedures to insert and delete lines, change video intensity, and produce sound effects. The program requires very few code lines to accomplish these tasks.

The CRT3.PAS program uses the **GotoXY** function to control the cursor. Table 8.6 shows the **GotoXY** statements used to move the cursor relative to its current position, assuming the cursor is not at the edge of a window.

Table 8.6 Statement Effects

Example	Description
`GotoXY( WhereX-1, WhereY )`	Moves one character to the left
`GotoXY( WhereX+1, WhereY )`	Moves one character to the right
`GotoXY( WhereX, WhereY-1 )`	Moves up one line
`GotoXY( WhereX, WhereY+1 )`	Moves down one line
`GotoXY( 1, WhereY )`	Moves to the beginning of the current line
`GotoXY(80, WhereY)`	Moves to the end of the current line

The CRT3.PAS program prompts you to type a character and then "bounces" that character around the screen. The character changes direction and beeps when it reaches the edge of the screen. The character's speed increases each time the character changes direction. When it reaches a maximum speed, the character slows back down to its original speed.

8.2.4 Using Windows

The **Crt** unit provides easy control of text windows. The **Window** procedure allows you to define a new active area of the screen.

Window accepts row and column coordinates for the upper left and lower right corners of the new window. These coordinates must be integers of type **Byte**. The rows range from 1–25 (or 1– 43 or 1–50, depending on the text mode) and the columns range from 1–80.

The **Window** procedure is analogous to choosing the size of a piece of paper to draw on; the coordinates you pass to the procedure tell QuickPascal how big to

make the piece of paper. For example, the following statement defines the entire screen area:

```
Window( 1, 1, 25, 80 );
```

Many applications use the top and bottom lines of the screen to display a menu or help text. To exclude these lines from the active screen area, use

```
Window( 1, 2, 24, 80 );
```

If you write an application that draws a frame around the screen, you may want to use the following statement to reduce the active screen area:

```
Window( 2, 2, 24, 79 );
```

When you create a new window, the upper left corner of the display area is (1, 1). Thus, **GotoXY**(1, 1) moves the cursor to the upper left corner of the active window, regardless of window size and location. **GotoXY** is analogous to setting your pen down at a specific place on the piece of paper. **WhereX** and **WhereY** return you to your current location.

Choose the foreground and background colors for the active window with the **TextColor** and **TextBackground** procedures. To continue the earlier comparison, **TextColor** lets you select your pen's color and **TextBackground** selects the color of the piece of paper. Note that after changing the background color, you must clear the screen to see the new color. Clearing the screen resets the cursor to (1, 1).

The program CRT4.PAS shows how these procedures work (although you need a color monitor to see the colors). It illustrates

- Windows that move while keeping their size fixed
- Windows that simultaneously move and change sizes
- Screen and cursor control of text within a window

CRT4.PAS gives you an idea of what you can do with windows. With a few similar lines of code, you can improve your screen's visual impact and add clarity and emphasis to your programs.

CHAPTER 9 Text Files

Text files store data as lines of ASCII characters. The lines do not have to be the same length. Each line terminates with an end-of-line character (carriage return). Any text editor or word processor that reads ASCII files can edit these files, including the QuickPascal environment.

Pascal writes and reads text files sequentially, in much the same way an audio cassette player records or plays back a tape. Adding basic file input and output (I/O) capabilities to your programs lets you store and retrieve data from this "tape" for both long- and short-term use.

This chapter covers how to name, open, read, write, and close a file, and how to redirect text information between your disk, screen, and printer. It also lists the standard procedures used to work with text files.

9.1 Working with Text Files

Working with text files means taking a few straightforward actions:

- Declaring a file variable and a file name
- Creating a new file or opening an existing one
- Writing or appending data to a file
- Reading data from a file
- Closing a file

The following sections address these steps in detail.

As a general introduction, look at the following sample program:

```
PROGRAM filetest;
VAR
    datafile : Text;
    i        : Integer;

BEGIN
    Assign( datafile, 'RAN_DATA.DAT' );
    Rewrite( datafile );

    FOR i := 1 TO 100 DO
        BEGIN
        Writeln( datafile, Random(50) );
        END;

    Close( datafile );
END.
```

The rest of this chapter frequently refers back to this example.

9.1.1 Declaring a File Variable and File Name

Declaring a file variable means telling QuickPascal how you want to refer to the file from within the program. It is the variable name by which the program knows the file. You declare it in the same way you would declare any other text variable:

VAR
FileVar **: Text**

When you later read or write information to the file, you refer to the file by the name you give *FileVar*. For example,

```
VAR datafile : Text;
```

creates a file variable with the name `datafile`.

QuickPascal also needs to know what name you want to assign to the text file that is saved on the disk. The **Assign** procedure associates the file variable with the disk file name:

Assign (*FileVar*, *FileName*)

QuickPascal accepts this assignment as meaning, "Whenever I say to read or write to the file variable *FileVar*, send the information to the disk file with the name *FileName*." You can make the two names similar, but keep in mind that *FileName* must follow the DOS file-naming conventions. For instance,

```
Assign( datafile, 'RAN_DATA.DAT' );
```

equates the file variable `datafile` with the disk file `RAN_DATA.DAT`.

For more versatility, you can use a string variable in place of a literal string such as `'RAN_DATA.DAT'`. Using a string variable allows your program to prompt for a file name. For example,

```
VAR
    datafile : Text;
    filename : String;

BEGIN
    Write('Enter name of data file to open: ');
    Readln( filename );
    Assign( datafile, filename );
    Reset( datafile );
    .
    .
    .
```

9.1.2 Opening a Text File

With the file variable and the file name both assigned, you can either create (and then open) a new file or open an existing one.

9.1.2.1 Opening a New Text File

The standard procedure **Rewrite** creates and opens a new text file. It uses the general syntax:

Rewrite(*FileVar*)

The example program at the beginning of the chapter creates and opens a new text file with the line:

```
Rewrite( datafile );
```

Since the **Assign** procedure associated the file variable `datafile` with the name `RAN_DATA.DAT`, this **Rewrite** statement creates a new file on the disk also called `RAN_DATA.DAT`.

Note that if a file already exists with the same name, **Rewrite** destroys the old file. So, it's best to use only file names you know are "safe," or have your program ask the user to confirm the name selected.

Once you open a new file, you can immediately write new text to it.

9.1.2.2 Opening an Existing Text File

You can perform both read and write operations with an existing text file. Keep in mind, however, that trying to open a nonexistent file causes a "File not found" run-time error. A short procedure that verifies the existence of the file could save you some time.

To open files for reading data, use the **Reset** procedure:

Reset(*FileVar*)

For example, to open the file named `RAN_DATA.DAT` created by this chapter's example program, you would use

```
Reset( datafile );
```

Reset opens the file and moves the "file pointer" (an internal bookmark that tells the program where it is in the file) to the first character in the file, ready to begin reading.

If you want to add text to the end of an existing file, open the file with the **Append** procedure:

Append(*FileVar*)

To append data to the `RAN_DATA.DAT` file, type in:

```
Append( datafile );
```

This opens the file and sets the file pointer to the end of the file. Text that is currently in the file remains unaltered.

Once you open a file, you can immediately read text from it or write new text to it.

9.1.3 Writing Text to a File

You write to a text file in much the same way as you write to the screen. You still use the **Write** or **Writeln** procedure and any of its standard formatting codes, but you specify the file variable as well.

In the example at the beginning of this chapter, the loop

```
FOR i := 1 TO 100 DO
   BEGIN
   Writeln( datafile, Random(50) );
   END;
```

sends `100` random integers to the `RAN_DATA.DAT` text file specified by the `datafile` file variable.

You can just as easily write text or formatted numbers to the file, but remember that even formatted numbers are stored in the file as text. Any acceptable form of **Write** or **Writeln** can send data to a text file.

9.1.4 Reading Text from a File

Use the **Read** or **Readln** procedure to read data from an open text file, specifying the file variable. For example,

```
Readln( datafile, line_o_text );
```

reads a line of text from the text file associated with the variable `datafile` into the string `line_o_text`. **Read** has the same effect but reads one variable at a time rather than an entire line.

In the example at the beginning of this chapter, with the **FOR** loop, the program creates a file called `RAN_DATA.DAT` filled with 100 random numbers between 0 and 50 (some numbers appear more than once). You could write a nearly identical program to read the data back from the file by altering the loop to

```
FOR i := 1 TO 100 DO
    BEGIN
    Readln( datafile, random_number_string );
    END;
```

Readln replaces **Writeln**, and you must declare `random_number_string` as a variable of type **STRING**. (You would also need to open the file for reading with **Reset** rather than writing.) To change `random_number_string` back into numerical data, you need to add

```
Val( random_number_string, ran_num, errpos )
```

after the **Readln** procedure. (You would also need to declare `ran_num` and `errpos` as integers.)

In cases where you don't know the length of a file in advance, you can use a loop that checks for the end of the file with the **Eof** function. For example, if you didn't know the length of the `RAN_DATA.DAT` file, you could rewrite the previous loop as

```
WHILE NOT Eof( datafile ) DO
    BEGIN
    Readln( datafile, random_number_string );
    Writeln( random_number_string );
    END;
```

Eof returns a Boolean result. It returns **False** as you read through the contents of a file and **True** after you read the file's last entry. The end-of-line function, **Eoln**, works in a similar manner, but returns True when you reach the end of a line.

9.1.5 Closing a Text File

You need to close a file when you finish working with it. All files close with the same instruction:

Close(*FileVar*)

where *FileVar* specifies an open file.

Trying to close a file that is not open causes a run-time error. However, unless you use a number of similarly named file variables, the compiler usually catches potential errors as either undefined variables (often caused by typing mistakes) or type mismatches (caused by placing a non-**Text** variable in the place of a file variable).

For example, trying to compile the program from the beginning of this chapter with the line

```
Close( datfile );  { 'datafile' misspelled }
```

results in an `Unknown identifier` compiler error.

QuickPascal automatically closes any text files still open when a program ends.

9.2 Increasing the Speed for Input and Output

The run-time system employs a buffer that temporarily collects text during **Read** and **Write** operations to text files. By default, the run-time system uses a 128-byte buffer.

Larger buffers enhance the speed of I/O-intensive programs, but tend not to affect programs with moderate or low levels of I/O activity. The larger the buffer, the greater the speed. However, unless your programs bog down due to I/O operations specifically, the default buffer size usually suffices. Keep in mind that while increasing the buffer size can speed up a program, it also increases the program's size.

The standard procedure **SetTextBuf** lets you allocate different buffer sizes. It uses the general syntax

SetTextBuf(*FileVar*, *Buffer*[[, *Size*]])

where *Buffer* refers to a variable to use as the buffer and *Size* optionally indicates the size of the buffer in bytes. You can declare *Buffer* as any type of variable, but you usually use an array of type **Char**. For example, if you wanted to increase the buffer size of the program presented earlier, you could rewrite the beginning as

```
PROGRAM filetest;
VAR
    datafile : Text;
    i        : Integer;
    buffer   : ARRAY[1..2048] OF Char;

BEGIN
    Assign( datafile, 'RAN_DATA.DAT' );
    Rewrite( datafile );
    SetTextBuf( datafile, buffer );
    .
    .
    .
```

This call to **SetTextBuf** provides a large array for intermediate storage. It's something of an overkill for a group of 100 random integers (based on the rest of the example), but would work well for reading or writing large text files.

WARNING *Buffer allocation must occur before or immediately after you open the text file. Changing the file buffer size after I/O operations have already occurred can lead to data loss.*

9.3 Redirecting Text Output

QuickPascal lets you access a number of standard DOS devices (such as a printer and the screen) by specifying the device as an output file name. For example, by reassigning the file variable, the same **Writeln** statement could send text data to a disk file, the printer, or the screen.

DOS devices use predefined names. The two most common are the printer name, PRN (assumed to connect to the LPT1 port), and the screen name, CON (short for "console"). Data sent to a device goes to the appropriate computer port.

To see how reassignment works, consider a program that, at the end of a particularly grueling data-generating session, presents the user with a choice of sending the data to the printer, console, or file. Based on the selection, the program assigns a file variable to the appropriate device or text file.

If `OutFileVar` is the file variable, choosing the printer leads to the assignment

```
Assign( OutFileVar, 'PRN' );
```

to direct the output to the printer. (You could also get the same effect by using the **Printer** unit and substituting `LST` for the file variable. **Printer** provides the `LST` text file variable already opened on the LPT1 printer port.)

Similarly,

```
Assign( OutFileVar, 'CON' );
```

and

```
Assign( OutFileVar, NewFile );
```

direct the output to the screen and a disk file, respectively. (`NewFile` must be declared as a string, and must contain the name of the new disk file.)

The file variable `OutFileVar` now refers to the correct output location, regardless of whether that output is the printer, screen, or a new text file. The program opens the device or file with

```
Rewrite( OutFileVar );
```

sends the data to the file with

```
Writeln( OutFileVar, TextOut );
```

and closes the file when finished with

```
Close( OutFileVar );
```

With some planning, the same section of program code can perform three different functions: print data, send to the screen, or send to a file. The only difference is the file name assigned to the file variable. In the example above, a CASE statement, or similar decision-making structure, would assign an appropriate filename variable based on the user's menu selection.

9.4 Standard Procedures and Functions for Input and Output

A number of the QuickPascal standard procedures and functions apply to all types of data files—text, typed, and untyped. Table 9.1 summarizes those procedures and functions available to all file types.

Table 9.1 Standard Procedures and Functions for All File Types

Routine	Purpose
Assign	Associates a file buffer with a filename
Close	Closes the file buffer
Eof	Returns the end-of-file status
Erase	Deletes a file
IOResult	Returns the error status of the last I/O
Read	Reads one or more elements from a file
Rename	Renames a file
Reset	Opens an existing file
Rewrite	Creates and opens a new file, after closing and erasing any file with the same name
Write	Writes one or more elements to a file

Several other standard procedures and functions apply only to text files. They appear in Table 9.2 below.

Table 9.2 Standard Procedures and Functions for Text Files

Routine	Purpose
Append	Opens an existing text file for adding more text to the end of the file
Eoln	Returns the end-of-line status
Flush	Clears the text buffer
Readln	Reads one or more data items, one line at a time
SeekEof	Returns the end-of-file status, ignoring any blanks, tabs, and end-of-line markers
SeekEoln	Returns the end-of-line status, ignoring any blanks and tabs
SetTextBuf	Assigns a text file I/O buffer
Writeln	Writes one or more data items and appends an end-of-line marker

CHAPTER 10
Binary Files
10

A binary file contains program data stored on disk. Each item in a binary file is stored in the same binary representation used by a QuickPascal program. Binary files provide optimal storage of numbers, Booleans, and enumerated types. For example, to store the integer 21,000 in a text file, you write a string of at least five characters to disk. To store the number in a binary file, you write just two bytes. However, you cannot display a binary file directly or view it with a word processor.

You access each binary file as either a typed file or an untyped file:

- A "typed file" contains a series of discrete units called "components." Each component must have the same type, which can be almost any data type supported by QuickPascal but is typically a record.
- An "untyped file" is treated as a raw, unstructured series of bytes. None of the text-oriented read and write functions is available. Typically, programs use untyped files for large block operations such as copying an entire file to another. Any file can be declared as an untyped file.

10.1 Typed Files

Like an array, a typed file is a series of components all having the same type. Unlike an array, a typed file has no definite size. A file starts at length zero and automatically grows as you append data. Furthermore, files serve as permanent records that exist after the program terminates.

In essence, typed files are formatted data files. The format is determined by the component type, which you should choose carefully to solve a given programming task. For example, to implement an airline reservation system, you would

set up a record type to store all of the needed data for one reservation. Then you might create a file made up of these records. Every program in the system must use this same record type to correctly read the file.

The structure of a typed file supports random access. For example, in a file of records, you can directly access record 367 without having to read through the first 366.

10.1.1 Declaring Typed Files

Not surprisingly, the syntax for defining a typed file is similar to that for arrays:

FILE OF *ComponentType*

In the syntax display above, *ComponentType* can be any valid data type, with one restriction: the component type cannot be a file type or a type that contains a file type. Thus, files of arrays are legal, as are arrays of files. However, a file of arrays of files is illegal.

The component type is frequently a record. (In fact, other programming languages often use the term "record" to refer to a component of a file.) For example, you might define a record type to hold a name and phone number for one person. To create a permanent list of phone numbers for many people, you could create a file made up of these records.

The following code shows examples of valid file types:

```
TYPE
    phonerec = RECORD
        name          : STRING[20];
        long_distance : Boolean;
        phone         : LongInt;
        END;

    phone_list = FILE OF phonerec;
    math_file = FILE OF ARRAY[1..10] OF Real;
VAR
    master_list   : ARRAY[1..20] OF phone_list;
    celebs        : phone_list;
    lucky_numbers : FILE OF Integer;
```

10.1.2 Accessing Data in a Typed File

After declaring a file variable, you may assign it to a physical disk file with the **Assign** procedure, as described in the last chapter. Then you can open the file for writing (with **Rewrite**) or for both reading and writing (with **Reset**).

For example, the following code declares a file of integers, assigns the file variable to the disk file MYFILE.DAT (in the root directory of drive C:), and then opens the file for reading and writing:

```
VAR
    intfile : FILE OF Integer;
BEGIN
    Assign( intfile, 'C:\MYFILE.DAT' );
    Reset( intfile );
```

After you open the file, you can read and write any number of components sequentially with the **Read** and **Write** procedures. (The next section shows how to use random access.) These procedures work with both text files and typed files, and take the same syntax in either case. But with typed files, each item you read or write must be a variable of the component type.

For example, if `int_file` is a file of integers, and `a`, `b`, and `c` are integer variables, you can read or write the file as follows:

```
Write( int_file, a,b,c );   { Write a, b, c to the file }
Read( int_file, n );        { Read next integer in file }
```

The **Read** and **Write** procedures do not do any text formatting when used with typed files. In the example above, the procedures read and write the numeric value of the integers directly to and from the disk. If `int_file` were a text file, the **Read** procedure would translate the numbers to character strings before writing them.

As you read components of a file, use the **Eof** function or **FileSize** function (described below) to make sure you don't read past the end of the file. The **Eof** function returns **True** if the last read operation took you beyond the end of the file.

You can use a number of other procedures with typed files, including the procedures listed in Table 9.1 in Chapter 9, "Text Files." In addition, you can use those listed below.

Procedure	Description
FilePos	Takes a file variable as a parameter, and returns the current file position (in terms of components or blocks)
FileSize	Takes a file variable as a parameter, and returns the size of the file in bytes; result has type **LongInt**
Seek	Takes a file variable and a long integer as parameters, and moves the file position to the component or block designated by the integer
Truncate	Takes a file variable as a parameter, and truncates the file at the current file position

The **FilePos** and **Seek** procedures let you treat a binary file as a random-access file, and the next section provides more detail on how to use them. Note that you do not have to treat a typed file as strictly a sequential or random-access file. You can use any combination of functions supported for the file type.

10.1.3 Using Random Access

You can use random-access procedures with any typed file. "Random access" is the capability to read or write components to any place in the file and in any order.

Random access is like placing a phone call. You can immediately connect to any place in the system by giving the right number. Sequential access is like reading a novel. You advance from one page to the next in the order given.

The two principal random-access procedures are **Seek** and **FilePos**. The **Seek** procedure sets the file buffer to the component denoted by the number you specify. The first component is denoted by 0, the second by 1, and so on. The syntax is

Seek (*FileVar*, *Position*)

in which *FileVar* is a file variable, and *Position* is a constant or variable of type **LongInt**. For example:

```
TYPE
   phone_rec = RECORD
       name,
       notes : STRING;
       number : LongInt;
       END;
VAR
    phone_list : FILE OF phone_rec;
    rec10, rec11, rec15, rec25 : phone_rec;
BEGIN
    Assign( phone_list, 'FONEHOME.DAT' );
    Reset( phone_list );
    Seek( phone_list, 9 );     { Get 10th & 11th record }
    Read( phone_list, rec10, rec11 );
    Seek( phone_list, 14 );     { Get 15th record }
    Read( phone_list, rec15 );
    Seek( phone_list, 24 );     { Get 25th record }
    Read( phone_list, rec25 );
```

The example above copies records at predefined locations in the file. More often, a practical application determines the record number interactively. For example, the following code prompts the user for the record number and data, then enters this data into the file:

```
VAR
    phone_list : FILE OF phone_rec;
    temp_rec : phone_rec;
    n : LongInt;
BEGIN
    Assign( phone_list, 'FONEHOME.DAT' );
    Reset( phone_list );
    Write( 'Enter record number: ' );
    Readln( n );
    Write( 'Enter name: ' );  { Prompt for data }
    Readln( temp_rec.name );
    Write( 'Enter number: ' );
    Readln( temp_rec.number );
    Seek( phone_list, n );  { Access record requested }
    Write( phone_list, temp_rec ); { Write data to file }
```

The **FilePos** function takes a file variable as a parameter and returns the number (again, a **LongInt**) of the current component.

The **Eof** function is useful for both sequential-access and random-access operations. This function takes a file variable as its parameter and returns **True** if the current component is past the end of the file. Thus, it tells you when you have read to the end of the file or have a record number corresponding to a nonexistent file component.

So far, you have seen how to read and overwrite existing files. You can append the end of files with

Seek(*f*, FilePos(*f*))

and rewrite files completely with the **Rewrite** procedure. But there is no easy way to insert new components into the middle of a file. The only way to insert a component is to read an entire file into memory, manipulate the contents, and write the file to disk again.

10.2 Untyped Files

Untyped file variables support direct, low-level I/O operations with any file. The **BlockRead** and **BlockWrite** functions used with untyped files allow for fast data transfer for copy and backup of files. You can also use untyped file I/O to create sequential binary files with variable-length records.

Untyped files differ from typed files in that

- Untyped files can contain any type of data, even text.
- Untyped files can be read or written with any record length using **BlockRead** and **BlockWrite**.

To declare a type or variable as an untyped file, just use the **FILE** keyword. For example,

```
TYPE
    low_level = FILE;
VAR
    my_file : low_level;
```

The **Read** and **Write** procedures, supported for use with text files and typed files, are not supported with untyped files. (Otherwise, any procedure supported for typed files is also supported for untyped files.) Instead, use the **BlockRead** and **BlockWrite** procedures to access data. **BlockRead** and **BlockWrite** read and write records to a file. In this context, "record" denotes a data block of a specific size. The default block size is 128 bytes if you use the standard file-open sequence:

```
Assign( file_var, 'FILE' );
Reset( file_var );
```

With the default block size, the **BlockRead** procedure reads in units of 128 bytes at a time. If the last **BlockRead** finds fewer than 128 bytes, an error occurs. Rarely are the contents of a file exactly equal to 128 * *n*. To avoid errors, you have two alternatives:

1. Create a file by writing records of a fixed size with **BlockWrite**. Then the file size will be exactly divisible by the size of the record.

2. Create a record size of one byte (since every file size is a multiple of one) by using the statements below:

   ```
   Assign( file_var, 'FILE' );
   Reset( file_var, 1 );
   ```

Reset and **Rewrite** have an optional parameter to define the number of bytes in a record. Once the record size is set to one byte, the procedures **BlockRead** and **BlockWrite** transfer multiples of one byte whenever they execute. No error occurs at the end of the file.

The syntax for **BlockRead** is

BlockRead(*FileVar*, *Buffer*, *Count* [[, *NumRead*]])

where **BlockRead** reads *Count* records (or the number of records remaining, whichever is less) from the file into *Buffer*. The *Buffer* parameter can be any variable large enough to hold the number of bytes read. The actual number of complete records read is returned in the optional parameter *NumRead*. Use *NumRead* to determine whether **BlockRead** was successful. If the parameter

NumRead is omitted and **BlockRead** reads fewer than *Count* records, an I/O error occurs. The parameter list of **BlockWrite** is the same as that for **BlockRead**.

The following simple program, DUPLICAT.PAS, shows a typical use of block I/O to copy a file:

```
PROGRAM duplicat;
CONST
    max_buf=16384;
VAR
    file_name, copyfile_name : STRING;
    source, target : FILE;
    buffer : ARRAY [1..max_buf] OF Char; { 16K buffer }
    bytes_read, bytes_written : Word;

BEGIN
    Write( 'Enter source file_name -> ' );
    Readln( file_name );
    Write( 'Enter name of target file -> ' );
    Readln( copyfile_name );
    Assign( source, file_name );
    Reset( source, 1 );              { 1 byte-block size }
    Assign( target, copyfile_name );
    Rewrite( target, 1 );            { 1 byte-block size }
    REPEAT
        BlockRead( source, buffer, SizeOf( buffer ), bytes_read );
        BlockWrite( target, buffer, bytes_read, bytes_written )
    UNTIL ( bytes_read = 0 ) OR ( bytes_read <> bytes_written );
    Close( source );
    Close( target );
END.
```

The program detects the end of the file by looking for either of the following two conditions:

1. No records were read by the last **BlockRead** call.
2. The requested number of records does not match the actual number of records read.

The block I/O techniques presented in the program above are used to implement an extended version of the DOS COPY command in the sample program EXCOPY.PAS, available on-line in QuickPascal. The other sample program components are the command-line arguments; the **FindFirst** and **FindNext** routines; a binary tree to detect duplicate file names; I/O error checking used with **BlockRead** and **BlockWrite**; and screen output informing the user of the file copy progress.

The EXCOPY.PAS procedure `copyfile` has the task of actually copying the files, one at a time. Notice the following aspects of the procedure:

- The **Reset** and **Rewrite** statements are accompanied by the {**$I–**} directive to prevent run-time errors from stopping the program. After calling **Reset** and **Rewrite**, the value of the function **IOResult** is compared with 0. If it is not 0, the procedure terminates. This behavior protects against errors resulting from bad file names or attempts to copy files that cannot be accessed.
- After the **BlockWrite** procedure is executed, the parameters `bytes_read` and `byte_written` are compared. If they are not equal, the destination disk becomes full while copying the current file.
- If the file cannot be copied, then a message is displayed to that effect and the target file is erased. Consequently, any partially used disk space is freed for other smaller files to be copied. In addition, the above procedure also wipes off zero-byte files that would otherwise appear in the target directory.

To run EXCOPY.EXE, first compile the program EXCOPY.PAS. The current directory should contain the source files you wish to copy. Enter the command line arguments as

EXCOPY *TargetDirectory* [[*FileList*]]

The *FileList* argument can contain one or more file names separated by spaces. Each file name can contain the wildcard characters * and ?. If you omit *FileList*, EXCOPY uses the default file specification *.* as the file list.

CHAPTER 11

11 Pointers and Dynamic Memory

A pointer is a variable that contains the numeric address of another data object. A pointer provides indirect access to data. For example, if you have a pointer to a record and you pass this pointer to a procedure, then the procedure can manipulate any field by using the pointer. The procedure does not need its own copy of the data.

In Pascal, you use pointers primarily as handles to dynamic-memory objects. "Dynamic memory" consists of memory that the program explicitly requests at run time. Dynamic memory gives you many advantages. It lets your memory usage grow and contract as your needs require—you do not need to specify a maximum size or limit.

Because dynamic memory is allocated at run time, your program cannot know in advance where the block is located. Pascal, therefore, returns a pointer when it allocates dynamic memory. The pointer provides the access to the data.

Dynamic memory enables you to create powerful data structures such as linked lists and binary trees. These structures, described at the end of this chapter, are networks of data in which pointers provide the connecting links.

In QuickPascal, you can also use pointers to point to ordinary (nondynamic) variables. This chapter begins by explaining the basics of pointers using nondynamic variables.

11.1 Declaring and Accessing Pointers

Using pointers consists of three major steps, which you must always do in this order:

1. Declare the pointer as a specific type.
2. Initialize the pointer.
3. Use the pointer by assigning its value, testing its value, or accessing the value that it points to.

11.1.1 Declaring Pointers

Like other variables, pointers have definite types and can only point to a variable of the appropriate type. You can declare a pointer with the following syntax:

PointerName : ^*DataType*

Example pointer declarations are shown below:

```
TYPE
    totals = ARRAY[1..10] OF Integer;
VAR
    int_ptr   : ^Integer;
    char_ptr  : ^Char;
    str_ptr   : ^STRING;
    real_ptr  : ^Real;
    total_ptr : ^totals;
```

After you declare a pointer, it does not point to any meaningful value; you can produce errors if you try to use it. The first thing you must do after declaring a pointer is initialize it.

11.1.2 Initializing Pointers

After declaring a pointer, you must initialize it to an address. You can always initialize a pointer to the special **NIL** value. This value indicates that the pointer is temporarily turned off—it has no object to point to. Your program can test for this condition and take appropriate actions. The **NIL** value is useful in indicating the end of a tree or linked list. Here is an example of an assignment to **NIL**:

```
my_ptr := NIL;
```

To assign the address of a variable to a pointer, you can use either the address-of (@) operator, or the **Addr** function. The syntax is

Pointer := **Addr**(*Variable*)
Pointer := **@***Variable*

The *Variable* can be any variable of the type that appears in the declaration of *Pointer*. An example is shown below:

```
VAR
    an_int   : Byte;
    byte_ptr : ^Byte;

BEGIN
    an_int := 5;

    { These assignment statements put the same
      address in pointer byte_ptr.
      Both Writeln statements print the number 5.
    }
    byte_ptr := Addr( an_int );
    Writeln( byte_ptr^ );

    byte_ptr := @an_int;
    Writeln( byte_ptr^ );
END.
```

In Section 11.2, you learn how to assign a value to a pointer by making a dynamic-memory procedure call.

11.1.3 Manipulating Pointers

Pointer manipulation in Pascal is extremely limited. In addition to the methods described above, the only way to manipulate a pointer is to assign it the value of another pointer of the same type. For example, the statement

```
ptr1 := ptr2;
```

causes `ptr1` to point to the same location that `ptr2` does. This kind of assignment is frequently useful in dealing with data structures such as linked lists (shown in Section 11.3).

Once you declare and initialize a pointer, you can use it in one of the following ways:

- Assign the value of the pointer itself to another pointer.
- Test the value of the pointer itself.
- Access the value of the variable pointed to.

The number of operations you can do with the value of the pointer is limited. As described above, you can assign the value of a pointer to another pointer of the same type. You can also test a pointer for equality to **NIL** or to another pointer. For example, the statement

```
IF (ptr1 = ptr2) THEN ...
```

executes the statement following **THEN** if `ptr1` and `ptr2` point to the same variable. Note that if `ptr1` and `ptr2` point to different locations, then the expression `ptr1 = ptr2` evaluates as **False**, even if the objects that `ptr1` and `ptr2` point to are equal.

You can also access the value of the variable indicated by the pointer. This value can be manipulated in any way you can manipulate the variable itself. Use the following syntax to access the variable indicated by the pointer:

PointerName^

This operation is called "dereferencing" the pointer. For example, the following code sets the value of `x` to `5`, and then assigns this value to `y`:

```
VAR
    x, y      : Byte;
    byte_ptr : ^Byte;

BEGIN
    byte_ptr := Addr( x ); { byte_ptr now points to x }
    byte_ptr^ := 5;        { assign 5 to x }
    y := byte_ptr^;        { assign value of x to y }
END.
```

In testing pointer values, bear in mind the difference between a pointer and the variable pointed to. For example, the statement

```
IF (ptr1^ = ptr2^) THEN ...
```

executes the statement following **THEN** if the objects pointed to by `ptr1` and `ptr2` are equal. Contrast this example with the previous **IF**-statement example, which evaluated to **True** only if `ptr1` and `ptr2` pointed to the same object.

11.2 Dynamic-Memory Allocation

In most programs, you need to evaluate the maximum amount of memory the program will require. If the amount of data becomes larger than you foresaw, you must rewrite the program and then recompile. However, dynamic memory lets your memory usage grow along with the needs of the program. The amount of physical memory available is the only ultimate limit to dynamic memory.

The use of pointers is essential to all dynamic-memory operations. When QuickPascal allocates memory, it returns a pointer. The pointer gives you access to the memory block.

There are two basic ways of dynamically allocating memory:

1. Allocating one object at a time (**New** and **Dispose**)
2. Allocating a block of memory (**GetMem** and **FreeMem**)

11.2.1 Allocating a Single Object

By using the **New** procedure, you allocate space equal to the size of the data type associated with the pointer. The syntax is

New(*Pointer*)

Once the **New** function executes, QuickPascal assigns the address of the dynamic-memory block to *Pointer*. The *Pointer* must be a variable previously declared. The following example shows how to declare a pointer of type `Byte`; allocate memory through the pointer; and then use the dynamic variable to hold, manipulate, and display a value.

```
VAR
    int_ptr : ^Byte;

BEGIN
    int_ptr := NIL;
    { create a Byte-type dynamic variable }
    New( int_ptr );
    int_ptr^ := 100;       { assign a value to the dynamic
                             variable }
    Inc( int_ptr^, 10 );   { increment it }
    Writeln( int_ptr^ );   { display its value }
    Dispose( int_ptr );
END.
```

As described in the previous section, `int_ptr^` is an example of a dereferenced pointer. `int_ptr` itself is a pointer, which can only be manipulated in a few restricted ways. However, `int_ptr^` is equivalent to an ordinary variable of type `Byte`. Use `int_ptr^` anywhere you would use a `Byte` variable.

To remove a dynamic-memory object created with the **New** function, use the **Dispose** function. See Section 11.3, "Linked Lists," for more examples of **New** and **Dispose**.

11.2.2 Allocating a Memory Block

The **GetMem** and **FreeMem** functions are similar to **New** and **Dispose**. However, **GetMem** and **FreeMem** deal with entire blocks of memory rather than one object at a time. Once a block is allocated, you access it as if it were an array of indefinite size.

Use the **GetMem** procedure to select the size of a dynamic-memory block. The size should be a multiple of the size of the element type of the array. Therefore, if `size` is the number of elements you want to allocate, and `base_type` is the element type of the array, then pass the following parameters to **GetMem**:

```
Size * SizeOf(base_type)
```

A common way to use **GetMem** is to declare an array type of `max_elements` elements first, where `max_elements` is the largest possible number of elements of the base type. Because the type is an array, you can access memory throughout the block with an array index. For example, the following code makes the necessary declarations and then calls **GetMem** to return a memory block:

```
CONST
    max_elements = 65520 DIV SizeOf( base_type );
TYPE
    big_array = ARRAY[1..max_elements] OF base_type;
VAR
    array_ptr : ^big_array;
BEGIN
    .
    .
    .
    GetMem( array_ptr, size * SizeOf(base_type) );
```

The `array_ptr` now points to an array of type `base_type`. You can treat `array_ptr` just like any array. The largest index in this array is `size`. To access any element in this array, use the following syntax:

ArrayPointer^[*Index*]

The example shown below requests a memory block `100` elements long. In this case, `array_size` is set to `100`, but at run time, the program could set `array_size` to whatever length it needed.

```
CONST
    max_elements = 65520 DIV SizeOf( Real );
TYPE
    some_reals : ARRAY[1..max_elements] OF Real;
VAR
    rptr       : ^some_reals;
    i          : Byte;
    array_size : Word;
BEGIN
    array_size := 100;
    GetMem( rptr, array_size * SizeOf(Real) );
    FOR i := 1 TO array_size DO
        BEGIN
        rptr^[i] := i;
        Writeln( rptr^[i] );
        END;
    FreeMem( rptr, array_size * SizeOf( Real ));
END.
```

The **FreeMem** procedure frees up memory blocks allocated by **GetMem**. If you no longer need to use a particular memory block, it is a good idea to free the memory. Otherwise, the program can use up all of the available memory over time. The **FreeMem** procedure takes the same parameters that **GetMem** does. Make sure that the size you specify in **FreeMem** matches the size allocated with **GetMem**.

Table 11.1 summarizes the procedures provided by QuickPascal for use with pointers.

Table 11.1 Pointer Procedures

Routine	Purpose	Example
Addr	Returns the address of a data object (same as the @ operator)	`aptr := Addr( I );`
Dispose	Disposes of a dynamic variable	`Dispose( nextptr );`
FreeMem	Disposes of a dynamic variable of given size in bytes	`FreeMem( aptr, 512 );`
GetMem	Creates a dynamic variable of a given size in bytes	`GetMem( aptr, 512 );`
New	Creates a dynamic variable	`New( aptr );`

11.3 Linked Lists

Stacks, queues, and trees are data structures that are linked lists. A "linked list" is a collection of dynamically allocated records, each having a field that is a pointer to the next record. Essentially, the pointers serve as the connectors between any two items. By altering the value of the pointers, you can sort or reorganize the list in any way—without physically moving any of the stored records.

If your program implements a straightforward algorithm, you do not need to use these data structures. However, these structures give you a great deal of power to solve complex computing tasks. They can grow to any size, and they let the program traverse, analyze, and restructure a network of data paths. The only limit to the complexity is your own imagination.

The LIST.PAS program adds records to a list, deletes them, and prints the contents of the list. The data is stored in a record declared as:

```
TYPE
    rec_ptr = ^stack_rec;
    stack_rec = RECORD
        data     : Integer;
        next_rec : rec_ptr;
        END;
```

Note that the second field of type `stack_rec` points to another record—also of type `stack_rec`. Though this self reference may seem paradoxical, it is perfectly legal. It simply means that the second field is the connector to another record of the same type. The `data` field contains the data to be stored. For more complex programs, the record could have any number of appropriate data fields.

To create a list, first declare a pointer to the start of the list and initialize this pointer to **NIL**:

```
VAR
    stack_ptr : rec_ptr;
BEGIN
    stack_ptr := NIL;
```

The program has two major procedures, `push` and `pop`. These procedures model the behavior of the **PUSH** and **POP** instructions of the processor. The linked list in this program is a last-in, first-out mechanism, just like the stack of the 8086 microprocessor. The `push` procedure adds items to the front of the list, and `pop` removes these items from the front as well. Therefore, the last item stored is also the first item retrieved.

The code in the `push` procedure inserts a new record at the front of the list and then assigns the new value (`x`) to the `data` field. These actions simulate the action of pushing `x` onto the top of a stack.

```
VAR
    temp :  rec_ptr;
BEGIN
    New( temp );
    temp^.next_rec := stack_ptr;
    stack_ptr := temp;
    stack_ptr^.data := x;
END;
```

The above lines of code show the four steps required for the `push` procedure to add a new record to the linked list. These four steps are listed below:

1. The first statement, `New(temp)`, allocates a memory location large enough to hold a record with the fields `data` and `next_rec`. The pointer `temp` now points to this new record.

2. To insert this record at the front of the list, the code reassigns two pointer values. First, the procedure sets the `next_rec` field to point to the current item at the front of the list. The pointer variable `stack_ptr` points to the front of the list, so the following line of code assigns the value `stack_ptr` to the `next_rec` field of the new record (the new record is referred to as `temp^`).

   ```
   temp^.next_rec := stack_ptr;
   ```

3. Next, the pointer `stack_ptr` must be reassigned to `temp`. The result is that the item previously at the front of the list is now the second item (because of step 2), and the new record is at the very front.

4. Now that the new record has been created and inserted, you can simply load the new data into the record. The following statement assigns the value of `x` to the `data` field of the new record. Note that because of step 3, the new record can be referred to as `stack_ptr^`.

   ```
   stack_ptr^.data := x;
   ```

Note that `temp` still points to the new record, but now `temp` can be ignored because `stack_ptr` also points to this record.

Figure 11.1 illustrates the `push` procedure.

The `pop` procedure works by executing the series of steps in reverse, as shown in the code below:

```
z := stack_ptr^.data;
temp := stack_ptr;
stack_ptr := stack_ptr^.next_rec;
Dispose(temp);
```

1. The procedure pops the value off the top of the stack by saving the value in the `data` field of the first record. The following statement saves this value by loading it into `z`, the output value of the procedure.

   ```
   z := stack_ptr^.data;
   ```

2. Note that the current record at the front of the list must be deleted. The pointer `temp` points to this record, so that the procedure can use `temp` to delete the record later on.

   ```
   temp := stack_ptr;
   ```

3. Then the pointer to the top of the stack, `stack_ptr`, is moved so it points to `stack_ptr^.next_rec`. Note that `stack_ptr` points to the top record in the list, which is a record with two fields: the `data` field and the `next_rec` field. The `next_rec` field in that top record currently points to the next record in the list. The statement

   ```
   stack_ptr := stack_ptr^.next_rec;
   ```

   moves `stack_ptr` to point to the same record that `stack_ptr^.next_rec` is pointing to. By assigning `stack_ptr` to point to the same record to which the top record was pointing, there is now no pointer in the list pointing to the record that was on top. The record has been removed from the list.

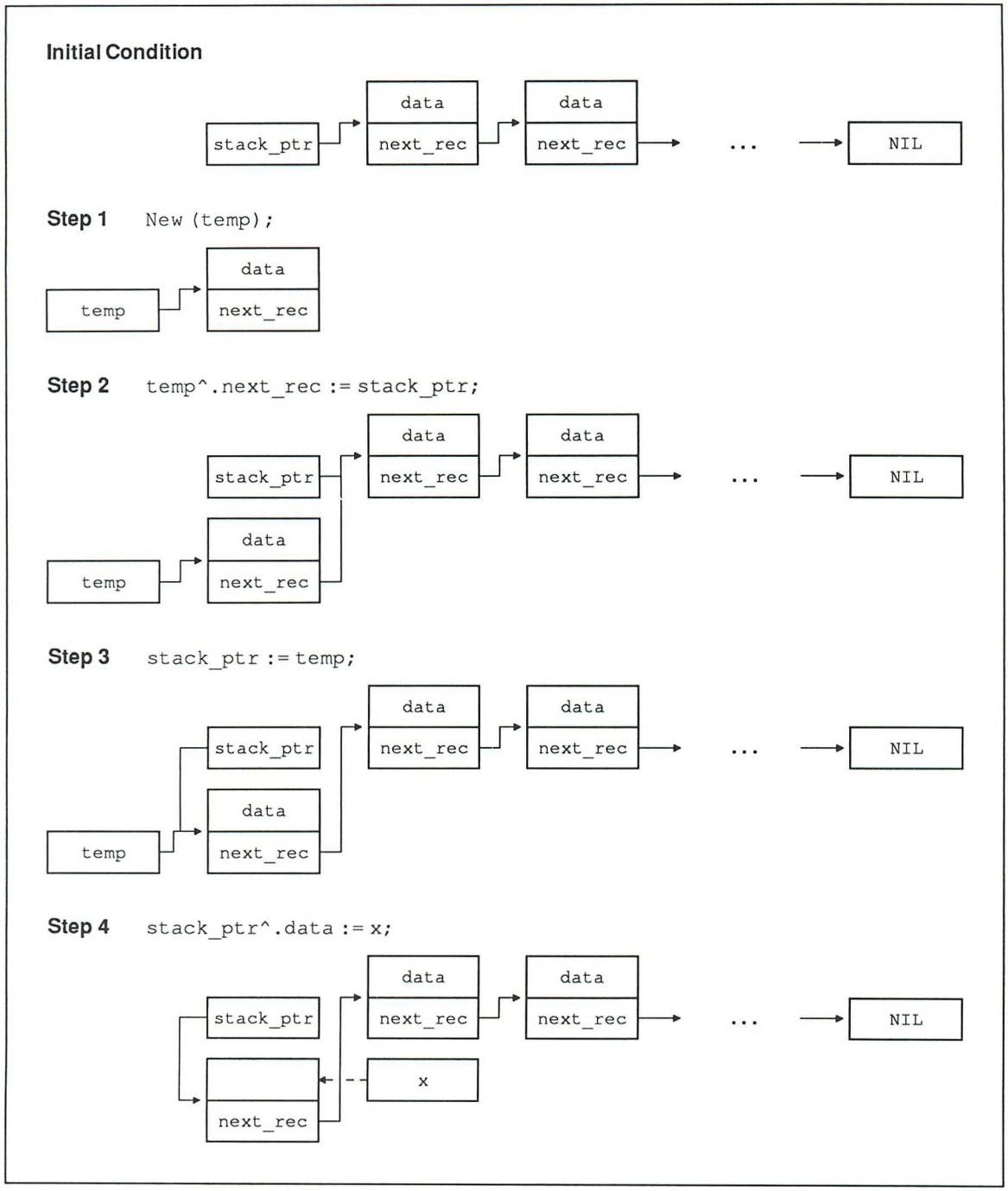

Figure 11.1 The Push Procedure

4. The old record at the front of the list, pointed to by `temp`, is now deleted from memory.

```
Dispose( temp );
```

Figure 11.2 illustrates the `pop` procedure.

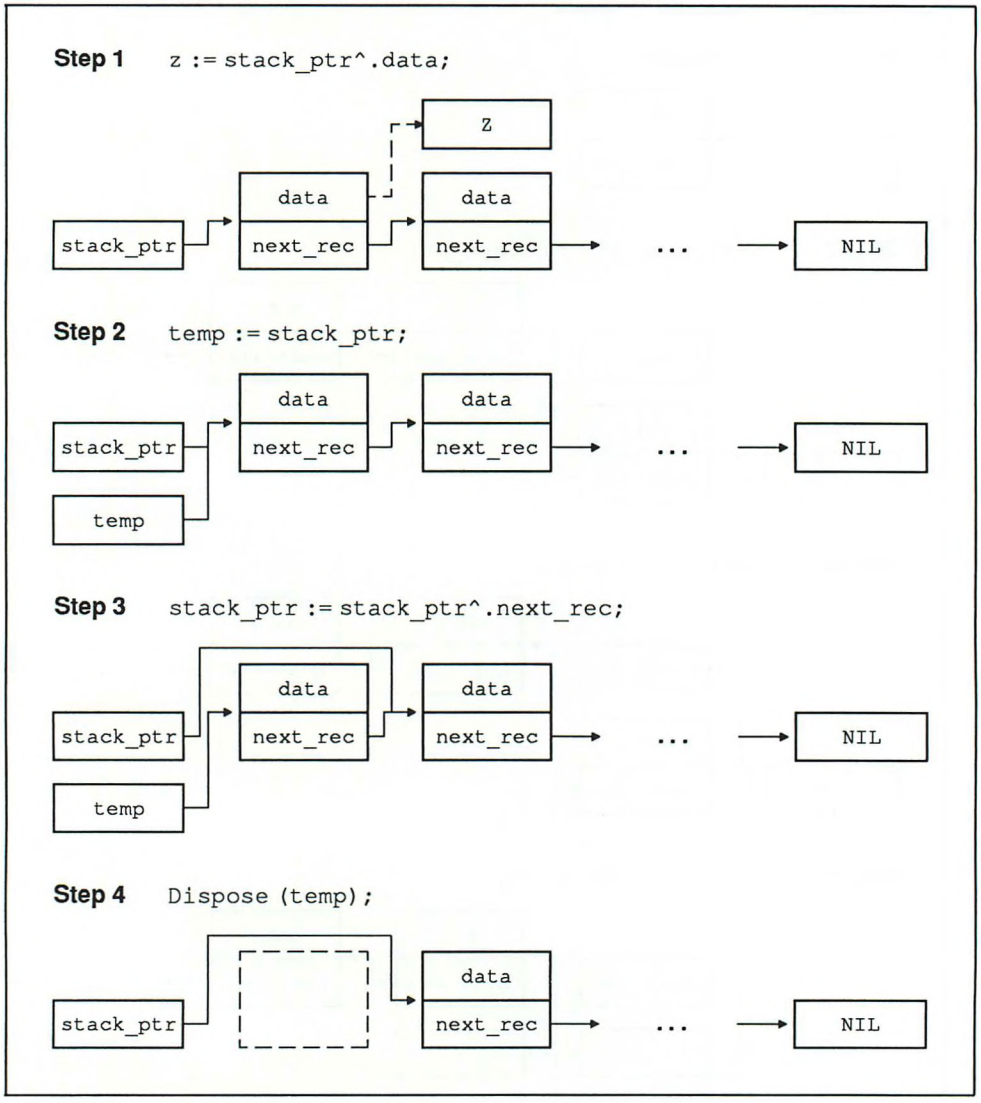

Figure 11.2 The Pop Procedure

11.4 Binary Trees

Binary trees are one of the many types of tree structures that can be created with pointer variables. Binary trees are more complex than the linked lists described in the previous section. Each record (called a "node") has not one, but two pointers. Each of these pointers connects the node to two other nodes (called "children")—one on its left and one on its right.

Figure 11.3 shows a sample binary tree. As you can see, a left child always contains a smaller value than the "parent node," and the right child contains a value larger than the parent node. Moreover, the *entire subtree* of a given node contains smaller values than the parent, if on the left side, or larger values than the parent, if on the right side. This organization permits efficient searching for any value in the tree.

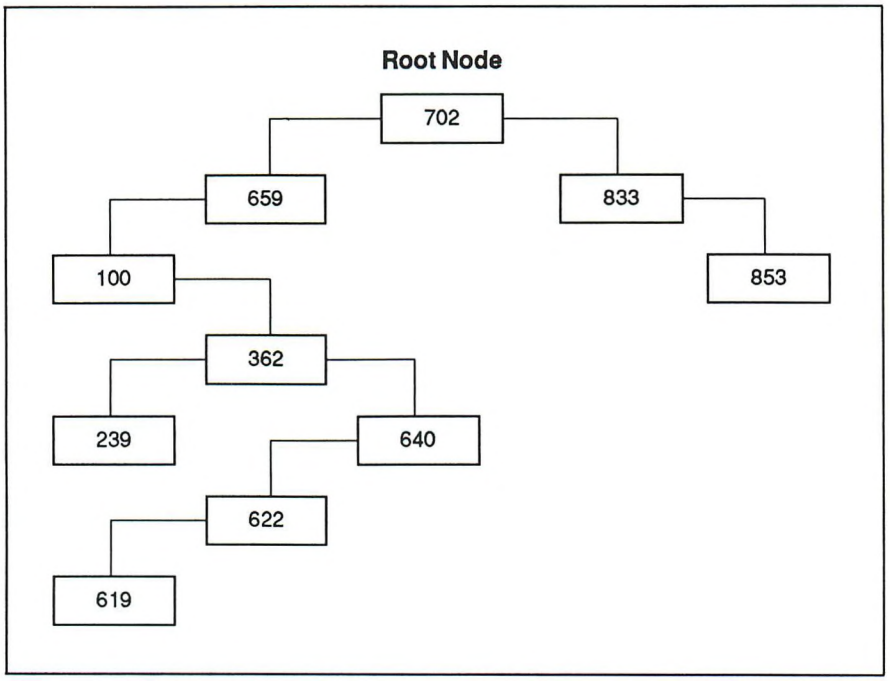

Figure 11.3 A Binary Tree

This sample binary tree contains only simple arithmetic nodes. However, you can create binary trees containing any kind of information (as long as each node is the same type). You can also organize or sort items in the tree using a variety of methods.

To build a binary tree, first declare the following types:

```
TYPE
    node_ptr = ^node;
    node = RECORD
        left,
        right : node_ptr;
        data  : Word;
        END;
```

Similar to the record type of a linked list, the `Node` type contains pointers that point to other records of the same type. If the node currently lacks a left or right child, the corresponding pointer field should be initialized to **NIL**.

To create a tree, first declare a pointer to the start of the tree and initialize this pointer to **NIL**, as shown below. Although no node currently exists in the tree, one will be inserted eventually. The first node will become the original ancestor of all other nodes in the tree.

```
VAR
    root_ptr : node_ptr;
BEGIN
    root_ptr := NIL;
```

You can start building the tree by inserting values, which can be generated in a number of ways—standard input, a data file, or with the **Random** function. Each time you get a value, add it to the tree by using the following five steps:

1. Examine `root_ptr`, the pointer to the first node of the tree. The expression `root_ptr^` is equivalent to the first node, if one exists. If no node exists, `root_ptr` is equal to **NIL**.
2. If the pointer is **NIL**, add a new record and assign the pointer to point to this record. Load the `data` field of the new record with the value to be added.
3. If the pointer is not **NIL**, then a node exists and must be compared to the new value. Compare the value to the `data` field of the node.
4. If the new value is less than the one at the node, use the `left` field as the new pointer value, and go to step 2.
5. If the new value is greater than the one at the node, use the `right` field as the new pointer value, and go to step 2.

You can use a recursive procedure to implement these steps. Linked lists and trees are often good subjects for recursive solutions. As explained in Chapter 3, "Procedures and Functions," a "recursive" procedure presents a simplified algorithm by calling itself repeatedly. In the case of binary trees, a recursive procedure traces left and right branches repeatedly until it finds a matching value at the end of the tree.

In the case of the steps discussed above, you initially call the procedure by passing the value you want to insert and `root_ptr` as arguments. The procedure calls itself to trace the left or right subtree, each time passing the `left` or `right` field as the pointer argument, as follows:

```
PROCEDURE insert( x : Integer; VAR ptr : node_ptr );
    BEGIN
        IF (ptr = NIL) THEN
            create_node( x, ptr )
        ELSE IF (x <ptr^.data) THEN
            insert( x, ptr^.left )
        ELSE
            insert( x, ptr^.right );
    END;
```

Notice how short the above procedure is. The actual work of inserting the node is the last step of the process, and it is carried out by a separate procedure written just for that purpose:

```
PROCEDURE create_node( x : Integer; VAR ptr : node_ptr );
    BEGIN
        New( ptr );
        ptr^.data  := x;
        ptr^.left  := NIL;
        ptr^.right := NIL;
    END;
```

The initialization of the `left` and `right` fields to NIL is critical. Otherwise, the `insert` procedure produces unpredictable results when it reaches the end of the tree and attempts a comparison. The procedures above effectively use NIL as the end-of-the-tree indicator.

CHAPTER 12

Advanced Topics

This chapter gives you a look inside QuickPascal. You can accomplish most any standard programming task by using the techniques presented in prior chapters. This chapter helps you deal with special situations, such as running out of dynamic memory, analyzing internal data formats, and linking to assembly language.

The bitwise operators are of special interest to assembly-language programmers, but are useful even if you don't use assembly language. You can use the bitwise operators to mask out bits within an integer, manipulate individual bits, or test a variable to see which bits are on.

After presenting the bitwise operators, the chapter shows a general picture of how QuickPascal organizes memory. Then the chapter illustrates internal data formats and explains how to link to assembly language.

12.1 The Bitwise Operators

You can access and manipulate bits by using the standard Boolean operators and the shift operators.

The logical operators **NOT**, **AND**, **OR**, and **XOR** work as bitwise operators when you use them with integer types. Bitwise operations take two data items of the same size and compare each bit in one operand to the corresponding bit in the other. For example, consider the following statement:

```
Result := $FF00 AND $9055;
```

QuickPascal implements this statement by comparing each bit in the constant `$FF00` to each corresponding bit in the constant `$9055`. The **AND** operator sets a bit in the result to 1, if and only if the corresponding bits in both operands have a value of 1:

```
          $FF00 = 1111 1111  0000 0000
AND       $90FF = 1001 0000  1111 1111

Result    $9000 = 1001 0000  0000 0000
```

The result of the operation is `$9000`. In the example above, using the **AND** operator with the constant `$FF00` in effect masks out the low 8 bits of a 16-bit integer. You can create other constants to selectively mask out any combination of bits. For example, you can use the **AND** operator to test whether a value is a multiple of four by masking out all but the lowest two bits and determining whether the result is zero:

```
IF (x AND $0003 = 0) THEN
    Writeln( 'x is multiple of 4.' );
```

Conversely, you can use the **OR** operator to set specific bits to 1. All of the bitwise operators work in a similar way. The following list shows how each operator works:

Operator	Sets a bit to 1 if:
NOT	The corresponding bit in operand is 0. (This operator takes just one operand.)
AND	Both corresponding bits in the operands have the value 1.
OR	Either one of the corresponding bits in the operand has the value 1.
XOR	Either one, but not both, of the corresponding bits has the value 1.

The **AND**, **OR**, and **XOR** operators all take two operands each. With each of these operators, the two integers you specify must be of the same type. For all the bitwise operators, the integer operands may be 8, 16, or 32 bits long. (Thus, operations with **LongInt** types are valid.)

The **SHL** and **SHR** operators take an integer operand and move the bits by the number of positions specified by the second operand. For example, the binary number for 12 is

```
$0C = 00001100
```

When you execute the statement `12 SHR 2`, each of the bits is moved two positions to the right, and the result looks like this:

```
$03 = 00000011
```

The result of the shift is the number 3. Note that shifting right by two is equivalent to dividing by 4. Left and right shifts are equivalent to multiplying and dividing by a power of two.

The general syntaxes for **SHL** and **SHR** are

IntVar **SHL** *NumPositions*
IntVar **SHR** *NumPositions*

The result is always of the same type as *IntVar*. The *NumPositions* argument determines the number of bit positions to shift. If this number is equal to or larger than the number of bits in *IntVar*, the result is always zero.

12.2 QuickPascal Memory Map

The memory map described in this section shows how a QuickPascal program uses memory. This information can help you develop strategies for very large programs that may run out of memory quickly.

When you execute a QuickPascal program, it uses all available memory (subject to the limits imposed by the {$M} compiler directive as described below). Figure 12.1 shows the general layout of memory in a QuickPascal program, and the rest of the section explains the meaning of items in this layout.

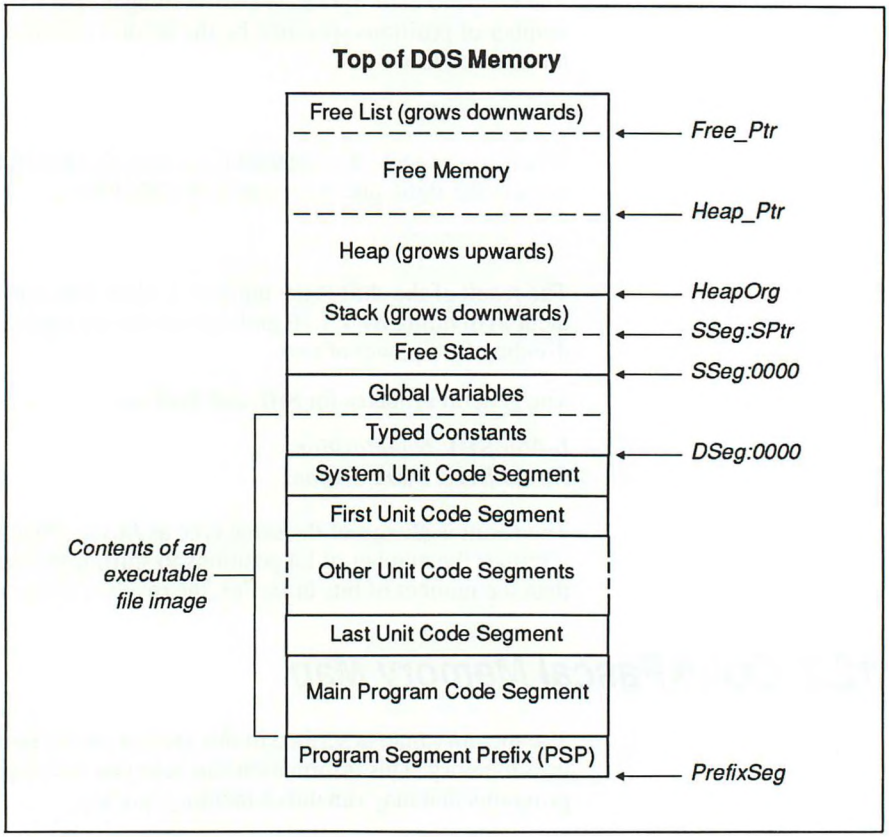

Figure 12.1 QuickPascal Memory Map

The solid lines in Figure 12.1 show demarcations between segments. A "segment" is an area of memory that can be up to 64K in length. Thus, as you can see from Figure 12.1, the code segment of the main program can never be more than 64K. However, for very large programs, you can surpass the 64K limit by simply adding additional units. The amount of code for each unit can be as large as 64K.

All global variables and typed constants are placed in a single segment. Thus, the total size of these data items cannot exceed 64K across all units.

The stack is placed in its own segment. You can set the stack size with the {$M} compiler directive. Increasing the stack lets the program accommodate more procedure calls, but may take away from memory available for the heap. (See Appendix B, "Compiler Directives.")

The heap consists of the rest of available RAM, unless you use the {$M} compiler directive to set maximum heap size. The heap contains all dynamic variables allocated through calls to **New** or **GetMem**. If you know in advance that your program *must* have a certain amount of heap space to run properly, you can use the {$M} compiler directive to specify a minimum heap size. DOS will not load the program unless it can allocate enough memory for the requested minimum heap size.

Certain variables appear in the QuickPascal memory map. These are public variables declared in the **System** unit, and your program can access them at any time (no special declaration is needed to use the **System** unit).

The following list describes the public variables appearing in the memory map:

Variable	Description
FreePtr	Pointer to beginning of "free list," (as described in Section 12.3). The free list is a series of records that describes the size and location of free memory blocks inside the heap. As the free list grows, **FreePtr** decreases because the free list grows downward.
HeapPtr	Pointer to top of heap. As the amount of space needed by dynamic variables increases, **HeapPtr** increases. It decreases only when memory is freed from the top of the heap.
HeapOrg	Pointer to beginning of heap. This variable has the generic type **POINTER**, as do the other pointers declared in the **System** unit. You cannot dereference pointers of this type, but you can assign the value of such a pointer to any other pointer, regardless of type.
PrefixSeg	Word variable containing segment address of Program Segment Prefix (PSP). When DOS loads your program, it constructs the PSP to store command-line arguments, interrupt vectors, and other information. See the *MS-DOS Programmer's Reference* or *MS-DOS Encyclopedia* for more information.

12.3 Managing the Heap

The simplest way to create dynamic variables is to make calls to **New** and **GetMem**. In small programs, you can generally call these procedures without worrying about how the heap is managed. However, if your programs make heavy use of dynamic memory, you can sometimes avoid running out of memory by knowing how the heap works.

The basic model of the heap is simple. Initially, the **New** and **GetMem** procedures simply increment **HeapPtr** by the size of the memory block you request and return a pointer to the beginning of the block.

The structure of the heap becomes more complicated when you start freeing blocks of memory. If you free memory at the current top of the heap, **HeapPtr** is decreased by the amount of memory freed. More often, however, the block to be freed is somewhere in the middle of the heap. In this case, the free-memory procedure (**Dispose** or **FreeMem**) adds a record to the free list to record the existence of the freed block.

The free block is then like a hole in the middle of the heap. The next time you request memory, the allocation procedure attempts to allocate memory from a free block if it can. Furthermore, if a block is freed adjacent to an existing free block, the two blocks form one larger free block, and the free list is updated to reflect this fact.

The maximum number of free blocks that Pascal programs can maintain is 8,192. Programs rarely reach this limit. However, you can use a variety of techniques to manage the heap, each of which is explained in a section below:

- Use the **Mark** and **Release** functions to free memory efficiently
- Determine the amount of heap space left and the number of records in the free list
- Set the **FreeMin** variable to prevent a deadlock between the heap and the free list
- Write a customized **HeapError** function to control how the program responds when you run out of heap space (the default action is to abort execution)

12.3.1 Using Mark and Release to Free Memory

The **Mark** and **Release** procedures provide a simpler and more efficient way to free memory than **Dispose** or **FreeMem**. However, the use of **Mark** and **Release** requires that you release memory in the reverse order to that in which you allocated it. Not all programs can adhere to this requirement.

The **Mark** procedure sets a pointer to point to the current top of the heap. The pointer can have any base type. Then if you allocate more dynamic memory, the new memory is higher in memory than the marked location (because the heap grows upward).

The **Release** procedure takes a single pointer as an argument, just as **Mark** does, and releases all dynamic memory above the pointer. In other words, it releases all memory allocated after the **Mark** procedure was called. The **Release** procedure basically works by setting **HeapPtr** to the value of the pointer argument.

For example, the following code allocates five dynamic variables:

```
New( PtrA );
New( PtrB );
Mark( Ptrs );
New( PtrC );
New( PtrD );
```

The next line of code releases the pointers declared after the `Mark( Ptrs )` statement. Specifically, it frees the memory pointed to by `PtrC` and `PtrD`:

```
Release( Ptrs );
```

Mark and **Release** impose significant limitations on program logic because they do not let you randomly free any block of memory; you can only free contiguous blocks at the top of the heap. However, if you use these procedures, you have none of the problems that sometimes occur with a free list.

Note that the **Mark** and **Release** procedures are not compatible with **Dispose** and **FreeMem**. Once you use one of the latter two procedures, you should not call **Release**.

12.3.2 Determining Free Memory and Size of the Free List

Two functions let you determine the amount of free memory available:

- The **MemAvail** function returns the total number of free bytes in the heap.
- The **MaxAvail** function returns the size of the largest single free block in the heap—in other words, the largest amount of memory that you can successfully request.

Generally, the **MaxAvail** function is the more useful of the two functions, since it lets you know whether any given call to **New** or **GetMem** will be successful. Both the **MemAvail** and **MaxAvail** functions return a **LongInt** result, and neither takes any parameters.

Determining the size of the free list also helps you discover if you are going to run out of memory. The free list is declared as follows:

```
TYPE
    FreeRec = RECORD
        OrgPtr, EndPtr : Pointer;
        END;

    FreeList = ARRAY [0..8190] OF FreeRec;
    FreeListP = ^FreeList;
```

Each record in the free list defines a single free memory block; the `OrgPtr` and `EndPtr` fields define the beginning and ending address, respectively, of the block. `EndPtr` points to the first byte after the end of the block. You can calculate the number of free blocks recorded in the free list by using the following statement:

```
free_blocks := (8192 - Ofs( FreePtr^ ) DIV 8) MOD 8192;
```

When the offset portion of the address in **FreePtr** is 0, the free list is empty. As a free block is added to the list, **FreePtr** decreases by eight bytes, and the free list also grows downward by eight.

12.3.3 Preventing Deadlock with the Free List

If the top of the heap is very close to the bottom of the free list, then deadlock can occur when you try to free blocks of memory. If the blocks to be freed are at the top of the heap, there is no problem. Otherwise, the free list is blocked from expanding.

This problem always causes a run-time error. You can neither allocate new dynamic memory nor free any existing memory. The only way to prevent this situation is to set a minimum amount of space between the locations pointed to by **HeapPtr** and **FreePtr**.

The **FreeMin** system variable sets the minimum free memory space (between the top of the heap and bottom of the free list). This variable has type **Word** and represents a distance in bytes. The **New** and **FreeMem** procedures will not allocate a block of memory if doing so would make the space between the heap and the free list fall below **FreeMin**. The **MemAvail** and **MaxAvail** functions also take **FreeMin** into account by subtracting it from the free memory area.

12.3.4 Writing a Heap-Error Function

By default, the **New** and **GetMem** procedures simply abort program execution when they cannot return a memory block of the requested size. However, you can create a customized function to respond to heap errors.

Your heap-error function is called by **New** and **GetMem** as needed, and it can return one of three values:

Value	Meaning
0	Failure; abort program
1	Failure; return the **NIL** pointer value to the caller of **New** or **GetMem**, but do not abort program
2	Memory successfully freed; authorize **New** or **GetMem** to retry

Your function should return the value 2 only if it was able to find a disposable dynamic variable in your program and free the necessary memory. If the function returns 2 without freeing memory first, then it will be called again.

To create a heap-error function, first declare a function similar to the one below:

```
{$F+}
FUNCTION heap_err( size : Word ) : Integer;
{$F-}
BEGIN
    { Cause FreeMem to return NIL but continue execution }
    heap_err := 1;
END;
```

You can give the function any name you want and supply any statements in the body of the function. However, the rest of the declaration should be the same. In particular, the **{$F+}** compiler directive is necessary to make sure the function is compiled for far calls. The single parameter of type **Word** gives the size of the memory block that **New** or **GetMem** failed to allocate.

After declaring your heap-error function, set the **HeapError** system variable to point to your function:

```
HeapError := @heap_err;
```

12.4 Internal Data Formats

Knowledge of internal data formats is useful if you want to write assembly-language procedures that access data from QuickPascal programs. Furthermore, you can use this knowledge to help determine what data types most efficiently store information for a given program.

Section 12.4.1 describes the data formats for all types except floating-point numbers. The floating-point data types are described separately in Section 12.4.2, because of their complexity.

12.4.1 Non-Floating-Point Data Types

The list that follows describes the internal formats of most data types, including structured variables such as arrays and records. Note that any given type is stored the same way, whether it is a simple variable or part of a record.

Where an integer value is designated as signed, QuickPascal uses the two's complement format for storing negative numbers. This format represents the negative of a number by logically negating each bit (as is done by the **NOT** operator) and then adding 1. For example, since 00000011 represents 3, then 11111100 is the logical negation, and 11111101 is the two's complement of 3.

Therefore, if you declare a **Byte** constant as –3, QuickPascal stores this value as 11111101. Generally speaking, you never have to carry out two's-complement conversion yourself. It is simply an internal format understood by both QuickPascal and the 8086 family of processors.

As a consequence, all nonnegative numbers have 0 in the most significant bit; all negative numbers have 1 in the most significant bit. The unsigned formats do not consider any number to be negative.

Data Type	Internal Format
Char	An unsigned byte. (Values range from 0 to 255.)
Byte	A signed byte. (Values range from –128 to 127.)
Integer	A signed word. (Values range from –32768 to 32767.)
Word	An unsigned word. (Values range from 0 to 65535.)
LongInt	A signed double word. (Values range from –2147483648 to 2147483647.)
Comp	A signed eight-byte integer. If the most-significant bit is 1 and the other bits are all 0, this type has the special value NAN (Not a Number). Note that with the {N+} directive, QuickPascal uses the 8087 coprocessor, if installed, to do calculations with **Comp** types.
Boolean	A byte assuming the value 0 (**False**) or 1 (**True**).

Enumerated types	Stored as an unsigned byte if the type can assume 256 values or fewer. Otherwise, the enumerated type is stored as an unsigned word. The first item in an enumerated-type definition corresponds to the value 0, and the rest are numbered sequentially in the order given.
STRING types	A sequence of bytes in which the first byte stores the current length of the string (which may be less than its maximum length), and the rest of the bytes store the string data. QuickPascal allocates enough space for a string's maximum length plus one additional byte (for the length indicator). The generic **STRING** type is equivalent to `STRING[255]`, which has the maximum limit.
CSTRING	Similar to **STRING** types, except that the first byte is not a length indicator, and the string data includes a terminating null byte. Maximum limit is 255 + null bytes. The generic **CSTRING** type defaults to this length.
SET types	Stored as a bit array, in which each bit corresponds to a specific element. If a bit is on, then the corresponding element is present. QuickPascal allocates one byte for each element, up to a maximum of 32 bytes (256 elements). Elements within a set are stored in order of their ordinal value, in which the first element is stored in the least-significant bit.
ARRAY types	Elements of the array are stored contiguously in memory, starting from the lowest-indexed component and going in sequence to the highest. In multidimensional arrays, the rightmost index changes fastest. Thus, in the array declared as `Grid[Rows,Cols]`, all elements for an entire row are stored next to each other in memory.
RECORD types	A sequence of variables stored contiguously in memory (the fields appear in memory in the same order they do in source code). With variant records, each variant case starts at the same address.
Pointer types	A double word that contains the offset address in the low word and the segment address in the high word. The special value **NIL** is a generic pointer containing the value zero.

12.4.2 Floating-Point Data Types

Figure 12.2 displays the formats used to represent floating-point numbers of different levels of precision. QuickPascal uses these data formats whether or not a math coprocessor is installed. If a coprocessor is not installed, then QuickPascal provides procedures to execute floating-point calculations. In cases of rounding, these procedures are not guaranteed to produce precisely the same results as a coprocessor.

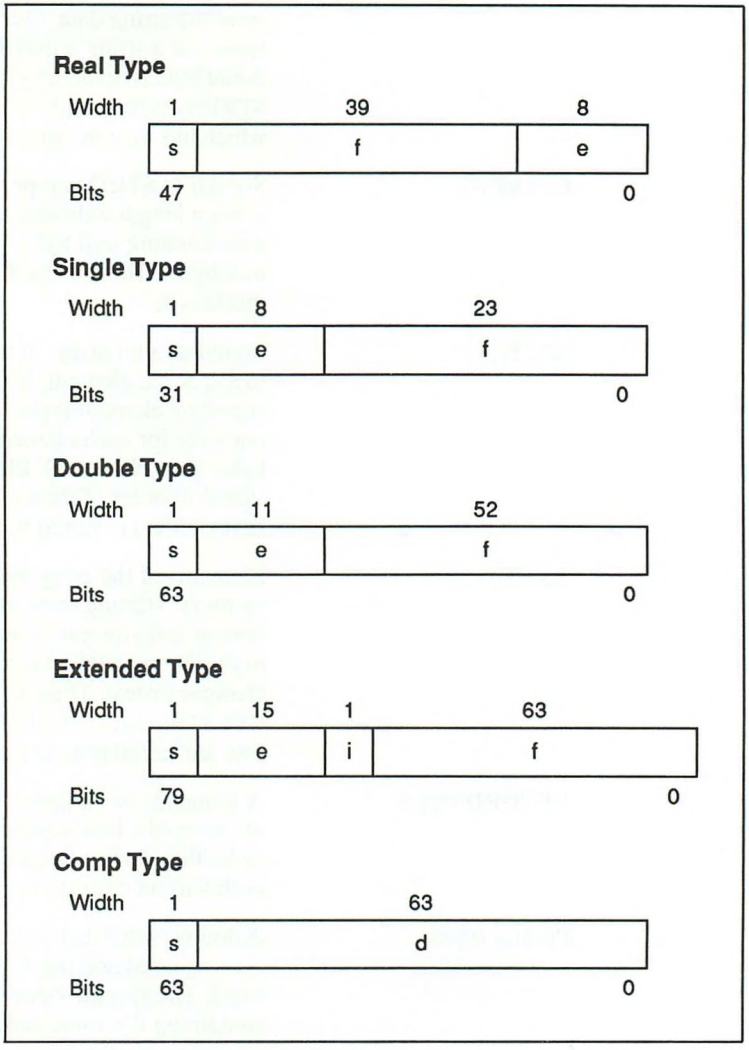

Figure 12.2 QuickPascal Data Formats

With each of the formats illustrated in Figure 12.2, the sign bit (s) indicates a negative number if on, and a nonnegative number if off. The fractional part (f) indicates the fractional part of the mantissa. To save space, the data formats all assume that the units' digit in the mantissa is equal to 1. Finally, the exponent is adjusted downward by a different number for each format. This means that for the four-byte **Single** type, the value represented is equal to

$$-1^{(s)} * 1.f * 2^{(e-127)}$$

For example, assume a variable of this type contains the following values:

```
s = 0
e = 127  decimal
f = 11110100000000000000000  binary
```

The value of the variable is 1.111101 binary. Note that (in contrast to Figure 12.2 and the example above) data storage on 8086 processors actually runs from least-significant bit (stored lowest in memory) to most-significant bit. This fact may make bytes appear to be backward if you use a debugging tool to dump a word at a time.

The exponents are automatically reduced by 129 for the six-byte **Real** format, by 1,023 for the **Double** format, and by 16,383 for the **Extended** format. This automatic reduction lets the exponent field represent a negative number (producing a floating-point value between 1 and –1).

In each format, if all bits are set to zero, then the resulting floating-point number is equal to zero.

12.5 Linking to Assembly Language

You can link QuickPascal programs to modules written with the Microsoft Macro Assembler and other assemblers. However, be forewarned that QuickPascal has a number of conventions and restrictions that differ from other Microsoft languages. When you write an assembly-language procedure to link to QuickPascal, your assembly-language module can include

- Procedures and functions that the main program can call
- References to global variables or data from a unit declared in an **INTERFACE** statement
- Static data that is private to your assembly-language module

However, the assembly-language module cannot declare public data for the QuickPascal code to reference. Furthermore, an assembly-language module must place all instructions in a single segment named CODE, and cannot use the **GROUP** directive. (Segment conventions are described in Section 12.5.2.)

Sections 12.5.1–12.5.6 list steps for writing an assembly-language module, in roughly the order that you would observe them in writing the code. You should read all of these sections carefully before attempting to write an assembly-language procedure (although the section on return values is optional and applies only when you write a function). Section 12.5.7 has a simple, but complete, example.

12.5.1 Setting Up a Link to Assembly Language

To link to assembly language, you must write an **EXTERNAL** declaration for each assembly-language procedure you're going to call and use the {$L} compiler directive to link in the object file.

To write an **EXTERNAL** declaration, simply write a procedure or function heading as you would normally and follow the heading with the keyword **EXTERNAL** as shown below:

```
FUNCTION compute( a, b, c: Integer ) : Integer; EXTERNAL;
PROCEDURE switcheroo( VAR a, b, c : Byte ); EXTERNAL;
```

Place the {$L} compiler directive at the beginning of your QuickPascal main program. The {$L} directive takes a single argument: the base file name of an object file. Note that you do not use LINK or any other utility to produce a program. Instead, QuickPascal sets up the link by copying the object file into the program and changing the file into its own internal object-code format. (The disk-based copy of the object file is not affected, however.)

12.5.2 Segment and Data Conventions

Observe the following conventions when declaring segments and data:

- Place all executable statements in a segment named CODE or CSEG. Place all data declarations in a segment named DATA or DSEG. All other segments are ignored, as are group statements.
- The segments can have the **BYTE** or **WORD** align type, but QuickPascal will always give segments word alignment.
- Do not specify class names.
- Declare **PUBLIC** each procedure that you want to call from QuickPascal. However, QuickPascal ignores any **PUBLIC** data declarations.

- Do not initialize variables in the **DATA** segment. (Variables must be initialized with instructions.)
- You can access data declared by the main program (or declared in an **INTERFACE** statement) by declaring it with the **EXTRN** directive. However, you cannot use the **HIGH** or **LOW** operators with external data. See Section 12.4, "Internal Data Formats," for information on how to evaluate each data type.

12.5.3 Entering the Procedure

The entry sequence for QuickPascal procedures is the same as that for other Microsoft languages. The first two instructions set up the BP register as the "framepointer," which points to the stack area of the current procedure. All parameters and local variables are available through BP:

```
push    bp
mov     bp, sp
sub     sp, local_size
```

In the code above, `local_size` is a placeholder for the total size of local parameters you wish to use. This last step is entirely optional. Your procedure may be able to use registers to hold all temporary values. If not, you can use locations on the stack to hold data. (Specifically, these locations are below the address pointed to by BP, but no more than `local_size` bytes below this address.)

Procedures called by QuickPascal must preserve the values of the BP, SP, SS, and DS registers. BP and SP are preserved by the standard entry and exit code. If you need to alter DS or SS, you must push their values onto the stack and pop them just before returning as shown below:

```
push    bp
mov     bp, sp
sub     sp, local_size
push    ds
```

12.5.4 Accessing Parameters

All parameters and local variables are accessed as indirect memory operands using the BP register. To determine the location of each parameter, you need to understand the QuickPascal calling conventions. This section summarizes those conventions and gives some examples with specific procedures.

QuickPascal pushes each parameter onto the stack—in the order that the parameters appear in the source code—before making the actual call. Consequently, the first parameter is highest in memory. (Recall that the stack grows downward.) You can pass parameters by value or by reference. In Pascal, **VAR** parameters are passed by reference; other parameters are passed by value.

When a parameter is passed by reference, QuickPascal pushes a four-byte pointer to the data. The offset portion of the pointer is always pushed first and is therefore higher in memory.

When a parameter is passed by value, Pascal takes different actions depending on the data type:

- If the value parameter is **Char**, **Boolean**, any Pointer, any integer, or any floating-point type, QuickPascal pushes the parameter onto the stack.
- If the parameter is a string or set type, QuickPascal passes a pointer to the data. This action is really the same as passing by reference. If you want to avoid any possibility of altering the data, make a temporary copy of the data, then work with the temporary copy. (QuickPascal procedures use this technique.)
- If the parameter is an array or record type, QuickPascal pushes the variable directly onto the stack if it is no more than four bytes long. Otherwise, it pushes a pointer to the data.

This calling convention for value parameters is not shared by other Microsoft high-level languages, which always push a value parameter directly onto the stack.

Note that the **PUSH** instruction can push only one word of data at a time. QuickPascal always pushes the most-significant portion first, consistent with the 8086-processor convention that the most-significant portion is stored highest in memory.

As noted in the next section, the QuickPascal code sometimes places an extra parameter on the stack for functions. (This extra parameter is a pointer to the location of the return value.)

Finally, the QuickPascal code calls the procedure with a **CALL** instruction. Calls to external procedures are always far, so you should declare your assembly procedure with the **FAR** keyword.

As an example, consider the following procedure call:

```
PROCEDURE quad( VAR x : Real; a, b, c : Integer ); EXTERNAL;
.
.
.
quad( result, 2, 3, 4 );
```

QuickPascal implements the call by placing the following items on the stack, and in this order:

1. The segment address of result
2. The offset address of result
3. The value 2 (as a word)
4. The value 3
5. The value 4
6. The segment of the return address (the address to return to when done)
7. The offset of the return address

A far CALL instruction places the last item, the return address, on the stack. After you perform the entry sequence, the BP register points to the location just below the return address. Each parameter can then be accessed with the following equates:

```
result  EQU     <[BP + 12]>
a       EQU     <[BP + 10]>
b       EQU     <[BP +  8]>
c       EQU     <[BP +  6]>
```

As another example, consider the procedure call

```
PROCEDURE repeat_it( num : LongInt;
                     stuff : STRING ); EXTERNAL;
.
.
.
repeat_it( n, 'This is a message.' );
```

With this procedure, QuickPascal implements the call by placing the following items on the stack:

1. The most-significant word of the long integer `n`
2. The least-significant word of `n`
3. The segment address of the string data
4. The offset address of the string data
5. The segment of the return address
6. The offset of the return address

A far CALL instruction places the last two items on the stack. After writing the entry sequence, you can access the parameters with the following equates:

```
num     EQU     <[BP + 10]>
stuff   EQU     <[BP +  6]>
```

12.5.5 Returning a Value

This section affects functions only. From the standpoint of assembly code, functions are just procedures that have a return value. To return a value to QuickPascal code, follow these conventions:

- For a six-byte **Real** type, place the result in DX:BX:AX, in which DX holds the most-significant word and AX the least.
- For a string, **CSTRING**, **Comp**, or floating-point type other than **Real**, QuickPascal passes an additional parameter. This parameter is pushed first and is a pointer to a temporary storage location. Place the result of the function in this location.
- For ordinal types (including **Char**, **Boolean**, and any integer), place the result in AL if one byte, AX if two bytes, and DX:AX if a double word (in which DX holds the most-significant byte).
- QuickPascal does not support functions that return array or record types. (However, you can set the value of an array or record if QuickPascal passes it as a **VAR** parameter.)

12.5.6 Exiting the Procedure

To exit the procedure, first pop any registers you preserved on the stack. Then write the following instructions:

```
mov   sp,bp
pop   bp
ret   param_size
```

In the code given above, `param_size` is a placeholder for the total size of parameters in your procedure. This last instruction is necessary to completely restore the stack.

12.5.7 A Complete Example

The following example of linking QuickPascal programs to modules written in assembly language is simple, but complete. The Pascal source code contains the following statements:

```
{$L SWITCH }
PROCEDURE switch( VAR a, b : Word ); EXTERNAL;
.
.
.
switch( x, y );
```

Since the parameters are **VAR** parameters, QuickPascal passes pointers to the parameters. Then, the assembly code must use indirect register operands to access the data. The following procedure exchanges the values of the two variables.

```
; SWITCH.ASM
;
CODE    SEGMENT WORD PUBLIC

        ASSUME  CS:CODE

        PUBLIC  switch          ; Make procedure public

a       EQU     <[BP + 10]>      ; Parameters for switch
b       EQU     <[BP + 6]>       ;  procedure

switch  PROC    FAR
        push    bp              ; Entry sequence
        mov     bp,sp
        push    ds

        lds     si, a           ; Load ptr into ds:si
        les     di, b           ; Load ptr into es:di
        mov     bx, es:[di]     ; temp = b
        xchg    bx, [si]        ; Switch temp and a
        xchg    bx, es:[di]     ; Switch temp and b

        pop     ds
        mov     sp,bp           ; Exit sequence
        pop     bp
        ret     8               ; Total of 8 bytes in
                                ;  parameters
switch  ENDP
```

PART 3

Graphics and Objects

PART 3

Graphics and Objects

Part 3 of *Pascal by Example* covers QuickPascal support of graphics, fonts, and object-oriented programming. Each of these topics requires a moderate degree of experience with Pascal. You need to be comfortable with all of the topics in Part 1 and may find familiarity with Part 2 helpful.

The graphics and fonts chapters show you how to use the QuickPascal graphics unit to create colorful and informative screen displays. The chapter on object-oriented programming introduces the fundamental concepts of programming with objects and how to implement those concepts in QuickPascal.

CHAPTERS

CHAPTER 13

Using Graphics

This chapter explains how to call graphics routines that set points, draw lines, change colors, and draw shapes such as rectangles and circles. The first section describes the general structure of any graphics program, defines important graphics terms, and works through an example program step by step, showing how to use the basic routines. Subsequent sections explain video modes, coordinate systems, and animation.

To run any graphics programs, your computer must have graphics capability. The QuickPascal graphics facility supports the Color Graphics Adapter (CGA), the Enhanced Graphics Adapter (EGA), and the Video Graphics Adapter (VGA) video modes available on IBM® and IBM-compatible computers. The graphics facility also supports the Olivetti® (and AT&T®) enhanced video modes and the Hercules® monochrome graphics mode.

This chapter discusses the techniques for writing graphics programs but does not cover every graphics procedure and function (there are more than 75). You can explore additional topics by using the QP Advisor and the example programs. Be sure to check the README.DOC file for any last-minute changes or additions.

13.1 Getting Started with Graphics

The subject of graphics and color display on computers is fairly complex. This chapter and the next one, "Using Fonts," provide an introduction to graphics and fonts. The next sections cover common graphics terms and list sources for more information on graphics.

13.1.1 Graphics Terms

There are several concepts you need to know before you can create graphics programs. The following list explains the most useful terms:

- The "*x* axis" determines the horizontal position on the screen. The "origin" (point 0, 0) is in the upper left corner. The maximum number of horizontal "pixels" (picture elements) varies from 320 to 640 to 720, depending on the graphics card installed and the graphics mode in effect.
- The "*y* axis" is the vertical position on the screen. The origin is the upper left corner. The number of vertical pixels ranges from 200 to 480.
- Each graphics mode offers a "palette" from which you may choose the colors to be displayed. You may have access to 2, 4, 8, 16, or 256 "color indexes," depending on the graphics card in the computer and the graphics mode in effect. The color is the index to the palette of colors displayable by a particular graphics adapter.
- The CGA modes offer four fixed palettes containing predefined colors that may not be changed. In EGA, MCGA, and VGA graphics modes, you may change any of the color indexes by providing a color value that describes the mix of colors you wish to use.
- A color index is always a short integer. A color value is always a long integer. When you're calling graphics functions that require color-related parameters, you should be aware of the difference between color indexes and color values.

13.1.2 For More Information

The following books cover a variety of graphics topics that you may find useful. They are listed only for your convenience. With the exception of its own publications, Microsoft does not endorse these books or recommend them over others on the same subject.

- Artwick, Bruce. *Microcomputer Displays, Graphics and Animation*. Englewood Cliffs, NJ: Prentice-Hall, Inc., 1985.

  Discussion of microcomputer graphics and animation from the creator of *Microsoft Flight Simulator*.

- Kliewer, Bradley D. *EGA/VGA: A Programmer's Reference Guide*. New York, NY: Intertext Publications, Inc., 1988.

  A detailed discussion of the EGA and VGA video modes.

- Norton, Peter, and Richard Wilton. *The New Peter Norton Programmer's Guide to the IBM PC and PS/2*. Redmond, WA: Microsoft Press, 1985.

  The standard guide to the inside of the IBM PC and PS/2 families of computers. Several chapters are devoted to video modes.

- Wilton, Richard. *Programmer's Guide to PC and PS/2 Video Systems*. Redmond, WA: Microsoft Press, 1987.

  An advanced guide to all the PC and PS/2 video modes.

13.2 Writing Your First Graphics Program

In QuickPascal, all graphics programs must follow these six basic steps:

1. Use the **MSGraph** unit.
2. Set a video mode.
3. Determine the video parameters.
4. Set up a coordinate system.
5. Create and display graphics.
6. Restore initial video configuration and exit the program.

A simple graphics program, 1STGRAPH.PAS, is shown below. In the next few pages, this program will be dissected into the six required elements of a graphics program.

```
PROGRAM FirstGraphics;
{ 1STGRAPH.PAS: Demonstrates basic graphics program structure }

USES
    MSGraph;

VAR
    a, i : integer;
    vc   : _VideoConfig;

BEGIN  { Begin main program. }

    _ClearScreen( _GClearScreen);

    { Set the highest resolution video mode and check for success }
    a := _SetVideoMode( _MaxResMode );
```

```
IF ( a = 0 ) THEN
        BEGIN
            Writeln( 'No valid graphics mode; hit RETURN to continue' );
            Readln;
            a := _SetVideoMode( _DefaultMode );
            Halt ( 0 );
        END;

    { Find out some screen characteristics. }
    _GetVideoConfig( vc );
    Writeln( 'mode: ',vc.Mode );
    Writeln( 'Horizontal resolution: ', vc.NumXPixels );
    Writeln( 'Vertical resolution: ', vc.NumYPixels );
    Writeln( 'Number of colors: ', vc.NumColors );

    { Draw a colored rectangle and ellipse in lower left quadrant. }
    _SetColor( 4 );
    _Rectangle( _GBorder, 0, vc.NumYPixels - 1,
                vc.NumXPixels DIV 4,
                vc.NumYPixels * 3 DIV 4 );
    _Ellipse( _GBorder, 0, vc.NumYPixels - 1,
              vc.NumXPixels DIV 4,
              vc.NumYPixels * 3 DIV 4 );

    { Draw a line from the corner of the rectangle to the screen center. }
    _SetColor( 3 );
    _MoveTo( vc.NumXPixels DIV 4, vc.NumYPixels * DIV 4 );
    _LineTo( vc.NumXPixels DIV 2, vc.NumYPixels DIV 2 );

    { Wait for RETURN key and then restore video mode. }
    Readln;
    _SetVideoMode(_DefaultMode);
END.
```

Using the MSGraph Unit

The first step in creating 1STGRAPH.PAS is to use the **MSGraph** unit within your QuickPascal program. The **MSGraph** unit contains the constants, data types, procedures, and functions used in graphics programs. QuickPascal units are explained in Chapter 7, "Units." The **MSGraph** unit is the library of graphics features you need to access from your program.

The **USES** section of your program calls **MSGraph**. This section appears immediately after the **PROGRAM** declaration line and looks like this:

```
USES
    MSGraph;
```

Note that some components of the **MSGraph** unit are incompatible with some functions within the **Crt** unit. Some of the **Crt** unit routines that deal with the display (such as **HighVideo** or **DirectVideo**) conflict with the graphics functions.

The following **Crt** routines are safe to use with the **MSGraph** unit:

AssignCRT	**NoSound**	**WhereY**
Delay	**ReadKey**	**Window**
GotoXY	**Sound**	
KeyPressed	**WhereX**	

Setting the Video Mode

Before you can start drawing pictures on the screen, your program must tell the graphics adapter to switch from text mode to graphics mode. Call **_SetVideoMode**, passing it a predefined constant that tells it which mode to display. The constants listed in Table 13.1 are defined in the **MSGraph** unit. The dimensions are listed in columns for video text modes and in pixels for graphics mode.

Table 13.1 Constants Set by _SetVideoMode

Constant	Video Mode	Mode Type/Hardware
_DefaultMode	Restores to original mode	Both/All
_MaxResMode	Highest resolution	Graphics/All
_MaxColorMode	Maximum colors	Graphics/All
_TextBW40	40 column text, 16 gray	Text/CGA
_TextC40	40 column text, 16/8 color	Text/CGA
_TextBW80	80 column text, 16 gray	Text/CGA
_TextC80	80 column text, 16/8 color	Text/CGA
_MRes4Color	320 × 200, 4 color	Graphics/CGA
_MResNoColor	320 × 200, 4 gray	Graphics/CGA
_HResBW	640 × 200, BW	Graphics/CGA
_TextMono	80 column text, BW	Text/MDPA
_HercMono	720 × 348, BW for HGC	Graphics/HGC
_MRes16Color	320 × 200, 16 color	Graphics/EGA
_HRes16Color	640 × 200, 16 color	Graphics/EGA
_EResNoColor	640 × 350, BW	Graphics/EGA
_EResColor	640 × 350, 4 or 16 color	Graphics/EGA
_VRes2Color	640 × 480, BW	Graphics/VGA/ MCGA
_VRes16Color	640 × 480, 16 color	Graphics/VGA
_MRes256Color	320 × 200, 256 color	Graphics/VGA/ MCGA
_OResColor	640 × 400, 1 of 16 colors	Graphics/Olivetti

These video modes are described more fully in Section 13.3.

The special modes **_MaxResMode** and **_MaxColorMode** are used when you simply want QuickPascal to determine the highest resolution or maximum color mode available.

If the **_SetVideoMode** function returns a 0, it means the hardware does not support the selected mode. You may continue to select alternate video modes until a nonzero value is returned. If the hardware configuration doesn't support any of the selected video modes, then exit the program.

The segment of example program 1STGRAPH.PAS below sets the video mode for maximum resolution and then checks for success. If the function **_SetVideoMode** fails (and returns a 0), the program prints a message, restores the video mode, and halts.

```
{ Set the highest resolution mode and check for success }
a := _SetVideoMode( _MaxResMode );
IF ( a = 0 ) THEN
    BEGIN
    Writeln( 'No valid graphics mode; hit RETURN to continue' );
    Readln;
     a := _SetVideoMode( _DefaultMode );
    Halt (0);
    END;
```

Determining the Video Parameters

After entering graphics mode, you can check the current video configuration. This is necessary when you have used the constant **_MaxResMode** or **_MaxColorMode**, since you do not know the specifics of the video mode selected.

The video configuration requires a special structure called **_VideoConfig**, which is defined in the **MSGraph** unit. The **_GetVideoConfig** procedure finds the current video configuration information.

The **_VideoConfig** structure contains the following elements:

```
{ Structure for GetVideoConfig }

  _VideoConfig = RECORD
    NumXPixels : Integer; { Horizontal resolution }
    NumYPixels : Integer; { Vertical resolution }
    NumTextCols : Integer; { Number of text columns available }
    NumTextRows : Integer; { Number of text rows available }
    NumColors : Integer; { Number of actual colors }
    BitsPerPixel : Integer; { Number of bits per pixel }
    NumVideoPages : Integer; { Number of available video pages }
    Mode : Integer; { Current video mode }
    Adapter : Integer; { Active display adapter }
    Monitor : Integer; { Active display monitor }
    Memory : Integer; { Adapter video memory in K bytes }
  END;
```

The variables within the **_VideoConfig** structure are initialized when you call **_GetVideoConfig**. In the following example program, the video mode is set to **_MaxResMode**. The program then calls **_GetVideoConfig** to determine the screen dimensions (in pixels) and the maximum number of colors allowed.

```
{ Find out some screen characteristics. }
_GetVideoConfig( ,vc );
Writeln( 'mode: 'vc.Mode );
Writeln( 'Horizontal resolution: ', vc.NumXPixels );
Writeln( 'Vertical resolution: ', vc.NumYPixels );
Writeln( 'Number of colors: ', vc.NumColors );
```

Setting Up Coordinate Systems

The third step in writing a QuickPascal graphics program is to establish the "coordinate system."

A coordinate system is used to identify a pixel location relative to a horizontal and vertical axis. In a graphics mode, each pixel on the screen can be located by means of a unique pair of coordinates. The QuickPascal graphics unit supports the following coordinate systems:

- Text coordinates
- Physical coordinates
- Viewport coordinates
- Window coordinates

Since this example graphics program does not specify otherwise, the coordinate system used is that provided by the physical screen dimensions. This is known as the "physical-coordinate system."

But how much screen is available to work with? There might be 720 × 348, 640 × 480, 640 × 400, 640 × 350, or 640 × 200 pixels. The **_VideoConfig** structure elements **NumXPixels** and **NumYPixels** provide the screen dimensions. On a VGA card, a request for maximum resolution would set the screen to the **_VRes16Color** mode. The horizontal resolution is 640 pixels and vertical resolution is 480. The horizontal resolution might be 640, but the pixels are numbered 0 – 639.

The physical-coordinate system places the origin (or the coordinate pair (0,0)) at the upper left corner of the screen. Increasing positive values of the *x* coordinate extend from left to right. Increasing positive values of the *y* coordinate extend from top to bottom.

The physical-coordinate system is dependent on the hardware and display configuration and cannot be changed. The other coordinate systems and the procedures used to translate between them are described in Section 13.4.

Drawing Graphics

The next few lines in this example program draw a rectangle and an ellipse and then a line to the center of the screen. Prior to drawing the rectangle or the line, the graphics color is set to two different values. The program segment that draws the rectangle and ellipse is shown below:

```
{ Draw a colored rectangle and ellipse in lower left quadrant. }
_SetColor( 4 );
_Rectangle( _GBorder, 0, vc.NumYPixels - 1,
            vc.NumXPixels DIV 4, vc.NumYPixels * 3 DIV 4 );
_Ellipse( _GBorder, 0, vc.NumYPixels - 1,
          vc.NumXPixels DIV 4, vc.NumYPixels * 3 DIV 4 );
```

The call to the **_Rectangle** procedure has five arguments. The first argument is the "fill flag," which may be either **_GBorder** or **_GFillInterior**. The fill flag determines whether the figure will be drawn with just the border or as a solid figure. Choose **_GBorder** if you want a rectangle of four lines (a border only, in the current line style). Or you can choose **_GFillInterior** if you want a solid rectangle (filled in with the current color and fill pattern).

The second and third arguments are the *x* and *y* coordinates of one corner of the rectangle. The fourth and fifth arguments are the coordinates of the opposite corner. The coordinates for the two corners are defined in terms of the measurements obtained from **_GetVideoConfig**. As a result, this program draws the rectangle correctly in any graphics mode.

The **_Ellipse** procedure draws an ellipse on the screen. Its parameters resemble the parameters for **_Rectangle**. Both procedures require a fill flag and two corners of a "bounding rectangle." When the ellipse is drawn, four points touch the edges of the bounding rectangle. A bounding rectangle defines the space in which a rounded figure is drawn. For a rounded figure, the center of the bounding rectangle defines the center of the rounded figure. A bounding rectangle is defined by the coordinates of the upper left corner and the lower right corner of the rectangle.

When the program first enters a graphics mode or sets a viewport (as discussed in Section 13.4), the drawing position is set to the center of the screen. Since the **_Rectangle** procedure takes the corners of the rectangle as parameters, there is no need to set the drawing position. But after the figures are drawn, the program moves the current drawing position (with the **_MoveTo** procedure) and draws a line from the new position to the center of the screen:

```
{ Draw a line from the corner of the rectangle to the screen center. }
_SetColor( 3 );
_MoveTo( vc.NumXPixels DIV 4, vc.NumYPixels * 3 DIV 4 );
_LineTo( vc.NumXPixels DIV 2, vc.NumYPixels DIV 2 );
```

Again, the coordinates are given in terms of **NumXPixels** and **NumYPixels** to ensure portability across video modes.

Restoring Initial Video Configuration and Exiting

The final step in any graphics program is to restore the default video mode. In the sample program, the **Readln** procedure waits for the RETURN key. The **_SetVideoMode** function restores the screen to the default mode, which sets the screen back to normal.

```
{ Wait for RETURN key and then restore the video mode. }
Readln;
a := _SetVideoMode(_DefaultMode);
```

13.3 Using Video Modes

The QuickPascal **MSGraph** unit provides support for the following video adapters and displays:

- Monochrome Display Printer Adapter (MDPA)
- Hercules-compatible graphics adapters
- CGA
- EGA
- VGA
- MCGA
- Olivetti-compatible graphics adapters (including AT&T 6300 series)

The sections that follow explain how to select a video mode, then discuss the major CGA, EGA, and VGA graphics modes. A complete listing of the QuickPascal video modes that can be set by the **_SetVideoMode** function is shown in Table 13.1.

NOTE *If you use a Hercules graphics card, you must run the MSHERC.COM program before attempting to display any graphics in* ***_HercMono*** *mode. If your computer also has a color graphics card, you must run MSHERC.COM with the /H (/HALF) option to set the Hercules graphics card in HALF mode. Otherwise, the results will be unpredictable.*

The **_VideoConfig** structure, which is returned by **_GetVideoConfig**, includes fields that denote the type of graphics adapter and monitor in use. Table 13.2 lists the constants for graphics adapters.

Table 13.2 Constants for Graphics Adapters

Constant	Value	Description
_MDPA	0x0001	Monochrome Display Adapter
_CGA	0x0002	Color Graphics Adapter
_EGA	0x0004	Enhanced Graphics Adapter
_VGA	0x0008	Video Graphics Array
_MCGA	0x0010	MultiColor Graphics Array
_HGC	0x0020	Hercules Graphics Card
_OCGA	0x0042	Olivetti Color Graphics Adapter
_OEGA	0x0044	Olivetti Enhanced Graphics Adapter
_OVGA	0x0048	Olivetti Video Graphics Array

Table 13.3 lists the constants for monitors.

Table 13.3 Constants for Monitors

Constant	Value	Description
_Mono	0x0001	Monochrome
_Color	0x0002	Color
_ENHColor	0x0004	Enhanced Color
_AnalogMono	0x0008	Analog Monochrome only
_AnalogColor	0x0010	Analog Color only
_Analog	0x0018	Analog Monochrome and Color modes

13.3.1 Selecting a Video Mode

Before you can display graphics, you must put the graphics adapter into a graphics mode. As shown in the example program, the **_SetVideoMode** function performs this task. Before calling **_SetVideoMode**, you must decide which graphics modes are acceptable for your purposes. There are several ways you can select an appropriate graphics mode:

- Set the mode to a specified value. If you want the program to work in a high-resolution EGA color mode, use **_SetVideoMode** to set the mode to **_EResColor**:

```
a := _SetVideoMode( _EResColor );
```

- Request either the highest resolution available using the **_MaxResMode** constant or the greatest color selection, **_MaxColorMode**, available with your monitor and adapter configuration:

```
{ Selects highest resolution }
a := _SetVideoMode( _MaxResMode );
{ or, selects most colors }
a := _SetVideoMode( _MaxColorMode );
```

- Specify a specific resolution mode (such as 320 × 200 pixels) by first determining the video adapter type (from the **_VideoConfig** structure) and then setting the appropriate mode for the adapter. The program fragment below shows how this technique can be used to set a 320 × 200 pixel resolution mode.

```
_GetVideoConfig( vc );     { Find out adapter type }
    IF (vc.Monitor = _Mono) THEN
        BEGIN
        Writeln( 'This program requires a color monitor.' );
        Halt( 0 );
        END
    ELSE
        CASE vc.Adapter OF
            _MDPA, _HGC :
                  BEGIN
                  Writeln( 'This program requires a color graphics adapter.' );
                  Halt( 0 );
                  END;
             _CGA, _OCGA : a := _SetVideoMode( _MRes4Color );
             _EGA, _OEGA : a := _SetVideoMode( _MRes16Color );
             _MCGA, _VGA : a := _SetVideoMode( _MRes256Color );
            END;
    { To get video configuration after mode is set }
    _GetVideoConfig( vc );
```

This program fragment begins by calling **_GetVideoConfig**, which determines the graphics configuration and tells the type of adapter currently in use. If the monitor is monochrome, the program displays a message and halts. Next, the **CASE** statement enters the appropriate graphics mode. Finally, the program segment calls **_GetVideoConfig** to get the new configuration values after the mode has been set.

To view every possible graphics mode, you can run the example program GRAPHIC.PAS, shown below. Sections 13.3.2–13.3.6 explain the various graphics modes.

```
PROGRAM graphic;  { Displays every graphic mode }

USES
    Crt, MSGraph;
CONST
    modes : ARRAY [0..11] OF Integer =
           (_MRes4Color, _MResNoColor, _HResBW, _HercMono,
            _MRes16Color, _HRes16Color, _EResNoColor, _EResColor,
            _VRes2Color, _VRes16Color, _MRes256Color, _OResColor );
VAR
    vc : _VideoConfig;
    ch, key   : Char;
    which : Char;
    a : Integer;

PROCEDURE print_menu;
{ Prints a menu on the screen }
BEGIN

    Writeln( 'Please choose a graphics mode' );
    Writeln( 'Type "x" to exit' );
    Writeln;
    Writeln( '0  _MRES4COLOR' );
    Writeln( '1  _MRESNOCOLOR' );
    Writeln( '2  _HRESBW' );
    Writeln( '3  _HERCMONO' );
    Writeln( '4  _MRES16COLOR' );
    Writeln( '5  _HRES16COLOR' );
    Writeln( '6  _ERESNOCOLOR' );
    Writeln( '7  _ERESCOLOR' );
    Writeln( '8  _VRES2COLOR' );
    Writeln( '9  _VRES16COLOR' );
    Writeln( 'a  _MRES256COLOR' );
    Writeln( 'b  _ORESCOLOR' );
END;


PROCEDURE show_mode( which : Char );
{ Shows the different video modes. }

VAR
    nc, i  : Integer;
    height : Integer;
    width  : Integer;
    mode   : STRING;
    r : Real;
    e,m : Integer;
```

```
BEGIN
    mode := which;
    IF (mode < '0') OR (mode > '9') THEN
       IF mode = 'a' THEN
           mode := '10'
       ELSE IF mode = 'b' THEN
               mode := '11'
           ELSE Halt; { Exit procedure }
    Val( mode, r, e );
    m := Trunc( r );
    a :=  _SetVideoMode( modes[m] );
    IF (a <> 0) THEN
        BEGIN
        _GetVideoConfig( vc );
        nc := vc.NumColors;
        width := vc.NumXPixels DIV nc;
        height := vc.NumYPixels DIV 2;
        FOR i := 1 TO ( nc - 1 ) DO
            BEGIN
            _SetColor( i );
            _Rectangle( _GFillInterior, i * width,
                        0, ( i + 1 ) * width, height );
            END;
        END { IF a not equal to 0 }
    ELSE
        BEGIN
        Writeln( 'Video mode ', which, ' is not available.' );
        Writeln( 'Please press ENTER.' );
        END;

    Readln;  { Wait for ENTER to be pressed }
    a := _SetVideoMode( _DefaultMode );
    print_menu;
END;


BEGIN  { Begin main program }

    key := ' ';
    _ClearScreen( _GClearScreen );
    print_menu;
    WHILE ( key <> 'x' ) DO
        BEGIN
        key := ReadKey;
        show_mode( key );
        END;
END.
```

13.3.2 CGA Color Graphics Modes

The CGA color graphics modes **_MRes4Color** and **_MResNoColor** display four colors selected from one of several predefined palettes of colors. They display these foreground colors against a background color that can be any one of the 16 available colors. With the CGA hardware, the palette of foreground colors is predefined and cannot be changed. Each palette number is an integer as shown in Table 13.4.

Table 13.4 Available CGA Colors

Palette Number	Color Index 1	2	3
0	Green	Red	Brown
1	Cyan	Magenta	Light Gray
2	Light Green	Light Red	Yellow
3	Light Cyan	Light Magenta	White

The **_MResNoColor** graphics mode produces palettes containing various shades of gray on black-and-white monitors. The **_MResNoColor** mode displays colors when used with a color display. However, only two palettes are available with a color display. You can use the **_SelectPalette** function to select one of these predefined palettes. Table 13.5 shows the correspondence between the color indexes and the palettes.

Table 13.5 CGA Colors: _MResNoColor Mode

Palette Number	Color Index 1	2	3
0	Blue	Red	Light Gray
1	Light Blue	Light Red	White

You may use the **_SelectPalette** function in conjunction with the **_MRes4Color**, **_MResNoColor**, and **_OResColor** graphics modes. To change palettes in other graphics modes, use the **_RemapPalette** or **_RemapAllPalette** routines.

In **_OResColor** mode, you can choose one of 16 foreground colors by passing a value in the range 0 – 15 to the **_SelectPalette** function. The background color is always black.

The following program sets the video mode to **_MRes4Color** and then cycles through background colors and palette combinations. It works on computers equipped with CGA, EGA, MCGA, or VGA cards. A color monitor is required.

```
PROGRAM cga;
{ Demonstrates CGA colors }

USES
    MSGraph;

CONST
    bkcolor : ARRAY [0..7] OF LongInt = (_Black, _Blue, _Green, _Cyan,
              _Red, _Magenta, _Brown, _White);

    bkcolor_name : ARRAY [0..7] OF STRING = ('BLACK  ', 'BLUE   ',
                   'GREEN  ', 'CYAN   ', 'RED    ', 'MAGENTA',
                   'BROWN  ', 'WHITE  ');

VAR
    a, i, j, k : Integer;

BEGIN  { Begin main program }
    a := _SetVideoMode( _MRes4Color );
    FOR i := 0 TO 3 DO
        BEGIN
        a := _SelectPalette( i );
        FOR k := 0 TO 7 DO
            BEGIN
            _SetBkColor( bkcolor[k] );
            FOR j := 0 TO 3 DO
                BEGIN
                _SetTextPosition( 1, 1 );
                Writeln( 'Background color: ', bkcolor_name[k] );
                Writeln( '   Palette: ', i );
                Writeln( '   Number: ', j );
                _SetColor( j );
                _Rectangle( _GFillInterior, 160, 100, 320, 200 );
                Readln;  { Wait for ENTER to be pressed }
                END; { for j }
            END; { for k }
        END; { for i }

    a := _SetVideoMode( _DefaultMode ); { restore original palette }
END.
```

13.3.3 EGA, MCGA, and VGA Palettes

The beginning of this chapter mentioned the difference between color indexes and color values. An analogy might make things clearer. Imagine a painter who owns 64 tubes of paint and a painter's palette that has room for only 16 globs of paint at any one time. A painting created under these constraints could contain

only 16 colors (selected from a total of 64). The EGA graphics modes (such as **_EResColor**) are similar: they have 16 color indexes chosen from a total of 64 color values.

The color values available in the EGA, MCGA, and VGA palettes are not predetermined like the CGA palettes. Instead, the color values available in EGA, MCGA, and VGA palettes are created by a process of "color mixing" of red, green, and blue elements. The next two sections describe color mixing.

VGA Color Mixing VGA offers the widest variety of color values: 262,144 (256K). Depending on the graphics mode, the VGA palette size may be 2, 16, or 256. When you select a color value, you specify a level of intensity ranging from 0 – 63 for each of the red, green, and blue color values. The long integer that defines a color value consists of four bytes (32 bits):

```
MSB                                 LSB
zzzzzzzz zzBBBBBB zzGGGGGG zzRRRRRR
```

The most-significant byte must contain all zeros. The two high bits in the remaining three bytes must also be 0. To mix a light red (pink), turn red all the way up, and mix in some green and blue:

```
00000000 00100000 00100000 00111111
```

To represent this value in hexadecimal, use the number `$0020203F`.

For white, turn all the elements on; for black, set all elements to 0.

EGA Color Mixing Mixing colors in EGA modes is similar to the mixing described above, but there are fewer intensities for the red, green, and blue components. In the modes that offer 64 colors, the R, G, and B values cover 2 bits and can range from 0 – 3. The long integer that defines an RGB color looks like this:

```
MSB                                 LSB
zzzzzzzz zzBB???? zzGG???? zzRR????
```

The bits marked `z` must be zeros and the bits marked with question marks can be any value. This format is used for compatibility with VGA color mixing. To form a pure red color value, you would use the constant `$00000030`. For cyan (blue plus green), use `$00303000`.

13.3.4 EGA Color Graphics Modes

If you have an EGA adapter, you should use the video mode **_MRes16Color**, **_HRes16Color**, or **_EResColor** for the best color-graphics results. The CGA modes also display on the EGA but with the lower CGA resolution and decreased color options.

The **_RemapPalette** function assigns a new color value to a color index. For example, when you first enter an EGA graphics mode, color index 1 equals the color value blue. To reassign the pure red color value to color index 1, you could use this line:

```
a := _RemapPalette( 1, $000030 );
```

Or, use the symbolic constant **_Red**, which is defined using the **MSGraph** unit:

```
a := _RemapPalette( 1, _Red );
```

After this function call, any object currently drawn in color index 1 instantly switches from blue to red.

The first value is an **Integer** in the range 0 –15 and the second value is a **LongInt** defined as a mixture of red, green, and blue (you may also use symbolic constants such as **_Red**).

The **_RemapAllPalette** procedure changes all of the color indexes simultaneously. You pass it an array of color values. The first color value in the list becomes the new color associated with the color index 0.

The number in a function call that sets the color (such as **_SetColor**) is an index into the palette of available colors. In the default text palette, an index of 1 refers to blue but the palette could be remapped to change index 1 to any other available color. As a result, the color produced by that pixel value also changes. The number of color indexes depends on the number of colors supported by the current video mode.

The **_RemapPalette** and **_RemapAllPalette** routines work in all modes but only with the EGA, MCGA, or VGA hardware. The **_RemapPalette** fails, returning a value of –1 when you attempt to remap a palette without the EGA, MCGA, or VGA hardware.

The following program draws a rectangle with a red interior. In the default EGA palette, the color index 4 is red. This color index is changed to **_Blue** in this program.

```
PROGRAM ega;
{ Demonstrates EGA/VGA palettes }

USES
    MSGraph;

CONST
    crlf = #13 + #10;

VAR
    a : LongInt;
    m : Integer;
```

```
BEGIN  { Begin main program }

    m := _SetVideoMode( _MaxColorMode );
    _SetColor( 4 );
    _Rectangle( _GFillInterior, 50, 50, 150, 150 );

    _SetTextPosition( 1, 1 );
    _OutText( 'Normal palette' + crlf );
    _OutText( 'Press ENTER' );
    Readln;

    a := _RemapPalette( 4, _Blue );

    _SetTextPosition( 1, 1 );
    _OutText( 'Remapped palette' + crlf );
    _OutText( 'Press ENTER' );
    Readln;

    a := _RemapPalette( 4, _Red );

    _SetTextPosition( 1, 1 );
    _OutText( 'Restored palette' + crlf );
    _OutText( 'Press ENTER to clear the screen' );
    Readln;

    _ClearScreen( _GClearScreen );
    m := _SetVideoMode( _DefaultMode );
END.
```

13.3.5 VGA Color Graphics Modes

The VGA card adds graphics modes **_VRes2Color**, **_VRes16Color**, and **_MRes256Color** to your repertoire. EGA and CGA modes can also be used with the VGA hardware, but with either lower resolution or fewer color choices.

The VGA color graphics modes operate with a range of 262,144 (256K) color values. The **_VRes2Color** graphics mode displays two colors, the **_VRes16Color** graphics mode displays 16 colors, and the **_MRes256Color** graphics mode displays 256 colors from the available VGA colors.

The **_RemapPalette** function changes a color index to a specified color value. The function below remaps the color index 1 to the color value given by the symbolic constant **_Red** (which represents red). After this statement is executed, whatever was displayed as blue now appears as red:

```
{ reassign color index 1 to VGA red }
a := _RemapPalette( 1, _Red );
```

Use the **_RemapAllPalette** procedure to remap all of the available color indexes simultaneously. The procedure's argument references an array of color values that reflects the remapping. The first color number in the list becomes the new color associated with color index 0.

Symbolic constants for the default color numbers are supplied so that the remapping of VGA colors is compatible with EGA practice. The names of these constants are self-explanatory. For example, the color numbers for black, red, and light yellow are represented by the symbolic constants **_Black**, **_Red**, and **_Yellow**.

All of the VGA display modes operate with any VGA video monitor. Colors are displayed as shades of gray when a monochrome analog display is connected.

The program HORIZON.PAS illustrates what can be done with the range of 256 colors if you have a VGA card:

```
PROGRAM horizon;  { Demonstrates VGA graphics with cycling of 256 colors }

USES
    Crt, MSGraph;
CONST
    Red = $00002a;
    GRN = $002a00;
    BLU = $2a0000;
    WHT = $2a2a2a;
    step = 21;
VAR
    vc : _VideoConfig;
    rainbow : ARRAY [1..512] OF LongInt;
    i, a      : Integer;
    col, gray : LongInt;
    rec : _XYCoord;

BEGIN  { Begin main program }
    a := _SetVideoMode( _MRes256Color );
    IF (a = 0) THEN
        BEGIN
        Writeln( 'This program requires a VGA or MCGA card' );
        Halt( 0 );
        END;

    FOR col := 0 TO 63 DO
        BEGIN
        gray := col OR (col SHL 8) OR (col SHL 16);
        rainbow[col] :=   BLU AND gray;
        rainbow[col + 256] := BLU AND gray;
        rainbow[col + 64] := BLU OR gray;
        rainbow[col + 64 + 256] := BLU OR gray;
        rainbow[col + 128] := Red OR (WHT AND NOT gray);
        rainbow[col + 128 + 256] := Red  OR (WHT AND NOT gray);
        rainbow[col + 192] := Red OR NOT gray;
        rainbow[col + 192 + 256] := Red OR NOT gray;
        END;
```

```
    _SetViewOrg( 160, 85, rec );

        FOR i := 0 TO 254 DO
            BEGIN
            _SetColor( 255 - i );
            _MoveTo( i, i - 255 );
            _LineTo( -i, 255 - i );
            _MoveTo( -i, i - 255 );
            _LineTo( i, 255 - i );
            _Ellipse( _GBorder, -i, -i DIV 2, i, i DIV 2 );
            END;

        i := 0;
        WHILE NOT KeyPressed DO
            BEGIN
            _RemapAllPalette( rainbow[i] );
            i := i + step;
            IF (i >= 256) THEN i := 0;
            END;

        a := _SetVideoMode( _DefaultMode );
    END.
```

13.3.6 Using the Color Video Text Modes

All video adapters offer support for video text modes (**_TextC40**, **_TextC80**, **_TextBW40**, **_TextBW80**, **_TextMono**). Using the video text modes, you can display color text without having to enter a graphics mode. On a color monitor, you can display normal or blinking text in any of 16 foreground colors with any of 8 background colors. On a monochrome monitor, you can specify text "colors" in exactly the same way, although the color values are interpreted differently by the hardware.

The text procedures and functions described in this section can be used in the video text modes as well as in all of the graphics modes.

Selecting Text Colors

In a video text mode, each displayed character requires two bytes of video memory. The first byte contains the ASCII code representing the character and the second byte contains the display attribute. In the CGA color video text modes, the attribute byte determines the color and whether it will blink. Sixteen

colors are available: the CGA indexes, and the default EGA and VGA indexes. Since the EGA and VGA palettes can be remapped, these values can be made to correspond to any set of 16 colors with the appropriate palette mapping.

Using Text Colors

Use the **_GetTextColor** and **_GetBkColor** functions to find foreground and background colors of the current text.

Values in the range 0 –15 are interpreted as normal color. Values in the range 16 –31 are the same colors as those in the range 0 –15 but with blinking text.

Set the foreground and background colors in a video text mode with the **_SetTextColor** and **_SetBkColor** functions. These functions use a single argument that specifies the index to be used for text displayed with the **_OutText** procedure. The color indexes for color video text modes are defined in Table 13.6.

Table 13.6 Text Colors

Number	Color	Number	Color
0	Black	8	Gray
1	Blue	9	Light Blue
2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Brown	14	Yellow
7	White	15	Light White

On monochrome displays, you can select text attributes in the same way as for color displays. For underlined text, use Blue or Light Blue. For highlighted text, use any color from Light Green to White. For blinking text, add 16 to the color constant.

Displaying Text Colors

The **_SetTextPosition** procedure moves the cursor to a row and column for displaying color text. The **_OutText** procedure displays the text.

The following program displays a chart showing all possible combinations of text and background colors:

```
PROGRAM coltext;  { Displays text in color }

USES
    Crt, MSGraph;

VAR
    message1, s, t : STRING;
    blink, fgd, bgd : Integer;
    a : Integer;

BEGIN  { Begin main program }

    _ClearScreen( _GClearScreen );
    _OutText( 'Color text attributes:' );

    FOR blink := 0 TO 1 DO
        FOR bgd := 0 TO 6 DO
            BEGIN
            _SetBkColor( bgd );
            _SetTextPosition( bgd + (blink  * 9) + 3, 1 );
            _SetTextColor( 7 );
            Str( bgd, s );
            message1 := 'Bgd: ';
            _OutText( message1 );
            _OutText( s );
            s := s + ' ';
            message1 := '  Fgd: ';
            _OutText( message1 );
            FOR fgd := 0 TO 15 DO
                BEGIN
                _SetTextColor( fgd + (blink*16) );
                a := fgd + (blink*16);
                Str( a, s );
                s := s + '  ';
                _OutText( s );
                END;
            END; { FOR loops }

    REPEAT UNTIL KeyPressed;
    a := _SetVideoMode( _DefaultMode );

END.
```

13.4 Understanding Coordinate Systems

Before you can write a program to print a word or to display graphics on the screen, you need a system that describes exactly where to print or display the item.

QuickPascal provides several coordinate systems. The physical coordinate system has already been used in the program 1STGRAPH.PAS. This section discusses the four coordinate systems supported by QuickPascal and shows how to translate between systems.

The coordinate systems supported by QuickPascal are:

- Text coordinates
- Physical coordinates
- Viewport coordinates
- Real coordinates in a window

13.4.1 Text Coordinates

QuickPascal divides the text screen into rows and columns. See Figure 13.1.

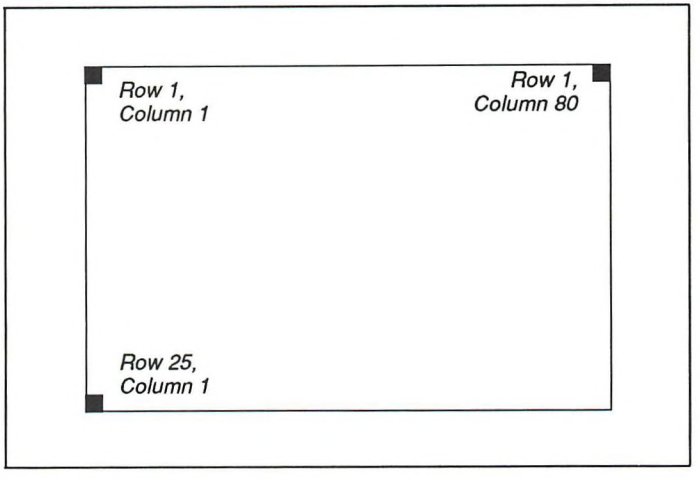

Figure 13.1 Text Screen Coordinates

Two important conventions to keep in mind about video text mode are:

1. Numbering starts at 1, not 0. An 80-column screen contains columns 1–80.
2. The row is always listed before the column.

If the screen is in a video text mode that displays 25 rows and 80 columns (as in Figure 13.1), the rows are numbered 1–25 and the columns are numbered 1–80. In routines such as **_SetTextPosition**, which is called in the next example program, the parameters you pass are row and column (in that order). Some monitors (such as the EGA or VGA) support more than 25 text rows. For these monitors, use the **_SetVideoModeRows** or **_SetTextRows** functions to specify the number of rows to display.

13.4.2 Physical Screen Coordinates

Suppose you write a program that calls **_SetVideoMode** and puts the screen into the VGA graphics mode **_VRes16Color**. This gives you a screen containing 640 horizontal pixels and 480 vertical pixels. The individual pixels are named by their location relative to the *x* axis and *y* axis, as shown in Figure 13.2.

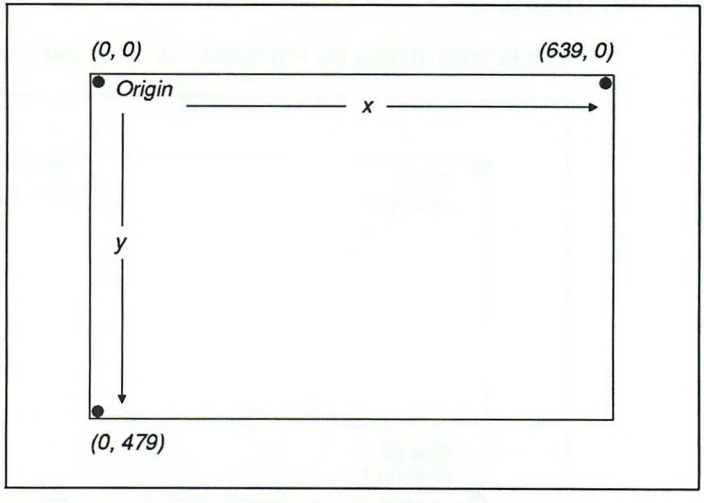

Figure 13.2 Physical Screen Coordinates

Two important differences between text coordinates and graphics coordinates are:

1. Numbering starts at 0, not 1. If there are 640 pixels, they're numbered 0 – 639.
2. The *x* coordinate is listed before the *y* coordinate.

The upper left corner is called the "origin." The *x* and *y* coordinates for the origin are always (0, 0). If you use variables to refer to pixel locations, declare them as integers. The "viewport" is the region where graphics will be displayed.

Changing the Origin with _SetViewOrg

The **_SetViewOrg** procedure changes the location of the viewport's origin. You pass two integers, which represent the *x* and *y* coordinates of a physical screen location. For example, the following line would move the origin to the physical screen location (50, 100):

```
_SetViewOrg( 50, 100, xyorg );
```

The effect on the screen is illustrated in Figure 13.3.

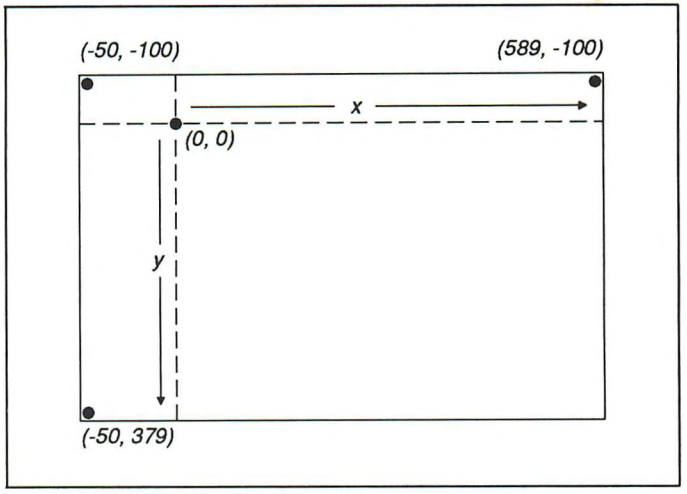

Figure 13.3 Coordinates Changed by _SetViewOrg

The number of pixels hasn't changed, but the names given to the points have changed. The *x* axis now ranges from –50 to +589 instead of 0 to 639. The *y* axis now covers the values –100 to +379. (If you own an adapter other than the VGA, the numbers are different but the effect is the same.)

All standard graphics functions and procedures are affected by the new origin, including **_MoveTo**, **_LineTo**, **_Rectangle**, **_Ellipse**, **_Arc**, and **_Pie**.

For example, if you call the **_Rectangle** procedure after relocating the viewport origin, and pass it the coordinate values (0, 0) and (40, 40), the rectangle would be drawn 50 pixels from the left edge of the screen and 100 pixels from the top. It would not appear in the upper left corner.

The values passed to **_SetViewOrg** are always physical screen locations. Suppose you called the same procedure twice:

```
_SetViewOrg( 50, 100, xyorg );
_SetViewOrg( 50, 100, xyorg );
```

The viewport origin would *not* move to (100, 200). It would remain at the physical screen location (50, 100).

Defining a Clipping Region with _SetClipRgn

The **_SetClipRgn** procedure creates an invisible rectangular area on the screen called a "clipping region." Attempts to draw inside the clipping region are successful, while attempts to draw outside the region are not.

When you first enter a graphics mode, the default clipping region is the entire screen. QuickPascal ignores any attempts to draw outside the clipping region.

Changing the clipping region requires one call to **_SetClipRgn**. Suppose you've entered the CGA graphics mode **_MRes4Color**, which has a screen resolution of 320 × 200. If you draw a diagonal line from the top left to the bottom right corner, the screen looks like Figure 13.4.

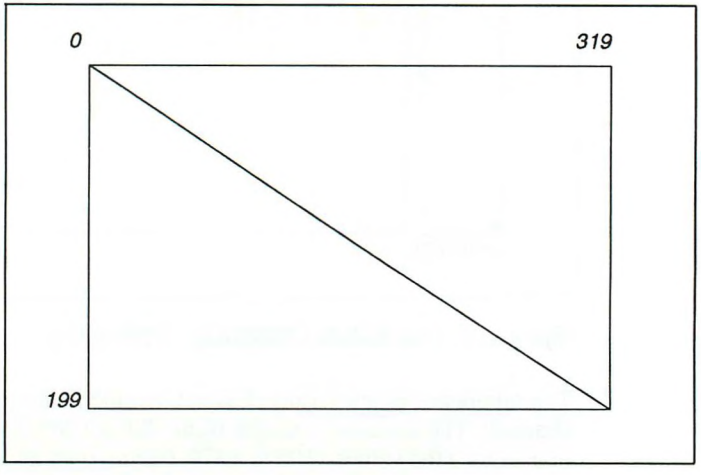

Figure 13.4 Line Drawn on a Full Screen

You could create a clipping region with the following:

```
_SetClipRgn( 10, 10, 309, 189 );
```

Then draw a line with the statement:

```
_LineTo( 0, 0, 319, 199 );
```

With the clipping region in effect, the same **_LineTo** command given above would put the line shown in Figure 13.5 on the screen.

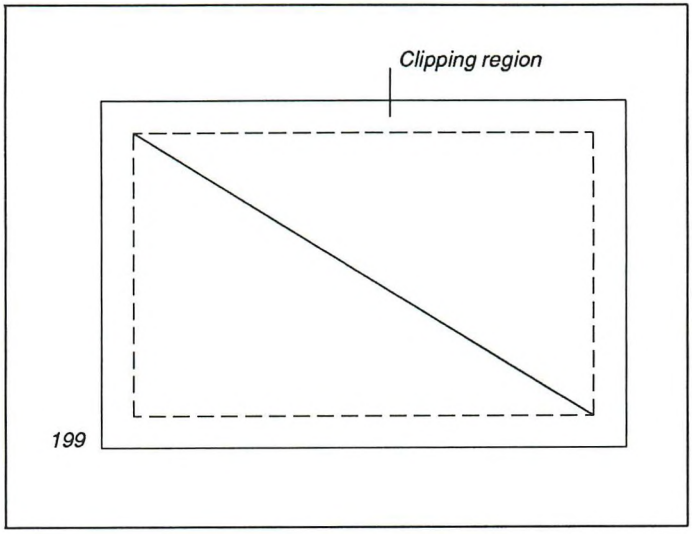

Figure 13.5 Line Drawn within a Clipping Region

The broken lines don't actually appear on the screen. They indicate the outer bounds of the clipping region.

13.4.3 Viewport Coordinates

Viewport coordinates are yet another coordinate system supported by Quick-Pascal. You can establish a new viewport within the boundaries of the physical screen by using the **_SetViewport** procedure. A standard viewport has two distinguishing features:

1. The origin of a viewport is in the upper left corner.
2. The clipping region matches the outer boundaries of the viewport.

The **_SetViewport** procedure does the same thing as calling **_SetViewOrg** and **_SetClipRgn** together.

13.4.4 Real Coordinates in a Window

QuickPascal supports a system of real coordinates for use in a window. This system lets you use floating-point values in graphics.

Functions and procedures that refer to coordinates on the physical screen and within the viewport require integer values. However, in real-life graphing applications, you might wish to use floating-point values—stock prices, the price of wheat, average rainfall, and so on.

Setting Window Coordinates

The **_SetWindow** procedure allows you to scale the screen to almost any size. In addition, the window-related functions and procedures take double-precision floating-point values instead of integers.

If, for example, you wanted to graph 12 months of average temperatures that range from – 40 to +100, you could add the following line to your program:

```
_SetWindow( TRUE, 1.0, -40.0, 12.0, 100.0 );
```

The first argument is the invert flag, which puts the lowest *y* value in the bottom left corner. The minimum and maximum Cartesian coordinates follow (the decimal point marks them as floating-point values). The new organization of the screen is shown in Figure 13.6.

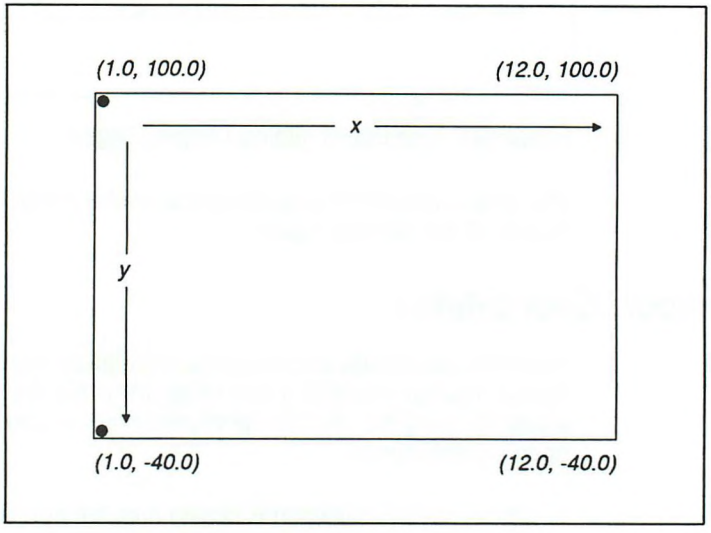

Figure 13.6 Window Coordinates

This procedure makes the temperatures for January and December appear on the left and right edges of the screen. In an application like this, it might be better to number the *x* axis from 0.0 to 13.0, to provide some extra space.

If you plot a point with **_SetPixel_w** or draw a line with **_LineTo_w**, the values are automatically scaled to the established window. You can also find the position of the graphics cursor at any time with **_GetCurrentPosition_wxy**.

Programming Real-Coordinate Graphics

The four steps to using real-coordinate graphics are:

1. Enter a graphics mode with **_SetVideoMode**.
2. Use **_SetViewPort** to create a viewport area. This step is optional if you plan to use the entire screen.
3. Create a real-coordinate window with **_SetWindow**, passing a Boolean invert flag and four **Double** *x* and *y* coordinates for the minimum and maximum values.
4. Draw graphics shapes with **_Rectangle_w** and other procedures. Do not confuse **_Rectangle** (the viewport procedure) with **_Rectangle_w** (the window procedure for drawing rectangles). All window procedures end with an underscore and a letter **w** or an underscore and **wxy**.

Real-coordinate graphics can give you a lot of flexibility. For example, you can fit either axis into a small range (such as 151.25 to 151.45) or into a large range (–50,000 to +80,000), depending on the type of data you're graphing. In addition, by changing the window coordinates, you can create the effects of zooming in or panning across a figure.

An Example of Real-Coordinate Graphics

The program below illustrates how to use the real-coordinate window routines.

```
PROGRAM realg; {Example of real-coordinate graphics}
USES
    MsGraph, Crt;

CONST
    bananas : ARRAY[0..20] OF Single =
        (
        -0.3,    -0.2, -0.224, -0.1,    -0.5, +0.21, +2.9,
        +0.3,    +0.2,  0.0,   -0.885, -1.1, -0.3,  -0.2,
         0.001, 0.005, 0.14,    0.0,   -0.9, -0.13, +0.3
         );
VAR
    halfx, halfy,
    a   : Integer;
    vc  : _VideoConfig;
    ch  : Char;
```

```
FUNCTION four_colors : Boolean;
BEGIN
    four_colors := False;
    IF (_SetVideoMode( _MaxColorMode ) > 0) THEN
        BEGIN
        _GetVideoConfig( vc );
        IF (vc.NumColors >= 4) THEN
            four_colors := True;
        END;

END;

PROCEDURE grid_shape;
VAR
    i, x1, y1, x2, y2  : Integer;
    x, y               : Real;
    s                  : STRING[80];
BEGIN

    FOR i := 1 TO vc.NumColors DO
        BEGIN
        _SetTextPosition( i, 2 );
        _SetTextColor( i );
        Str( i, s );
        _OutText( 'Color ' + s );
        END;

    _SetColor( 1 );
    _Rectangle_w( _GBorder, -1.0,  -1.0,  1.0,  1.0  );
    _Rectangle_w( _GBorder, -1.02, -1.02, 1.02, 1.02 );

    x := -0.9;
    i :=  0;
    WHILE  x < 0.9 DO
        BEGIN
        _SetColor( 2 );
        _MoveTo_w( x, -1.0 );    _LineTo_w( x, 1.0 );
        _MoveTo_w( -1.0, x );    _LineTo_w( 1.0, x );
        _SetColor( 3 );
        _MoveTo_w( x - 0.1, bananas[i] );
        Inc( i );
        _LineTo_w( x, bananas[i] );
        x := x + 0.1;
        END;

    _MoveTo_w( 0.9, bananas[i] );
    Inc(i);
    _LineTo_w( 1.0, bananas[i] );
    END;

PROCEDURE three_graphs;
VAR
    upleft, botright : _WXYCoord;
    xwidth, yheight, cols, rows : Integer;
```

```
BEGIN
    _ClearScreen( _GClearScreen );
    xwidth  := vc.NumXPixels;
    yheight := vc.NumYPixels;
    halfx   := xwidth  DIV 2;
    halfy   := yheight DIV 2;
    cols    := vc.NumTextCols;
    rows    := vc.NumTextRows;

    { first window }
    _SetViewport( 0, 0, halfx-1, halfy-1 );
    _SetTextWindow( 1, 1, rows DIV 2, cols DIV 2 );
    _SetWindow( False, -2.0, -2.0, 2.0, 2.0 );
    grid_shape;
    _Rectangle( _GBorder, 0, 0, halfx-1, halfy-1 );

    { second window }
    _SetViewport( halfx, 0, xwidth-1, halfy-1 );
    _SetTextWindow( 1, cols DIV 2+1, rows DIV 2, cols );
    _SetWindow( False, -3.0, -3.0, 3.0, 3.0 );
    grid_shape;
    _Rectangle_w( _GBorder, -3.0, -3.0, 3.0, 3.0 );

    { third window }
    _SetViewport( 0, halfy, xwidth-1, yheight-1 );
    _SetTextWindow( rows DIV 2+2, 1, rows, cols );
    _SetWindow( True, -3.0, -1.5, 1.5, 1.5 );
    grid_shape;
    upleft.wx   := -3.0;
    upleft.wy   := -1.5;
    botright.wx :=  1.5;
    botright.wy :=  1.5;
    _Rectangle_wxy( _GBorder, upleft, botright);

END;

BEGIN { main program }

IF four_colors THEN
    BEGIN
    _OutText( 'This program requires a CGA, EGA, or VGA graphics card' );
    three_graphs;
    END;

ch := ReadKey;
a := _SetVideoMode( _DefaultMode );

END.
```

The main body of the program is short. It calls the `four_colors` function (defined below), which attempts to enter a graphics mode in which at least four colors are available. If it succeeds, the `three_graphs` function is called. This function uses the numbers in the `bananas` array to draw three graphs. The REALG.PAS screen output is shown in Figure 13.7.

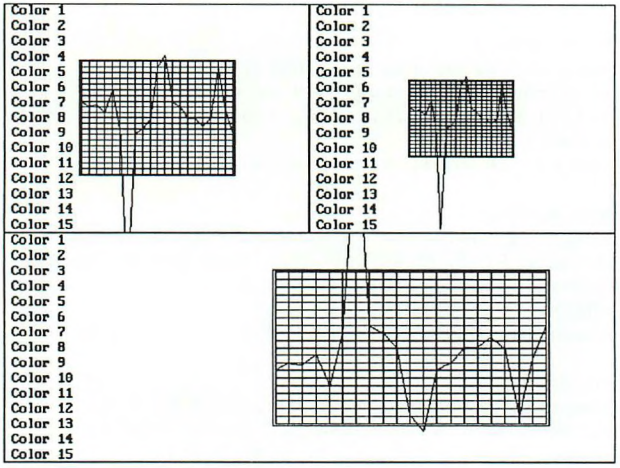

Figure 13.7 REALG.PAS Program

It's worth noting that the graphs are all drawn using the same numbers. However, the program uses three different real-coordinate windows. The two windows in the top half are the same size in physical coordinates, but they have different window sizes. In all three cases, the grid is two units wide. In the upper left corner, the window is four units wide; in the upper right, the window is six units wide, which makes the graph appear smaller.

In two of the three graphs, one of the lines goes off the edge, outside the clipping region. The lines do not intrude into the other windows, since defining a window creates a clipping region.

Finally, note that the graph on the bottom of the screen seems to be upside down with respect to the two graphs above it. This is the result of setting the invert flag to **True**.

Entering a Graphics Mode

The first step in any graphics program is to enter a graphics mode. The function `four_colors` performs this step:

```
FUNCTION four_colors : Boolean;
BEGIN
    four_colors := False;
    IF (_SetVideoMode( _MaxColorMode , > 0) THEN
        BEGIN
        _GetVideoConfig( vc );
        IF (vc.NumColors >= 4) THEN
            four_colors := True;
        END;

END;
```

The **_GetVideoConfig** procedure places some information into the structure **_VideoConfig** called `screen`. Then you use the member `screen.adapter` of the **_VideoConfig** structure in a **CASE** statement construct to turn on the matching graphics mode. The symbolic constants **_CGA** and the rest are defined in the **MSGraph** unit. The modes containing the letter "O" are Olivetti modes.

If the computer is equipped with a color card, this function returns a **True**. If it is not, it returns a **False**, which causes the program to skip the function `three_graphs` and to end the program.

If the `four_colors` function works properly, the program calls the function below, which prints the three graphs.

```
PROCEDURE three_graphs;
VAR
    upleft, botright : _WXYCoord;
    xwidth, yheight, cols, rows : Integer;

BEGIN
    _ClearScreen( _GClearScreen );
    xwidth  := vc.NumXPixels;
    yheight := vc.NumYPixels;
    halfx   := xwidth  DIV 2;
    halfy   := yheight DIV 2;
    cols    := vc.NumTextCols;
    rows    := vc.NumTextRows;
```

```
{ first window }
    _SetViewport( 0, 0, halfx-1, halfy-1 );
    _SetTextWindow( 1, 1, rows DIV 2, cols DIV 2 );
    _SetWindow( False, -2.0, -2.0, 2.0, 2.0 );
    grid_shape;
    _Rectangle( _GBorder, 0, 0, halfx-1, halfy-1 );

    { second window }
    _SetViewport( halfx, 0, xwidth-1, halfy-1 );
    _SetTextWindow( 1, cols DIV 2+1, rows DIV 2, cols );
    _SetWindow( False, -3.0, -3.0, 3.0, 3.0 );
    grid_shape;
    _Rectangle_w( _GBorder, -3.0, -3.0, 3.0, 3.0 );

    { third window }
    _SetViewport( 0, halfy, xwidth-1, yheight-1 );
    _SetTextWindow( rows DIV 2+2, 1, rows, cols );
    _SetWindow( True, -3.0, -1.5, 1.5, 1.5 );
    grid_shape;
    upleft.wx   := -3.0;
    upleft.wy   := -1.5;
    botright.wx :=  1.5;
    botright.wy :=  1.5;
    _Rectangle_wxy( _GBorder, upleft, botright);

END;
```

Working with Windows

Although entering a graphics mode automatically clears the screen, it doesn't hurt to be sure, so `three_graphs` calls the **_ClearScreen** procedure:

```
_ClearScreen( _GClearScreen );
```

The **_GClearScreen** constant causes the entire physical screen to clear. Other options include **_GViewport** and **_GWindow**, which clear the current viewport and the current text window, respectively.

The First Window After assigning values to some variables, the procedure `three_graphs` creates the first window:

```
_SetViewPort( 0, 0, halfx - 1, halfy - 1 );
_SetTextWindow( 1, 1, rows / 2, cols / 2 );
_SetWindow( False, -2.0, -2.0, 2.0, 2.0 );
```

First a viewport is defined to cover the upper left quarter of the screen. Next, a text window is defined within the boundaries of that border. (Note the numbering starts at 1 and the row location precedes the column.) Finally, a window is defined. The **False** constant forces the *y* axis to increase from top to bottom. The corners of the window are (–2.0, –2.0) in the upper left and (2.0, 2.0) in the bottom right corner.

Next, the function `grid_shape` is called, and a border is added to the window:

```
grid_shape;
_Rectangle( _GBorder, 0, 0, halfx-1, halfy-1 );
```

Note that this is the standard **_Rectangle** procedure, which takes coordinates relative to the viewport (*not* window coordinates).

Two More Windows The two other windows are similar to the first. All three call `grid_shape` (defined below), which draws a grid from location (–1.0, –1.0) to (+1.0, +1.0). The grid appears in different sizes because the coordinates in the windows vary. The second window ranges from (–3.0, –3.0) to (+3.0, +3.0), so the width of the grid is one-third the width of the second window, while it is one-half the width of the first.

Note also that the third window contains **True** as the first argument. This causes the *y* axis to increase from bottom to top, instead of top to bottom. As a result, this graph appears to be upside down in relation to the other two.

After calling `grid_shape`, the program frames each window with one of the following procedures:

```
_Rectangle( _GBorder, 0, 0, halfx -1, halfy -1 );
_Rectangle_w( _GBorder, -3.0, -3.0, 3.0, 3.0 );
_Rectangle_wxy( _GBorder, upleft, botright );
```

All three procedures contain a fill flag as the first argument. The procedure **_Rectangle** takes integer arguments that refer to the viewport screen coordinates. The procedure **_Rectangle_w** takes four double-precision, floating-point values referring to window coordinates: upper left *x*, upper left *y*, lower right *x*, and lower right *y*. The procedure **_Rectangle_wxy** takes two arguments: the addresses of structures of type **_WXYCoord**, which contains two **Double** types named **wx** and **wy**. The structure is defined in the **MSGraph** unit. The values are assigned just before **_Rectangle_wxy** is called.

Drawing Graphics

The `grid_shape` procedure is shown here:

```
PROCEDURE grid_shape;
VAR
    i, x1, y1, x2, y2  : Integer;
    x, y               : Real;
    s                  : STRING[80];
BEGIN

    FOR i := 1 TO vc.NumColors DO
        BEGIN
        _SetTextPosition( i, 2 );
        _SetTextColor( i );
        Str( i, s );
        _OutText( 'Color ' + s );
        END;

    _SetColor( 1 );
    _Rectangle_w( _GBorder, -1.0,  -1.0,  1.0,  1.0  );
    _Rectangle_w( _GBorder, -1.02, -1.02, 1.02, 1.02 );

    x := -0.9;
    i :=  0;
    WHILE  x  0.9 DO
        BEGIN
        _SetColor( 2 );
        _MoveTo_w( x, -1.0 );   _LineTo_w( x, 1.0 );
        _MoveTo_w( -1.0, x );   _LineTo_w( 1.0, x );
        _SetColor( 3 );
        _MoveTo_w( x - 0.1, bananas[i] );
        Inc( i );
        _LineTo_w( x, bananas[i] );
        x := x + 0.1;
        END;

    _MoveTo_w( 0.9, bananas[i] );
    Inc(i);
    _LineTo_w( 1.0, bananas[i] );
    END;
```

First, the number of available color indexes is assigned to the `numc` variable and a loop displays all of the available colors:

```
FOR i := 1 TO numc DO

    BEGIN
    _SetTextPosition( i, 2 );
    _SetTextColor( i );
    Str(i, s);
    _OutText( 'Color ' + s );
    END;
```

The names of the procedures are self-explanatory. The advantage of using **_OutText** in graphics mode is that you can control the text color and limit output to the currently defined text window.

The procedure and function names that end with **_w** work the same as their viewport equivalents, except you pass double-precision, floating-point values instead of integers. For example, you pass integers to **_LineTo** but floating-point values to **_LineTo_w**.

13.5 Animation

The QuickPascal **MSGraph** unit provides several functions that can be used to animate your graphics programs. These functions provide two means of animation:

1. Video-page animation (also used in text modes)
2. Bit-mapped animation

"Video-page animation" takes advantage of the fact that the EGA, VGA, and Hercules video cards have enough memory to store more than one video display page. You can animate by switching between the pages. "Bit-mapped animation" captures bit-mapped images of the screen and then stores them in a memory buffer. These images can be redisplayed at a new location to perform animation.

This section discusses these two animation techniques and shows two sample programs that bring your graphics to life.

13.5.1 Video-Page Animation

Most video adapters contain enough memory so that more than one display page can be stored at a time. The **MSGraph** unit provides several functions that allow you to manipulate these video pages. Two terms are used to describe these pages—the "active page" is the page where text and graphics commands operate; the "visual page" is the page that you see displayed.

The number of video pages available in an adapter depends on the amount of video memory on the adapter and the mode in which the adapter is used. For example, the CGA adapter with 16K of video memory supports four video pages in the text mode **_TextC80**. An EGA adapter with a full 256K of video memory has room for two video pages even in the high resolution **_EResColor** mode. Virtually all adapters provide for multiple pages in text modes; only the EGA and VGA adapters with 256K of video memory support two video pages in the high-resolution graphics modes. The Hercules graphics mode (**_HercMono**) can support two pages, but only if it is the only graphics adapter present and only when MSHERC.COM is started without the /H option.

Use the procedure **_GetVideoConfig** to obtain information about the video configuration. After calling **_GetVideoConfig**, use the **NumVideoPages** element of the **_VideoConfig** structure to determine the number of video pages supported.

A simple use of video pages is to draw graphics offscreen (on the active page) and then make this active page the visual page. In this way, you do not see the process of creating the graphics; you only see the final result. This process of drawing offscreen and then switching can be extended to provide animation.

The procedure **_SetVisualPage** changes the page that you see. The procedure **_SetActivePage** changes the page where drawing takes place. The first page in any graphics system is number 0, the second is number 1, and so on. A corresponding set of functions **_GetActivePage** and **_GetVisualPage** return the value of the current active or visual page, respectively.

To animate any sequence of screens using video-page animation, use the following steps:

1. Perform regular graphics initialization (select video mode, check for error, and so on).
2. Draw on the active page.
3. Swap the visual and active pages.
4. Repeat steps 2 and 3 until finished with animation.
5. Restore the screen and exit the program.

The example program PAGES.PAS uses four video pages to animate a simple set of character images in text mode.

```
PROGRAM page_animation;
{ PAGES.PAS: Use video pages to animate screens }

USES
    MSGraph, Crt;

CONST
    jumper    :  ARRAY[0..3] OF STRING =
              ('/O\',  '-O-', '\O/', 'WOW');
VAR
    a, i      :  Integer;
    oldvpage  :  Integer;
    oldapage  :  Integer;
    vc : _VideoConfig;
    oldcursor : Boolean;

BEGIN  { Begin main program }

    _ClearScreen( _GClearScreen);

    oldapage  := _GetActivePage;
    oldvpage  := _GetVisualPage;
```

```
{ Set the video mode for a large text size }
    a := _SetVideoModeRows( _TextBW40, 25 );
    _GetVideoConfig( vc );

    IF ((a = 0) OR (vc.NumVideoPages < 4)) THEN
        BEGIN
        Writeln( '_TEXTBW40 mode not available; hit Return to continue' );
        Readln;
        a := _SetVideoMode( _DefaultMode );
        Halt( 0 );
        END;

    { Turn off flashing cursor. }
    oldcursor := _DisplayCursor( False );

    { Draw image on each page. }
    FOR i := 0 TO 3 DO
        BEGIN
        _SetActivePage( i );
        _SetTextPosition( 12, 20 );
        _OutText( jumper[i] );
        END;

    { Cycle through pages 0 to 3. }
    REPEAT
        FOR i := 0 TO 3 DO
            BEGIN
            _SetVisualPage( i );
            Delay( 500 );
            END;
    UNTIL KeyPressed;

    { Restore everything before ending the program. }
    a := _SetVideoMode( _DefaultMode );
    _SetActivePage( oldapage );
    _SetVisualPage( oldvpage );

END.
```

The program PAGES.PAS begins with a call to **_ClearScreen** and then sets the video mode using the **_SetVideoModeRows** function. The cursor is turned off with the **_DisplayCursor** function.

The **FOR** loop draws a graphics image on each of four pages. The loop goes to each of the video pages and outputs the graphics image using the **_OutText** procedure. When you run this program, notice that you don't see any of this activity. Since the drawing is taking place on the active pages, it is not visible on the screen (the visual page).

Once the pages have been drawn, the **REPEAT** loop cycles through each page and makes it the visual page (so you can see it) and then delays 500 milliseconds between calls to **_SetVisualPage**.

Finally, when a key is pressed, the program ends by restoring the video mode to **_DefaultMode** using the **_SetVideoMode** function.

13.5.2 Bit-Mapped Animation

Bit-mapped animation gives you the ability to draw graphics figures and store them in memory for later use in animation. The **_ImageSize** function determines the amount of memory required to store a specified bit-mapped image. The image is specified in terms of a bounding rectangle. The **_GetImage** procedure copies the bit map of pixels inside a specified rectangle to a buffer area in memory. The **_PutImage** procedure copies a bit-mapped image from a memory buffer to the screen at a location specified by the program.

The **_PutImage** procedure uses a *CopyMode* argument to control how the stored image interacts with what is already on the screen. The *CopyMode* argument specifies one of the following screen display operations:

Constant	Action
_Gand	Logical AND of the transfer image and screen image
_Gor	Superimposition of the transfer image onto the existing screen image
_GPReset	Direct transfer from memory to screen; color inverted
_GPSet	Direct transfer from memory to screen
_Gxor	Screen inversion only where a point exists in the transfer image

The two *CopyMode* arguments best suited for animation are **_Gxor** and **_GPSet**.

Animation done using **_GPSet** is faster, but erases the screen background. In contrast, **_Gxor** is slower, but preserves the screen background.

Animation with **_Gxor** is done with the following four steps:

1. Put the object on the screen with **_Gxor**.
2. Calculate the new position of the object.
3. Put the object on the screen a second time at the old location, using **_Gxor** again—this time to remove the old image.
4. Go to step 1, but this time put the object at the new location.

Movement done with these four steps leaves the background unchanged after step 3. Flicker can be reduced by minimizing the time between steps 4 and 1, and by making sure that there is enough time delay between steps 1 and 3. If more than one object is being animated, every object should be processed at once, one step at a time.

If it is not important to preserve the background, animation can be performed using the **_GPSet** option. If the border of the bounding rectangle around the image is as large as or larger than the maximum distance the object will move, then each time the image is put in a new location, the border will erase all traces of the image in the old location.

Using Bit-Mapped Images

The process of animating using bit-mapped images follows these eight steps:

1. Perform regular graphics initialization (select graphics mode, check for error, and so on).
2. Draw the graphics image using the **MSGraph** unit procedures and functions.
3. Use the **_ImageSize** function to determine the amount of memory required to store the image.
4. Use the **GetMem** procedure to allocate the amount of memory needed (as found in step 2).
5. Call **_GetImage** to copy the bit map of pixels from the screen to the memory buffer created in step 3.
6. Call **_PutImage** to display the image stored in memory. This display can be at any location on the screen.
7. Repeat steps 3 through 6 (possibly with different images) until finished with animation.
8. Restore the screen and exit the program.

An Example of Bit-Mapped Animation

The program ANIMATE.PAS demonstrates this process, drawing a rectangle and then redisplaying it at random locations on the screen.

```
PROGRAM Animate;
{ ANIMATE.PAS: Demonstrates animation using image buffers }

USES
    MSGraph, Crt;

CONST
    max_buffer = 65520;

VAR
    q       : Integer;
    vc      : _VideoConfig;
    buffer  : POINTER;
    imsize  : LongInt;
    x0, y0  : Integer;
    x, y    : Integer;
```

```
BEGIN  { Begin main program. }

    _ClearScreen( _GClearScreen);

    { Set the video mode and check for success }
    q := _SetVideoMode( _MaxResMode );
    IF (q = 0) THEN
        BEGIN
        Writeln( 'Graphics mode unavailable; hit Return to continue' );
        Readln;
        q := _SetVideoMode( _DefaultMode );
        Halt( 0 );
        END;

    { Find out some screen characteristics. }
    _GetVideoConfig( vc );

    { Draw and store a simple figure. }
    _SetColor( 3 );
    x := vc.NumXPixels DIV 4;
    y := vc.NumYPixels DIV 4;

    _Rectangle( _GFillInterior, 0, 0, x, y );
    imsize := _ImageSize( 0, 0, x, y );
    IF (imsize > max_buffer) THEN
        BEGIN
        Writeln( 'Image too big.' );
        Readln;
        Halt( 0 );
        END
    ELSE
        BEGIN
        GetMem( buffer, imsize );
        IF (buffer = NIL) THEN
            BEGIN
            Writeln( 'Not enough heap memory.' );
            Readln;
            Halt( 0 );
            END;
        END;

    _GetImage( 0, 0, x, y, buffer^ );
    _ClearScreen( _GClearScreen );

    { Draw axes centered on the screen }
    _SetColor (2);
    x0 := vc.NumXPixels DIV 2 -1;
    y0 := vc.NumYPixels DIV 2 -1;
    _MoveTo ( x0 ,0);
    _LineTo( x0, vc.NumYPixels );
    _MoveTo( 0, y0 );
    _LineTo( vc.NumXPixels, y0 );
```

```
_SetTextPosition(1,1);
    _OutText( '_Gxor');
    WHILE NOT KeyPressed DO
        BEGIN
        _PutImage( Random( vc.NumXPixels - x ),
                   Random( vc.NumYPixels - y ), buffer^, _Gxor );
        Delay( 500 );
        END;

    _ClearScreen( _GClearScreen );
    q := _SetVideoMode( _DefaultMode );
END.
```

Initializing Graphics and Drawing the Image The ANIMATE.PAS program begins by clearing the screen, initializing the random-number generator, and setting the video mode to the highest resolution possible.

The following code fragment draws and stores a simple image:

```
{ Draw and store a simple figure. }
_SetColor( 3 );
x := vc.NumXPixels DIV 4;
y := vc.NumYPixels DIV 4;

_Rectangle( _GFillInterior, 0, 0, x, y );
imsize := _ImageSize( 0, 0, x, y );
IF (imsize > max_buffer) THEN
    BEGIN
    Writeln( 'Image too big.' );
    Readln;
    Halt( 0 );
    END
ELSE
    BEGIN
    GetMem( buffer, imsize );
    IF (buffer = NIL) THEN
        BEGIN
        Writeln( 'Not enough heap memory.' );
        Readln;
        Halt( 0 );
        END;
    END;

_GetImage( 0, 0, x, y, buffer^ );
_ClearScreen( _GClearScreen );
```

Allocating Memory The **_Rectangle** procedure draws a rectangle that is about one-sixteenth the size of the screen. The **_ImageSize** function uses the same bounding rectangle measurements to determine the amount of memory required to hold this figure. The **GetMem** procedure then allocates the necessary memory space for the image. Because **GetMem** can allocate at most 65,520 bytes (64K–16), you must check the image size before requesting the buffer. If the image is larger than 65,520 bytes, you will need to allocate additional buffers and copy part of the image to each buffer.

The **_GetImage** procedure copies the image from the screen and stores it into the memory area specified by `buffer`. Finally, the **_ClearScreen** procedure clears this image off the screen. A more complex program could make use of the active and visual pages discussed in the previous section so that the image is drawn on the active page and the **_ClearScreen** procedure is not needed.

Next, the program draws a coordinate axis centered on the screen. This will clarify later how the stored image interacts with images already displayed on the screen.

Displaying the Image The program then repeatedly calls **_PutImage** to display the stored image at random locations on the screen. The process ends when a key is pressed. Once a key is pressed, the program ends with a call to restore the video mode.

```
    WHILE NOT KeyPressed DO
        BEGIN
        _PutImage( Random( vc.NumXPixels - x ),
             Random( vc.NumYPixels - y ), buffer^, _Gxor );
        Delay( 500 );
        END;

    _ClearScreen( _GClearScreen );
    q := _SetVideoMode( _DefaultMode );
END.
```

The **_PutImage** procedure takes four arguments. The first two specify the *x* and *y* coordinates of the upper left corner where the image is to be displayed from the memory buffer. The third argument specifies the memory buffer created by **GetMem** and used by **_GetImage** to store the bit map. Finally, the last argument specifies the interaction between the stored image and the currently displayed image. Notice that in this case (using the **_Gxor** argument) that the image is inverted when it overlaps a currently displayed figure (like the axes or another rectangle).

CHAPTER 14
Using Fonts

You can write QuickPascal programs that generate graphics and display text. In any graphics image, QuickPascal can display various styles and sizes of type. These collections of stylized text characters are called "fonts." Fonts are simple to learn and easy to use. Yet they can add a touch of polish to your program.

This chapter explains how to use fonts. It assumes you have already read Chapter 13, "Using Graphics." You should understand such terms as "graphics mode" and "text mode," and be familiar with such procedures as **_SetVideoMode** and **_MoveTo**.

Note that the QuickPascal fonts can be used only in graphics modes. Fonts cannot be used in text modes.

14.1 Overview of QuickPascal Fonts

Each font in QuickPascal consists of a typeface and several type sizes.

"Typeface" is a printer's term that refers to the style of the displayed text—Courier, for example, or Roman. The list on the following page shows six of the typefaces available with the QuickPascal font functions.

"Type size" measures the screen area occupied by individual characters. This term is also borrowed from the printer's lexicon, but for our purposes, it is specified in units of screen pixels. For example, "Courier 16 × 9" denotes text of Courier typeface, with each character occupying a screen area of 16 vertical pixels by 9 horizontal pixels.

The QuickPascal font functions use two methods to create fonts. The first technique generates the typefaces Courier, Helv, and Tms Rmn through a "bit-mapping" technique. Bit mapping defines character images with binary data. Each bit in the map corresponds to a screen pixel. If a bit is 1, its associated

pixel is set to the current screen color. A bit value of 0 clears the pixel. Video adapters use this same technique to display text in graphics mode.

The second method creates the remaining three type styles—Modern, Script, and Roman—as "vector mapped" fonts. Vector mapping represents each character in terms of lines and arcs. In a literal sense, vector-mapped characters are drawn on the screen. You might think of bit-mapped characters as being stenciled.

Each method of creating fonted text has advantages and disadvantages. Bit-mapped characters are formed more completely since the pixel mapping is predetermined. However, they cannot be scaled to arbitrary sizes. Vector-mapped text can be scaled to any size, but the characters lack the solid appearance of the bit-mapped characters.

Any function or procedure affecting the current graphics position (such as the **_MoveTo** procedure or the **_LineTo** procedure) will also affect the font display when **_OutGText** is called. Other routines (such as **_SetColor** or **_RemapPalette**) that affect drawing characteristics also affect font text output.

The QuickPascal fonts appear on your screen as follows:

Typeface	Sample Text
Courier	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
Helv	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
Tms Rmn	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
Modern	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz
Script	*ABCDEFGHIJKLMNOPQRSTUVWXYZ* *abcdefghijklmnopqrstuvwxyz*
Roman	ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz

Table 14.1 describes the characteristics of each font. Notice that bit-mapped fonts come in preset sizes as measured in pixels. The exact size of any font character depends on the screen resolution and display type.

Table 14.1 Typefaces and Type Sizes in QuickPascal

Typeface	Mapping	Size (in Pixels)	Spacing
Courier	Bit	13 × 8, 16 × 9, 20 × 12	Fixed
Helv	Bit	13 × 5, 16 × 7, 20 × 8 13 × 15, 16 × 6, 19 × 8	Proportional
Tms Rmn	Bit	10 × 5, 12 × 6, 15 × 8 16 × 9, 20 × 12, 26 × 16	Proportional
Modern	Vector	Scaled	Proportional
Script	Vector	Scaled	Proportional
Roman	Vector	Scaled	Proportional

14.2 Using Fonts in QuickPascal

Data for both bit-mapped and vector-mapped fonts reside in files on disk. A .FON extension identifies the files. The names of the .FON files indicate their content. For example, the MODERN.FON, ROMAN.FON, and SCRIPT.FON files hold data for the three vector-mapped fonts.

The QuickPascal .FON files are identical to the font files supplied with Microsoft QuickC®, Version 2.0, as well as being identical to the .FON files used in the Microsoft Windows operating environment. Consequently, you can use any of the Windows .FON files with the QuickPascal font functions. The Windows .FON files are also available for purchase separately. In addition, several vendors offer software that create or modify .FON files, allowing you to design your own fonts.

Your programs should follow these four steps to display fonted text:

1. Set a graphics video mode
2. Register fonts
3. Set the current font from the register
4. Display text using the current font

Sections 14.2.1–14.2.3 describe each of the font-specific steps in detail. The procedure for using video modes for graphics is discussed in Section 13.3. An example program in the final section of this chapter demonstrates how to display the various fonts available in the QuickPascal .FON files.

14.2.1 Registering Fonts

The fonts you plan to use must be organized into a list in memory, a process called "registering." The register list contains information about the available .FON files. You register fonts by calling the function **_RegisterFonts**. This function reads header information from the specified .FON files. It builds a list of file information but does not read mapping data from the files.

The **MSGraph** unit defines the **_RegisterFonts** function as

Function **_RegisterFonts(** *PathName* **: CSTRING) :** *Integer*

The argument *PathName* is a string containing a file name. The file name is the name of the .FON file for the desired font. The file name can include wild cards, allowing you to register several fonts with one call to **_RegisterFonts**. For example, the function call below registers all of the .FON files in the current directory and checks for a successful registration:

```
result := _RegisterFonts( '*.FON' );
IF (result < 0) THEN
    BEGIN
    Writeln( 'Unable to register fonts');
    Halt( 0 );
    END;

{ the rest of your fonts program goes here }
```

As illustrated above, the **_RegisterFonts** function returns a negative number if it is unable to register any fonts or if the .FON file is corrupt. If it successfully reads one or more .FON files, **_RegisterFonts** returns the number of fonts registered.

14.2.2 Setting the Current Font

To set a font as the current font, call the function **_SetFont**. This function checks to see if the requested font is registered, then reads the mapping data from the appropriate .FON file. A font must be registered and marked current before your program can display text using that font.

The **MSGraph** unit defines the **_SetFont** function as

Function **_SetFont(** *Options* **: CSTRING) :** *Integer*

The *Options* argument is a string that describes the desired characteristics of the font. The string uses letter codes that describe the desired font, as outlined here:

Option Code	Meaning
t'*FontName*'	Typeface of the font in single quotes. The *FontName* string is one of the following: courier — modern helv — script tms rmn — roman Notice that the *FontName* string is surrounded by a pair of two single quotes. This is necessary to embed the *FontName* string within the *Options* string, and Pascal uses single quotes to specify a string. Notice the space in "tms rmn." Other products' font files use other names for *FontName*. Refer to the vendor's documentation for these names.
h*y*	Character height, where *y* is the height in pixels.
w*x*	Character width, where *x* is the width in pixels.
f	Select only a fixed-spaced font.
p	Select only a proportional-spaced font.
v	Select only a vector-mapped font.
r	Select only a bit-mapped font.
b	Select the best fit from the registered fonts. This option instructs **_SetFont** to accept the closest-fitting font if a font of the specified size is not registered. If at least one font is registered, the **b** option always guarantees that **_SetFont** will be able to set a current font. If you do not specify the **b** option and an exact matching font is not registered, **_SetFont** will fail. In this case, any existing current font remains current. The **_SetFont** function uses four criteria for selecting the best fit. In descending order of precedence the four criteria are pixel height, typeface, pixel width, and spacing (fixed or proportional). If you request a vector-mapped font, **_SetFont** sizes the font to correspond with the specified pixel height and width. If you request a bit-mapped font, **_SetFont** chooses the closest available size. If the requested type size for a bit-mapped font fits exactly between two registered fonts, the smaller size takes precedence.

n*x* — Select font number *x*, where *x* is less than or equal to the value returned by **_RegisterFonts**. For example, the option **n3** makes the third registered font current, assuming that three or more fonts are registered.

This option is primarily useful for cycling through all registered fonts in a loop. Because .FON files often contain several fonts, and the files are loaded into memory in reverse order from which they are registered, it is difficult to know which font will be number 3.

Option codes are not case-sensitive and can be listed in any order. You can separate codes with spaces or any other character that is not a valid option code. The **_SetFont** function ignores all invalid codes.

For example, the function call below specifies that the font should be a "script" typeface with a character height of 30 pixels and a character width of 24 pixels. If the function is unable to do this, a "best fit" font is requested. The multiple single quotes around `script` are required since the entire argument used by **_SetFont** is a string. The double single quote specifies to QuickPascal that the string contains a single quote.

```
result := _SetFont( 't''script''h30w24b' );
IF (result =  -1) THEN
    BEGIN
    Writeln( 'Unable to set requested font');
    Halt( 0 );
    END;

Writeln( 'Font set' );
{ the rest of your font program goes here }
```

As illustrated above, the **_SetFont** function returns a `-1` if it is unable to set the requested font. If it successfully sets a current font, the value `0` is returned.

Once a font is set as the current font, the **_SetFont** function updates a data area with the parameters of the current font. The data area is in the form of a **_FontInfo** record, defined in the **MSGraph** unit as

```
{ structure for GetFontInfo }

    _FontInfo = RECORD
        fonttype  : Integer;      { b0 set = vector,clear = bit map     }
        ascent    : Integer;      { pix dist from top to baseline       }
        pixwidth  : Integer;      { character width in pixels, 0 = prop }
        pixheight : Integer;      { character height in pixels          }
        avgwidth  : Integer;      { average character width in pixels   }
        filename  : CSTRING[81];  { file name including path            }
        facename  : CSTRING[32];  { font name                           }
        END;
```

If you wish to retrieve the parameters of the current font, call the function **_GetFontInfo**, which is defined in the **MSGraph** unit as

Function **_GetFontInfo(VAR** *FInfo* **: _FontInfo) :** *Integer*

14.2.3 Displaying Text Using the Current Font

Now you can display the font-based text. This step consists of two parts:

1. Select a screen position for the text with the graphics procedure **_MoveTo**. Note that all of the font-based text is displayed using graphics functions. Consequently, the **_MoveTo** procedure (rather than the text procedure **_SetTextPosition**) positions the text. The **_MoveTo** procedure takes pixel coordinates as arguments. The coordinates specify the upper left point of the first character in the text string. Optionally, you can use the procedure **_SetGTextVector** to change the orientation of the text on the screen.
2. Display the font-based text at that position with the procedure **_OutGText**.

14.3 A Few Hints on Using Fonts

Fonted text is simply another form of graphics, and using fonts effectively requires little programming effort. Still, there are a few things to watch:

- Remember that the video mode should be set only once to establish a graphics mode. If you generate an image (as with the **_Rectangle** procedure) and wish to incorporate fonted text above it as a title, don't reset the video mode prior to calling the font routines. Doing so will blank the screen, destroying the original image.
- The **_SetFont** function reads specified .FON files to obtain mapping data for the current font. Each call to **_SetFont** causes a disk access and overwrites the old font data in memory. If you wish to show text of different styles on the same screen, display all of the text of one font before moving on to the others. By minimizing the number of calls to **_SetFont**, you'll save time spent in disk I/O and memory reloads.
- When your program finishes with the fonts, you might want to free the memory occupied by the register list. Call the **_UnRegisterFonts** procedure to do this. As its name implies, this procedure frees the memory previously allocated by **_RegisterFonts**. The register information for each type size of each font takes up approximately 140 bytes of memory. Thus the amount of memory returned by **_UnRegisterFonts** is significant only if you have many fonts registered.

- As for screen aesthetics, the same suggestions for the printed page apply to fonted screen text. Typefaces are more effective when they are not competing with each other for attention. Restricting the number of styles per screen to one or two generally results in a more pleasing, less cluttered image.

14.4 Example Program

The QuickPascal font functions shine when they are used in conjunction with your other graphics functions. They allow you to dress up any image on the screen, yet they can make a visual impression when used by themselves, as the example program SAMPLER.PAS illustrates. This program displays sample text in all of the available fonts, then exits when a key is pressed. Make sure the .FON files are in the current directory before running the program.

Notice that SAMPLER.PAS calls the graphics procedure **_MoveTo** to establish the starting position for each text string. Section 13.4 "Understanding Coordinate Systems," describes the **_MoveTo** procedure. The function **_SetFont** takes a character string as an argument. The string is an options list that specifies typeface and the best fit for a character height of 30 pixels and a width of 24 pixels.

```
PROGRAM sampler;  { Demonstrates using different fonts }

USES
        Crt, MSGraph;

CONST
        CRLF = #13 + #10;
        nfonts = 6;

        texttypes : ARRAY[ 1..2, 1..nfonts ] OF CSTRING[8] =
        (
        ( 'roman', 'courier', 'helv', 'tms rmn', 'modern', 'script' ),
        ( 'ROMAN', 'COURIER', 'HELV', 'TMS RMN', 'MODERN', 'SCRIPT' )
           );

        faces : ARRAY[ 1..nfonts ] OF CSTRING[12] =
                (
                 ' t''roman''',
                 ' t''cour''',
                 ' t''helv''',
                 ' t''tms rmn''',
                 ' t''modern''',
                 ' t''script'''
                );
        fontpath : CSTRING = '*.FON';
VAR
    list : CSTRING;
    vc   : _VideoConfig;
    i, a : Integer;
    stra : STRING[3];
    ch   : Char;
```

```
BEGIN  { Begin main program }

       { Read header information from all .FON files in
         the current directory
       }
       a := _RegisterFonts( fontpath );
       IF  a < 0 THEN
           BEGIN
           _OutText('Error: Cannot register fonts.' + CRLF);
           Halt(1);
           END;

       { Set the highest available video mode }
       a := _SetVideoMode( _MaxResMode );
       Str( a, stra );
       _OutText( 'MaxresMode = ' + stra );

       { Copy video configuration into structure vc }
       _GetVideoConfig(vc);

       { Display six lines of sample text }
       FOR i := 1 TO nfonts DO
               BEGIN
               list := faces[i] + 'bh24w24' ;
               a:= _SetFont( list );

               IF ( a <> -1) THEN
                       BEGIN
                       _SetColor( i + 1);
                       _MoveTo( 0, i * 30 );
                       _OutGText( texttypes[2,i] );
                       _MoveTo( vc.NumXPixels DIV 2, i * 30 );
                       _OutGText( texttypes[1,i]+CRLF );
                       END
               ELSE
                       BEGIN
                       a := _SetVideoMode(_DefaultMode);
                       _OutText('Error: Cannot set font.');
                       Halt(1);
                       END;
           END;

       ch := ReadKey;

       a := _SetVideoMode(_DefaultMode);

       _UnRegisterFonts;  { Returns memory used by fonts }

END.
```

CHAPTER 15

15 Object-Oriented Programming

Object-oriented programming is widely hailed as the programming style of the future. QuickPascal offers you object-oriented programming today, through its object extensions to standard Pascal. Although they make only a few syntactic additions to the language, the QuickPascal object extensions provide a powerful and efficient framework for creating programs.

15.1 Overview

Standard Pascal programs, along with programs written in other procedural languages, are organized around a set of data structures, with separate procedures and functions manipulating the data. An example is a graphics program that declares each shape as a unique **TYPE**. Various routines draw, erase, and move the shapes, likely using a **CASE** statement to differentiate between them.

Object-oriented programs operate differently. Instead of being organized around data, they are organized around a set of "objects." An object is a structure that combines both data and routines into one type. It is similar to a Pascal **RECORD** type, but can store both functions and procedures as well as data.

Objects have a property called "inheritance." Once an object has been declared, another object can be derived that inherits all of the data and routines associated with the parent type. New data and routines can be added, or existing inherited routines modified.

A graphics program that was written with object extensions to QuickPascal would declare an initial "generic shape" object. The generic shape would define all of the data and routines—such as draw, erase, and size—that were common to every shape. New shapes would be derived from the generic shape, and then these new shapes would declare additional data fields, override existing routines, and add new ones.

One of the primary benefits of object-oriented programming is the ease with which programs can be changed and portions reused. In the hypothetical standard Pascal graphics application, to add an octagon shape to the program, you would need to declare an entire new type as well as modify each routine that dealt with the shapes. With object extensions to QuickPascal, you would define an octagon object, already derived from the generic shape object, and add or modify any data or routines the octagon exclusively used. The old routines would still operate the same way on old types of objects. Instead of making changes throughout the entire program, all of the changes would occur in one localized area and apply only to that object or its descendants.

The example in Section 15.5 demonstrates basic object-oriented programming techniques.

15.2 Object Programming Concepts

Object-oriented extensions are based on four concepts: classes, objects, methods, and inheritance. A "class" is similar to a Pascal **RECORD**. It describes an overall structure for any number of types based upon it. The main difference between a class and a record is that a class combines data fields (called "instance variables") and procedures and functions (called "methods") that act upon the data. Instance variables can include standard Pascal data types as well as objects.

An "object" is a variable of a class (often called a class instance). Like a class, an object is declared as a **TYPE**. All objects derived from a class are considered members of that class and share similar characteristics of the superclass.

"Methods" are procedures and functions encapsulated in a class or object. Calling a method is referred to as "passing a message to an object." The object extensions to QuickPascal create programs that do most of their work by sending messages to objects, and by instructing objects to send messages to each other. Methods are stored in an object-type method table and do not occupy memory when an object is declared as a variable.

Members of the same class exhibit similar behavior through inheritance. This means the variable instances and methods found in a superclass are also present in objects derived from the superclass. Additionally, objects have their own space for storing data and methods local to the object. If necessary, an object can also override a parent class's method, replacing the inherited method's instructions with its own. If it does, only the descendant object's methods are altered, while the parent's remain unchanged.

15.3 Using Objects

As mentioned before, the object extensions to QuickPascal add only a few new keywords and types. All of the standard Pascal identifiers, constructs, and routines are available when programming with objects. The differences in using

object extensions are in the areas of declaring class and object data structures and of calling procedures and functions through methods.

15.3.1 Setting the Method Compiler Directive

The first step in using object extensions is to enable the Method compiler directive. The {$M+} directive should appear at the beginning of any source file that uses objects. (The {$M+} directive is enabled by default.) This directive instructs the compiler to check whether or not memory for an object has been allocated before the object's method is executed. See Appendix B, "Compiler Directives," for more information.

15.3.2 Creating Classes

Since all objects are derived from classes, classes are created first. A class should incorporate all data and methods that descendant objects will have in common.

You use the following syntax to declare an object class:

```
TYPE
  ClassName = OBJECT
    DataFields
    {PROCEDURE | FUNCTION}[[Methods]]
  END;
```

The parts of the syntax are defined below:

Argument	Discription
ClassName	A unique name that identifies the class.
OBJECT	A QuickPascal keyword that instructs the compiler to treat the structure as an object.
DataFields	The declaration of one or more data structures. The syntax is the same as that used for declaring the fields of a record.
Methods	A list of method declarations. Each method declaration is like a procedure or function heading, except that the name may be qualified with the name of the class: *ClassName.MethodName*. Although not required, such a qualification is good programming style. Methods are declared immediately following the class and object type declarations.

For example, the following code fragment creates a generic shape for a graphics program:

```
TYPE
     shape = OBJECT
             color: colors;
             height, width: Integer;
             PROCEDURE shape.init;
             PROCEDURE shape.draw;
             PROCEDURE shape.move(hoz, vert: Integer);
             FUNCTION shape.area: Integer;
     END;

     PROCEDURE shape.init
         BEGIN { code for init method here }
         .
         .
         .
         END;

     PROCEDURE shape.draw;
         BEGIN  { code for draw method here }
         .      { remainder of methods }
         .
         .
         END;
```

15.3.3 Creating Subclasses

Once a class has been created, subclasses can be defined. The syntax for creating a subclass is similar to that of a class:

TYPE
ObjectName = **OBJECT(** *ParentClass* **)**
DataFields
{PROCEDURE | FUNCTION}[[*Methods***]] [[; OVERRIDE]]**
END;

The two special aspects of declaring objects are the use of parent class and of overriding inherited methods. The argument *ParentClass* is the name of a parent class. Since the subclass is derived from a class, you would enclose the name of the class in parentheses.

If the subclass redefines a method from the parent class, the **OVERRIDE** statement should appear after the method header.

For example, the following code fragment declares a descendant of the `shape` class:

```
TYPE
     circle = OBJECT(shape)
         radius: Integer;
         PROCEDURE circle.init; OVERRIDE;
         PROCEDURE circle.draw; OVERRIDE;
         FUNCTION circle.area: Integer; OVERRIDE;
         PROCEDURE circle.changeradius(new_radius: Integer);
     END;
```

Because the `circle` type is being derived from the `shape` class, there is no need to declare all of the instance variables and methods from `shape`. The only variables and methods that need declaring are those that are new and exclusive to the `circle` object. In this case, the new items are the `radius` field and the `changeradius` method. A `circle` object will have color, height, width, and radius fields.

Since the `init`, `draw`, and `area` methods will be different for `circle` than they were for `shape`, the **OVERRIDE** keyword instructs the compiler to use the method local to `circle` when one of these messages is passed to the object.

15.3.4 Defining Methods

After a method has been associated with an object, it must be defined. Methods are defined with the **PROCEDURE** or **FUNCTION** keywords. The actual statements that compose the method are defined after all classes and subclasses have been created. Either the **PROCEDURE** or **FUNCTION** keyword precedes the object name, followed by a period (.) and the method name. Methods that are overridden follow the same syntax. (See the example at the end of the chapter.)

The first method you should define is one that initializes all of the object's data fields, allocates memory, or performs any other actions the object may need before being used. This method should be called immediately after space has been allocated for the object.

Instance variables that belong to the object can be accessed from within a method by using their identifier preceded by the pseudovariable **Self**, as shown below:

```
PROCEDURE circle.init;
    BEGIN
        Self.color  := blue;
        Self.height := 20;
        Self.width  := 20;
        Self.radius := 0;
    END;
```

Self simply instructs the object to operate on itself.

An object's data may be accessed by a program directly, as if the object were a record:

```
the_radius := circle.radius;
```

Also, to call a method belonging to the object from within a different method, you may precede it with the **Self** variable. In the code fragment below, `Self.draw` is equivalent to `circle.draw`.

```
PROCEDURE circle.move(hoz, vert: Integer);
    BEGIN
    .
    .
    .
        Self.draw;
    END;
```

Note that you are not restricted solely to using methods when you use object extensions to QuickPascal. Methods are only used with objects. Standard Pascal procedures and functions can be implemented to manipulate other forms of data.

15.3.5 Using INHERITED

The **INHERITED** keyword negates an override of an inherited method. If the class of method performs only a portion of what an object needs to have done, the parent method can be called from the descendant method.

For example, suppose that in the `shape` initialization method, you set the following values:

```
PROCEDURE shape.init;
    BEGIN
        color  := blue;
        height := 20;
        width  := 20;
    END;
```

If the `circle` object used these values, but overrode the method to initialize its own data field, **INHERITED** could be used to call the ancestor method. This would initialize the common fields without needing to initialize them in the descendant method.

```
PROCEDURE circle.init;
    BEGIN
        radius := 0;
        INHERITED Self.init;
    END;
```

15.3.6 Declaring Objects

Declaring an object is similar to declaring a dynamic variable. The syntax is

VAR
ObjectIdentifier : *Class*

ObjectIdentifier is the QuickPascal identifier for the object, and *Class* is the type of the object.

For example, this code declares an object of the class `circle`:

```
VAR
    my_circle: circle;
```

15.3.7 Allocating Memory

Before an object can be used in a program, memory space must be allocated for it. This is done with the Pascal **New** procedure:

```
New( my_circle );
```

A common mistake in object-oriented programming is forgetting to allocate memory for an object and then trying to access it. Allocating memory for objects should be one of the first actions of the program body.

15.3.8 Calling Methods

After classes and objects have been declared, and memory has been allocated for the object, you can call a method (that is, send a message to the object) from within the main body of the program to manipulate the object's instance variables. Sending a message is similar to calling a procedure or function in standard Pascal. The only difference is that you specify both the object and the method.

For example, different methods for `my_circle` are executed by

```
my_circle.draw;
my_circle.move( 30, 30 );
circle_area := my_circle.area;
```

15.3.9 Testing Membership

The **Member** function tests if a particular object is in a class, as shown below:

```
IF Member( a_circle, shape ) THEN
   num_shapes := num_shapes + 1;
```

The function is passed the object and the class. It returns **True** if the object is an instance of, or a decendant of, the class.

15.3.10 Disposing of Objects

When you are finished using an object, the memory allocated for it should be freed. This is done with the **Dispose** procedure:

```
Dispose( my_circle );
```

Before disposing of an object, be sure it will not be used further in the program.

Often, a free method is declared to reallocate data-structure memory, close files, and perform other housecleaning. Such a method should be called before using **Dispose**.

15.4 Object Programming Strategies

The greatest difficulty facing programmers who are learning object extensions to QuickPascal is the need to plan and write their programs in an object-oriented manner. All too often, a programmer's first object-oriented programs exhibit a procedural style with objects sprinkled in haphazardly. Programming in this fashion reduces the value of object extensions for producing efficient and reusable code. Sections 15.4.1–15.4.5 discuss several issues you should keep in mind as you create object-oriented programs.

15.4.1 Object Style Conventions

Although style conventions for programs are often a matter of personal preference, adoption of certain style conventions for object programming can make your source code easier to read. For example, since both a Pascal record and an object use a period (.) to access their data fields and methods, it can be difficult to distinguish objects from records. This is made more complicated by the difficulty telling whether an identifier following an object is an instance variable or a method.

Here are some style conventions for object programming:

- Classes and objects are preceded with an uppercase "T." This identifies the variable as an object type, as shown below:

  ```
  Tcircle = OBJECT(Tshape)
  ```

- Object instance variables are preceded with a lowercase "f." The "f" indicates at a glance that the identifier is a field. Identifiers without the "f" are methods, as in the following example:

  ```
  the_radius := Tcircle.fradius;
  Tcircle.draw;
  ```

- Global variables are preceded with a lowercase "g." This is helpful in identifying objects that can be passed messages from objects outside their own class (see below):

```
gTtemp_circle.color := blue;
```

15.4.2 Object Reusability

The essence of object-oriented programming is reusability. When you create objects, you should give some thought to their future use, both in terms of the current program and for later ones. It's best to create classes from which other objects can be derived. Inheriting methods is generally more important than inheriting data. On a larger scale, class libraries are useful for dealing with common tasks. A set of related classes can reside in a UNIT and be called at any time.

15.4.3 Modularity

Object extensions to QuickPascal lend themselves to modularized programs. A class's methods can easily be kept together, instead of being strung out through the source code. A properly constructed object program should require only a few localized modifications to add and alter methods.

15.4.4 Methods

Methods should be treated as replaceable components of the object building blocks of QuickPascal. Methods are designed to serve a single purpose; a method that is multipurpose is more difficult to modify because it performs a variety of tasks. Whenever you want to perform an action, create a method to do it. Methods should be short, at most several dozen statements in length.

15.4.5 Data Fields

It isn't necessary to declare an instance variable for each data item an object may use. If more than one object method uses a specific data item, the data should be incorporated as an instance variable. If only a single method accesses the data, it can be passed as a parameter to the method. You should use object instance variables in place of global variables to promote modularity.

15.5 Example Program

This example shows how a typical object-oriented program works. A class is declared (`geo_shape`), with two subclasses (`rectangle` and `circle`). Both the subclasses demonstrate how to add instance variables and methods and how to override existing parent methods. The body of the OBJECTDE.PAS program has examples of defining methods, accessing instance variables, and declaring and disposing of objects.

```
Program objectdemo;
{ Demonstrates object techniques with geometric shapes }

{M+}

TYPE

     geo_shape = OBJECT
                 area: Real;
                 height: Real;
                 what: STRING;
                 PROCEDURE geo_shape.init;
                 PROCEDURE geo_shape.say_what;
                 FUNCTION geo_shape.get_area : Real;
     END;

     rectangle = OBJECT(geo_shape)
                 len: Real;
                 FUNCTION rectangle.is_square : Boolean;
                 PROCEDURE rectangle.init; OVERRIDE;
                 FUNCTION rectangle.get_area : Real; OVERRIDE;
     END;

     circle = OBJECT(geo_shape)
              radius: Real;
              PROCEDURE circle.init; OVERRIDE;
              FUNCTION circle.get_area : Real; OVERRIDE;
     END;

PROCEDURE geo_shape.init;
BEGIN
    Self.area := 0;
    Self.height := 0;
    Self.what := 'Geometric shape';
END;

PROCEDURE geo_shape.say_what;
BEGIN
    Writeln(Self.what);
END;

FUNCTION geo_shape.get_area : Real;
BEGIN
    Self.area := Self.height * Self.height;
    get_area := Self.height;
END;
```

```
PROCEDURE circle.init;
BEGIN
    INHERITED Self.init;
    Self.radius := 4;
    Self.what := 'circle';
END;

FUNCTION circle.get_area: Real;
BEGIN
    Self.area := ( Pi * Sqr( Self.radius ) );
    get_area := Self.area;
END;

PROCEDURE rectangle.init;
BEGIN
    INHERITED Self.init;
    Self.height := 5;
    Self.len := 5;
    Self.what := 'Rectangle';
END;

FUNCTION rectangle.is_square: Boolean;
BEGIN
    is_square := False;
    IF Self.len = Self.height THEN
    is_square := True;
END;

FUNCTION rectangle.get_area: Real;
BEGIN
    Self.area := ( Self.len * Self.height );
    get_area := Self.area;
END;

VAR

    the_circle : circle;
    the_rect : rectangle;

BEGIN
    New( the_circle );
    the_circle.init;
    New( the_rect );
    the_rect.init;

    the_circle.say_what;
    Writeln( 'area: ', the_circle.get_area );

    Writeln;

    the_rect.say_what;
    Writeln( 'area: ', the_rect.get_area );

    Dispose( the_circle );
    Dispose( the_rect );
END.
```

Appendixes

Appendix A

ASCII Character Codes and Extended Key Codes

A.1 ASCII Character Codes

The ASCII character codes for printable and control characters are listed on the following two pages. The value of each character is its ordinal value within the type **Char**. For example, Ord ('P') returns the decimal value 80.

Ctrl	Dec	Hex	Char	Code
^@	0	00		NUL
^A	1	01	☺	SOH
^B	2	02	☻	STX
^C	3	03	♥	ETX
^D	4	04	♦	EOT
^E	5	05	♣	ENQ
^F	6	06	♠	ACK
^G	7	07	•	BEL
^H	8	08	◘	BS
^I	9	09	○	HT
^J	10	0A	◙	LF
^K	11	0B	♂	VT
^L	12	0C	♀	FF
^M	13	0D	♪	CR
^N	14	0E	♫	SO
^O	15	0F	☼	SI
^P	16	10	►	DLE
^Q	17	11	◄	DC1
^R	18	12	↕	DC2
^S	19	13	‼	DC3
^T	20	14	¶	DC4
^U	21	15	§	NAK
^V	22	16	▬	SYN
^W	23	17	↨	ETB
^X	24	18	↑	CAN
^Y	25	19	↓	EM
^Z	26	1A	→	SUB
^[	27	1B	←	ESC
^\	28	1C	∟	FS
^]	29	1D	↔	GS
^^	30	1E	▲	RS
^_	31	1F	▼	US

Dec	Hex	Char
32	20	
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	⌂†

† ASCII code 127 has the code DEL. Under DOS, this code has the same effect as ASCII 8 (BS). The DEL code can be generated by the CTRL + BKSP key combination.

Dec	Hex	Char
128	80	Ç
129	81	ü
130	82	é
131	83	â
132	84	ä
133	85	à
134	86	å
135	87	ç
136	88	ê
137	89	ë
138	8A	è
139	8B	ï
140	8C	î
141	8D	ì
142	8E	Ä
143	8F	Å
144	90	É
145	91	æ
146	92	Æ
147	93	ô
148	94	ö
149	95	ò
150	96	û
151	97	ù
152	98	ÿ
153	99	Ö
154	9A	Ü
155	9B	¢
156	9C	£
157	9D	¥
158	9E	₧
159	9F	ƒ

Dec	Hex	Char
160	A0	á
161	A1	í
162	A2	ó
163	A3	ú
164	A4	ñ
165	A5	Ñ
166	A6	ª
167	A7	º
168	A8	¿
169	A9	⌐
170	AA	¬
171	AB	½
172	AC	¼
173	AD	¡
174	AE	«
175	AF	»
176	B0	░
177	B1	▒
178	B2	▓
179	B3	│
180	B4	┤
181	B5	╡
182	B6	╢
183	B7	╖
184	B8	╕
185	B9	╣
186	BA	║
187	BB	╗
188	BC	╝
189	BD	╜
190	BE	╛
191	BF	┐

Dec	Hex	Char
192	C0	└
193	C1	┴
194	C2	┬
195	C3	├
196	C4	─
197	C5	┼
198	C6	╞
199	C7	╟
200	C8	╚
201	C9	╔
202	CA	╩
203	CB	╦
204	CC	╠
205	CD	═
206	CE	╬
207	CF	╧
208	D0	╨
209	D1	╤
210	D2	╥
211	D3	╙
212	D4	╘
213	D5	╒
214	D6	╓
215	D7	╫
216	D8	╪
217	D9	┘
218	DA	┌
219	DB	█
220	DC	▄
221	DD	▌
222	DE	▐
223	DF	▀

Dec	Hex	Char
224	E0	α
225	E1	ß
226	E2	Γ
227	E3	π
228	E4	Σ
229	E5	σ
230	E6	µ
231	E7	τ
232	E8	Φ
233	E9	Θ
234	EA	Ω
235	EB	δ
236	EC	∞
237	ED	φ
238	EE	ε
239	EF	∩
240	F0	≡
241	F1	±
242	F2	≥
243	F3	≤
244	F4	⌠
245	F5	⌡
246	F6	÷
247	F7	≈
248	F8	°
249	F9	∙
250	FA	·
251	FB	√
252	FC	ⁿ
253	FD	²
254	FE	■
255	FF	

A.2 Extended-Key Codes

The **Crt** unit's **ReadKey** function returns a pair of values (instead of a single ASCII character) when a key on an extended keyboard is pressed. The first code is a null character (ASCII 0), which indicates that the next character will be an extended-key code.

This table lists the key or key combination along with the decimal extended-key code. This information is also available in the on-line help (in a form that can be pasted directly into your program).

Key Type	Extended Key	Code	Extended Key	Code
Arrow	UP	72	DOWN	80
	LEFT	75	RIGHT	77
	PGUP	73	PGDN	81
	HOME	71	END	79
	INS	82	DEL	83
CTRL +	CTRL+PRTSC	114	CTRL+HOME	119
Arrow	CTRL+LEFT	115	CTRL+END	117
	CTRL+RIGHT	116		
	CTRL+PGUP	132	Null key	3
	CTRL+PGDN	118	SHIFT+TAB	15
ALT +	ALT+A	30	ALT+N	49
Letter	ALT+B	48	ALT+O	24
	ALT+C	46	ALT+P	25
	ALT+D	32	ALT+Q	16
	ALT+E	18	ALT+R	19
	ALT+F	33	ALT+S	31
	ALT+G	34	ALT+T	20
	ALT+H	35	ALT+U	22
	ALT+I	23	ALT+V	47
	ALT+J	36	ALT+W	17
	ALT+K	37	ALT+X	45
	ALT+L	38	ALT+Y	21
	ALT+M	50	ALT+Z	44

Key Type	Extended Key	Code	Extended Key	Code
ALT +	ALT+1	120	ALT+6	125
Number	ALT+2	121	ALT+7	126
	ALT+3	122	ALT+8	127
	ALT+4	123	ALT+9	128
	ALT+5	124	ALT+0	129
	ALT+MINUS	130	ALT+ =	131
Function	F1	59	F6	64
	F2	60	F7	65
	F3	61	F8	66
	F4	62	F9	67
	F5	63	F10	68
SHIFT +	SHIFT+F1	84	SHIFT+F6	89
Function	SHIFT+F2	85	SHIFT+F7	90
	SHIFT+F3	86	SHIFT+F8	91
	SHIFT+F4	87	SHIFT+F9	92
	SHIFT+F5	88	SHIFT+F10	93
CTRL +	CTRL+F1	94	CTRL+F6	99
Function	CTRL+F2	95	CTRL+F7	100
	CTRL+F3	96	CTRL+F8	101
	CTRL+F4	97	CTRL+F9	102
	CTRL+F5	98	CTRL+F10	103
ALT +	ALT+F1	104	ALT+F6	109
Function	ALT+F2	105	ALT+F7	110
	ALT+F3	106	ALT+F8	111
	ALT+F4	107	ALT+F9	112
	ALT+F5	108	ALT+F10	113
Extended	F11	133	F12	134
Function	SHIFT+F11	135	SHIFT+F12	136
	CTRL+F11	137	CTRL+F12	138
	ALT+F11	139	ALT+F12	140

Appendix B

Compiler Directives

QuickPascal uses a variety of compiler directives to affect code output. They can be classed as

- Switch Directives (on and off switches that control compiler options)
- Parameter Directives (commands that require numbers or a file name)
- Conditional Directives (commands that compile specific portions of the source file, depending if certain conditions are met)

Compiler directives can be specified in the source file, in the QuickPascal environment, or at the DOS command line.

B.1 Switch Directives

Switch directives are identified by a single letter enclosed in braces. The letter is preceded by a dollar sign ($) and is followed by either a plus (+) that enables the switch, or a minus (–) that disables it. If you do not specify a switch directive, the default setting will automatically be used.

A space must not come between the left brace and the dollar sign. If one does, the compiler considers the line to be a comment. This is useful for commenting out directives.

Multiple switches can be set in a single statement. For example,

```
{$A-,S+,R-,V+}
```

Certain switch directives must be declared globally at the beginning of the program, just after the header. That is, they must appear before any data declaration or code. Other switch directives can occur locally, anywhere within the source file.

The individual switch directives are described below.

Align Data

{$A+}** or **{$A–}
Default: **{$A+}**
Type: Local

Changes from byte alignment to word alignment of typed constants and variables. When the align data directive is enabled, all typed constants and variables larger than one byte are aligned on a machine-word boundary. Although word

alignment does not affect program execution on the 8088 processor, on 80x86 processors, programs can execute faster since word-sized items are accessed in one memory cycle.

When the directive is disabled, no alignment occurs, and all typed constants and variables are placed at the next available memory address.

Boolean Evaluation

{$B+} or {$B–}
Default: {$B–}
Type: Local

Determines which type of code will be generated for the Boolean **AND** and **OR** operators. When the directive is enabled, code is created for the complete Boolean expression, even if the result is already known. If the directive is disabled (the default), the evaluation stops as soon as the result becomes evident, resulting in faster programs. Enable this switch when the second or subsequent operands in the Boolean expression include a procedure or function call that needs to be executed, regardless of the value of the first operand.

Debug Information

{$D+} or {$D–}
Default: {$D+}
Type: Global

Inserts debugging information into your compiled program. When the directive is enabled, you can use the QuickPascal debugger to single step through code to view VARS, you need {$L+}. Additionally, when a run-time error is reported, QuickPascal will move to the line in the source code that caused the error.

FAR Calls

{$F+} or {$F–}
Default: {$F–}
Type: Local

Determines which model is used for compiled procedures and functions. When the FAR calls directive is enabled, procedures and functions always use the FAR call model. If the directive is disabled, QuickPascal determines the call model based on location of the routine. If the procedure or function is declared in the **INTERFACE** portion of a unit, the FAR model is used. Otherwise, the NEAR model is used.

See Chapter 12, "Advanced Topics," for a full discussion of FAR and NEAR call models.

I/O Checking

{$I+} or {$I–}
Default: {$I+}
Type: Local

Generates code that examines the result of an input/output procedure. If an I/O error occurs while the directive is enabled, the program displays a run-time error and terminates. If the I/O checking directive is disabled, you should use the **IOResult** function to check for I/O errors. See Chapter 9, "Text Files," for more information on I/O routines.

Local Symbol Data

{$L+} or {$L–}
Default: {$L+}
Type: Global

Generates symbol information in your compiled code. When the directive is enabled, you can use the QuickPascal debugger to examine and modify variables within your program. The local symbol data directive is used in conjunction with the debug information directive. If the debug information directive is disabled, the local symbol data directive is automatically disabled.

Method Checking

{$M+} or {$M–}
Default: {$M–}
Type: Global

Determines if memory for an object has been allocated before an object's method is called. If the directive is enabled and an object hasn't had memory allocated, a run-time error is generated. You should always enable the method-checking directive when you are using objects and programming in object-oriented Pascal.

Numeric Processing

{$N+} or {$N–}
Default: {$N–}
Type: Global

Specifies which model of floating-point code is generated by the compiler. When the directive is enabled, all floating-point calculations are performed by the 80x87 math coprocessor. If the directive is disabled, all real type calculations are performed in software using the run-time libraries.

When the directive is enabled ($N+), the resulting program will run only on machines equipped with 80x87 coprocessors. Attempting to run the program on a computer that is not 80x87-equipped generates an error message informing the user that a coprocessor is required.

When the directive is disabled ($N–), the resulting program will run on any machine, whether or not it has a coprocessor. However, the code will not use the coprocessor, even if present. Use of the $N– directive involves some differences in the way **Extended** and **Comp** data types are handled. See Chapter 2, "Programming Basics," for more information.

Range Checking

{$R+} or {$R–}
Default: {$R–}
Type: Local

Generates code that determines if all array and string indexing expressions are within bounds, and all assignments to scalar and subrange variables are within range. If range checking fails while the directive is enabled, a run-time error is reported. The range checking directive should be used only for debugging, as it decreases program performance and increases program size.

Stack Checking

{$S+} or {$S–}
Default: {$S+}
Type: Local

Creates code that checks whether enough space remains on the stack for local variables before executing a procedure or function. If a stack overflow occurs when the directive is enabled, the program displays a run-time error message and terminates.

VAR String Checking

{$V+} or {$V–}
Default: {$V+}
Type: Local

Compares the declared type of a **VAR** type string parameter with the string type actually being passed as a parameter when the directive is enabled. The types must be identical. If the directive is disabled, no type checking is performed.

B.2 Parameter Directives

Parameter directives instruct the compiler to take a certain action, based on parameters that are passed to the compiler. A space always separates the single directive letter and the parameter.

The individual parameter directives are described below.

Include

{**$I** *Filename*}
Type: Local

Includes a named source file in the compilation immediately after the directive. QuickPascal assumes the .INC extension if none is specified. If the include file is in a different path from the one defined in the QuickPascal Environment command in the Options menu, you must type in the entire pathname.

You cannot have an include file within a **BEGIN...END** block.

Link

{**$L** *ObjectFile*}
Type: Local

Links a relocatable object file containing external routines written in assembly language. If the .OBJ file is in a different path from the one defined in the QuickPascal Environment command in the Options menu, you must type in the entire pathname. For additional information on linking with assembly language, see Chapter 12, "Advanced Topics."

Memory

{**$M** *StackSize*, *MinHeap*, *MaxHeap*}
Default: {**$M 16384, 0, 655360**}
Type: Global

Sets stack and heap memory allocation. Three parameters are passed: stack size, minimum heap size, and maximum heap size. Each parameter must be an integer within a specific range. See the list of ranges in Table B.1 below.

Table B.1 Minimum and Maximum Memory Allocation Parameters

Parameter	Minimum Value	Maximum Value
StackSize	1024	65520
MinHeap	0	655360
MaxHeap	*MinHeap*	655360

B.3 Conditional Directives

Conditional directives produce different code from the same source file if a conditional identifier is defined. The conditional directives are, in effect, a variation of the IF control statement. If a condition is true, code between the **{$IF***xxx Condition***}** and **{$ENDIF}** directives is compiled. If the condition is false, the compiler skips the code between the two directives. An optional **{$ELSE}** directive provides further control.

```
{$DEFINE DEBUG}

{$IFDEF DEBUG}
    Writeln('tax: ', tax, 'total: ', total)
{$ELSE}
    Writeln('amount: ',tax + total);
{ENDIF}
```

You define and undefine identifiers with the **{$DEFINE** *Name***}** and **{$UNDEF** *Name***}** directives. Conditional identifiers are treated separately from similarly named program constants and variables. In addition to user-defined control identifiers, QuickPascal incorporates several predefined identifiers (listed in Table B.2 below).

Table B.2 QuickPascal Predefined Conditional Identifiers

Conditional Identifier	State	Purpose
VER10	Defined	Indicates the compiler version
MSDOS	Defined	Indicates the operating system
CPU86	Defined	Indicates the 80x86 CPU family
CPU87	Defined if an 80x87 is present at compile time	Indicates the use of an 80x87 CPU

Conditional directives may be placed anywhere in the source file. The individual conditional directives are described below.

DEFINE

{$DEFINE *Identifier***}**

Defines a conditional identifier. The identifier exists through the rest of the compilation, or until undefined with the UNDEF directive.

ELSE

{$ELSE}

Compiles or skips the source based on the identifier in the preceding IF directive.

ENDIF

{$ENDIF}

Ends the conditional compilation associated with the most recent IF directive.

IFDEF

{$IFDEF *Identifier*}

Compiles the source that follows if the identifier has been defined.

IFNDEF

{$IFNDEF *Identifier*}

Compiles the source following the directive if the identifier is undefined.

IFOPT

{$IFOPT *Switch*}

Compiles the source that follows if the specified switch has been set as described. For example,

```
{$IFOPT I-}
    result := IOResult;
{$ENDIF}
```

UNDEF

{$UNDEF *Identifier*}

Undefines a previously declared identifier for the remainder of the compilation or until the identifier is once again defined.

Appendix C

Summary of Standard Units

QuickPascal comes with several standard units that expand your programs' capabilities. These units—collections of variables, constants, data types, procedures, and functions—enhance your control over the screen and text, DOS routines, printer use, and graphics. QuickPascal automatically inserts the **System** standard unit whenever you compile a program. **System** supplies all of the QuickPascal standard procedures and functions.

Crt Unit

The **Crt** unit extends your control of the screen, keyboard input, and sound. **Crt** has procedures and functions that manipulate the screen colors, cursor position, text attributes, and windows. It also contains functions for checking keyboard activity and lets you turn the internal speaker on and off.

Dos Unit

The **Dos** unit gives you access to file-handling and operating-system routines. Programs that need to access DOS procedures and functions in order to set or get the time and date, to determine disk sizes and available disk space, or to control software interrupts can do so with the **Dos** unit. Other procedures and functions within the **Dos** unit let you manipulate file names and file attributes.

Printer Unit

The **Printer** unit lets you use a printer. It declares a text file, "Lst," and sends the output of **Write** and **Writeln** calls that use Lst to the printer port rather than to the screen. **Printer** accepts any type of formatting available to **Write** and **Writeln**. (Lst directs its output to LPT1. Make sure your printer connects to that port.)

MSGraph Unit

The **MSGraph** unit provides graphics in QuickPascal. It supports a large number of procedures and functions that draw geometric shapes, fill figures with colors and patterns, manipulate images, create windows and viewports, and display text in various sizes and type styles. You can display graphics on a wide variety of video adapters and can choose the best combination of screen resolution and colors for your needs.

System Unit

The **System** unit contains all of the standard procedures and functions (**Writeln**, **Pred**, **Copy**, and so on), which you think of as always being available. You do not need to include **System** in the **USES** list; QuickPascal automatically uses the **System** unit whenever you compile a program.

Appendix D

Quick Reference Guide

D.1 Keywords, Procedures, and Functions

Use the QP Advisor for complete descriptions and information.

Abs(*x* **)**

Returns the absolute value of x.

ABSOLUTE

Declares a variable to reside at a specific memory location.

Addr(*x* **)**

Returns the address of *x*, where *x* is any variable, typed constant, procedure identifier, or function identifier.

AND

Acts as logical or bitwise **AND** operator.

Append(*FileVariable* **)**

Opens an existing **Text** file *FileVariable* to append additional text.

ArcTan(*x* **)**

Returns the arctangent of x.

ARRAY [*Ranges* **] OF** *Type*

Defines *Type* as the base type, and *Ranges* as the range of indices, for an array type, variable, or constant.

Assign(*FileVariable*, *Name* **)**

Assigns the file variable *FileVariable* to the external file named in *Name*.

BEGIN

Starts a statement block.

BlockRead(*FileVariable*, *Buffer*, *Count* [[, *RecordsRead*]] **)**

Reads *Count* number of records into memory from the file specified by *FileVariable*, beginning with the first byte occupied by *Buffer*. Optionally **BlockRead** returns the number of records successfully read in *RecordsRead*.

BlockWrite(*FileVariable*, *Buffer*, *Count* [[, *RecordsWritten*]] **)**

Writes *Count* number of records from memory to the file specified by *FileVariable*, beginning with the first byte occupied by *Buffer*. Optionally **BlockWrite** returns the number of records successfully written in *RecordsWritten*.

CASE *Selector* **OF** *Constant* **:** *StatementBlock*; ... **END**

Executes the *StatementBlock* whose *Constant* matches *Selector*.

ChDir(*NewDir* **)**

Makes *NewDir* the current directory.

Chr(*x* **)**

Returns the ASCII character with ordinal value *x*.

Close(*FileVariable* **)**

Closes the open file specified by the file variable *FileVariable*.

Concat(*Str1* [[, *Str2*,*Str3*...]] **)**

Returns the argument strings (*Str1*, *Str2*, etc.) as a single string.

CONST

Starts a constant definition section.

Copy(*String*, *Start*, *Count* **)**

Returns a substring of *String*, *Count* characters long, beginning with character number *Start*.

Cos(*x* **)**

Returns the cosine of x.

CSeg

Returns the value of the CS register.

CSTRING [[[*Length*]]]

Defines a variable or constant as a series of up to 255 characters ending in a null byte, as in the C programming language. The integer *Length* specifies the maximum length of the string.

Dec(*x*[[, *Step*]] **)**

Decrements the variable *x* by 1; if *Step* is specified, *x* is decremented by *Step*.

Delete(*String*, *Start*, *Count* **)**

Returns a copy of *String* with *Count* characters removed, beginning with character number *Start*.

Dispose(*p* **)**

Removes the dynamic variable that *p* points to and returns the memory to the heap.

DIV

Acts as the integer division operator. *i* **DIV** *j* returns the quotient of *i* divided by *j* rounded to the integer nearest zero.

DO

Introduces the statement block with **WHILE**, **FOR**, and **WITH**.

DOWNTO

Indicates that a **FOR** statement's end value is less than its start value and that the control variable is decremented by 1 with each iteration.

DSeg

Returns the value of the DS register.

ELSE

Begins the default clause in an **IF...THEN...ELSE** or **CASE** statement.

END

Ends a statement block.

Eof[[(*FileVariable*)]]

Returns the end-of-file status for the file *FileVariable*. **Eof** returns **True** if the end of the file has been reached, otherwise it returns **False**. If you omit *FileVariable*, **Eof** checks the status of the standard input file.

Eoln[[(*FileVariable*)]]

Returns the end-of-line status. **Eoln** returns **True** if the end of the line has been reached, otherwise it returns **False**. If you omit *FileVariable*, **Eoln** checks the status of the standard input file.

Erase(*FileVariable* **)**

Erases the file referred to by *FileVariable*.

Exit

In a procedure or function, causes control to return to the main program. In the main program, **Exit** halts program execution.

Exp (*x* **)**

Returns the exponential of *x*, that is, the value *e* raised to the power of *x*.

EXTERNAL

Identifies a separately compiled procedure or function written in assembly language.

FILE [[OF *ComponentType*]]

Declares a file type composed of values of type *ComponentType*.

FilePos(*FileVariable* **)**

Returns the current position of an open file referred to by file variable *FileVariable*.

FileSize(*FileVariable* **)**

Returns the size (in bytes) of an open file referred to by file variable *FileVariable*.

FillChar(*Variable*, *Count*, *Character* **)**

Fills *Count* contiguous bytes of memory with *Character* (either an ASCII value or a literal character enclosed in single quotes), starting with the first byte occupied by *Variable*.

Flush(*FileVariable* **)**

Writes to disk the buffer of the text file referred to by *FileVariable*.

FOR *ControlVariable* **:=** *StartVal* {**TO** | **DOWNTO**} *EndVal* **DO** *StatementBlock*

Executes *StatementBlock* as long as the value of *ControlVariable* is between its start value *StartVal* and its end value *EndVal*, inclusive. Use **TO** if *EndVal* is greater than *StartVal*, or **DOWNTO** if *EndVal* is less than *StartVal*. **FOR** increments (with **TO**) or decrements (with **DOWNTO**) *ControlVariable* by 1 each time it executes *StatementBlock*.

FORWARD

Declares a procedure but omits its definition until a second declaration. This permits mutually referencing procedures.

Frac(*x* **)**

Returns the fractional portion of the real number *x*.

FreeMem(*Pointer*, *Size* **)**

Frees *Size* bytes of dynamic memory at address *Pointer*.

FUNCTION *Identifier*[[([[**VAR**]] *Param1*[[, *Param2* ...]] :*Ptype1*
[[; [[**VAR**]] *Param3*[[, *Param4* ...]] :*Ptype2* ...]])]]:*Typename*

Defines a function named *Identifier* that returns a value of type *Typename*. Any parameters (*Param1*, *Param2*, etc.) to the function must be declared along with their type (*Ptype1*, *Ptype2*, etc.). Every function must return a value.

GetDir(*Drive*, *Path*)

Returns the current directory in the string *Path*, given the integer *Drive*. Setting *Drive* to 0 uses the current drive, *Drive* to 1 uses drive A, *Drive* to 2 uses drive B, and so on.

GetMem(*Pointer*, *Size*)

Creates a new dynamic variable of *Size* bytes from heap memory, setting location *Pointer*.

GOTO *LabelName*

Unconditionally transfers program control to the statement at label *LabelName*.

Halt[[(*Code*)]]

Halts program execution and returns to DOS. You can optionally include the program's exit code.

Hi(*x*)

Returns the high-order byte of *x*, a word or integer.

IF *BooleanExpression* **THEN** *StatementBlock1* [[**ELSE** *StatementBlock2*]]

Executes *StatementBlock1* if *BooleanExpression* is **True**, otherwise executes *StatementBlock2*.

IMPLEMENTATION

Indicates the beginning of the unit section that defines the unit's procedures and functions. The identifiers declared in this section are private.

IN

Acts as the member-of operator for sets. **IN** is used to test for the presence of an element in a set.

Inc(*x* [[,Step]])

Increments the variable *x* by 1; if *Step* is specified, *x* is incremented by *Step*.

INHERITED

Modifies a message to refer to the parent method.

INLINE(*MachineCode* **)**

Defines machine code that is inserted into the program.

Insert(*String*, *SubString*, *Start* **)**

Inserts the string *SubString* into the string *String*, beginning at character number *Start*. If the resulting string is more than 255 characters long, it is truncated to 255 characters.

Int(*x* **)**

Returns the integral portion of the real number *x*.

INTERFACE

Indicates the beginning of the unit section that declares the variables, constants, procedures, and functions available to the calling program.

INTERRUPT

Declares a procedure as an interrupt procedure. Interrupt procedures may handle program interrupts.

IOResult

Returns the status of the most recent I/O operation. A status of 0 indicates the I/O operation was successful.

LABEL {*Identifier* | 0..9999}

Declares the label *Identifier* or a number. Labels are the destination of GOTO statements.

Length(*String* **)**

Returns the length of the string *String*.

Ln(*x* **)**

Returns the natural logarithm of *x*.

Lo(*x* **)**

Returns the low-order byte of *x*, a word or integer.

Mark(*Pointer* **)**

Saves the current top of the heap in *Pointer*.

MaxAvail

Returns the size (in bytes) of the largest continuous block of free memory in the heap.

MemAvail

Returns the total free memory in the heap in bytes.

Member(*ObjectVariable, ClassId* **)**

Returns **True** if *ObjectVariable* is a member of the class *ClassId*, and **False** otherwise.

MkDir(*NewDir* **)**

Creates a new directory with the path given in the string *NewDir*.

MOD

Acts as the modular division operator. The expression *i* **MOD** *j* returns the remainder of *i* **DIV** *j*.

Move(*Source, Destination, Count* **)**

Copies *Count* bytes of *Source* to *Destination*.

New(*Pointer* **)**

Allocates space for a new dynamic variable in the heap and sets *Pointer* to the address of the new variable. The type of pointer determines how much memory **New** allocates.

NIL

Indicates the value of a pointer that does not point to anything.

NOT

Acts as the logical or bitwise negation operator.

OBJECT[[(*Parent***)]]**

Defines an object type or class, descended from the *Parent* class.

Odd(*x* **)**

Returns **True** if the integral *x* is odd, **False** if *x* is even.

OF

Identifies the base type of an **ARRAY**, **FILE**, or **SET**, or introduces the constant list found in a **CASE** statement.

Ofs(*x* **)**

Returns the offset of *x*, a variable, typed constant, procedure, or function name.

OR

Acts as the logical or bitwise **OR** operator.

Ord(*x* **)**

Returns the ordinal number of *x* (an ordinal-type variable).

OVERRIDE

Redefines a parent method to do something new.

PACKED

Required by standard Pascal. Serves only to distinguish types in QuickPascal.

ParamCount

Returns the number of command-line parameters.

ParamStr(*i* **)**

Returns a string that is the *i*th command-line parameter. Parameter number 0 is the program path in DOS versions 3.1 and later.

Pi

Returns the value Pi (3.1415926535897932385). The precision of the value varies, depending on the floating-point hardware present.

Pointer

Defines a variable as a generic pointer type. A variable of type **POINTER** must be assigned to a variable of a specific pointer type before it can be dereferenced.

Pos(*SubString*, *String* **)**

Returns the starting position of string *SubString* in string *String*.

Pred(*x* **)**

Returns the predecessor of *x* in the list of values of its ordinal type.

PROCEDURE *Identifier*[[([[**VAR**]] *Param1*[[, *Param2* ...]] :*Ptype1*
[[; [[**VAR**]] *Param3*[[, *Param4* ...]] :*Ptype2* ...]])]]

Defines a procedure named *Identifier*. Any parameters (*Param1*, *Param2*, etc.) to the procedure must be declared along with their type (*Ptype1*, *Ptype2*, etc.).

PROGRAM [[*ProgName*]]

Declares a program with the name *ProgName*.

Ptr(*Seg*, *Off* **)**

Converts a segment and offset to a pointer address; *Seg* and *Off* are both of type **Word**.

Random[[(*Limit*)]]

Returns a random real number between 0 and 1, or a random whole number between 0 and *Limit*.

Randomize

Initializes the random number generator.

Read([[*FileVariable*,]]*Var1* [[, *Var2* ...]])

Reads one or more values from the standard input device or, optionally, from the file specifier *FileVariable*.

Readln([[*FileVariable* ,]]*Var1* [[, *Var2* ...]])

Executes the **Read** procedure, then skips to the beginning of the next line of the text file specified by *FileVariable*.

RECORD *FieldList* **END**

Creates a compound variable consisting of the items listed in *FieldList*. The different fields in a record can have different types.

Release(*Pointer*)

Returns the value of the heap-top pointer to *Pointer*, which was previously obtained by **Mark**.

Rename(*FileVariable*, *NewName*)

Renames the external file specified by file variable *FileVariable* to the name string *NewName*.

REPEAT *StatementBlock* **UNTIL** *BooleanExpression*

Executes *StatementBlock* as long as *BooleanExpression* remains **False**. When *BooleanExpression* becomes **True**, control passes to the statement following the **UNTIL** statement.

Reset(*FileVariable* [[, *Size*]])

Opens the file specified by *FileVariable* with a data transfer unit size of *Size*.

Rewrite(*FileVariable* [[, *Size*]])

Creates and opens a new file *FileVariable*, with a data transfer unit size of *Size*. If *FileVariable* already exists, it is truncated to a length of zero.

RmDir(*Dir*)

Removes the empty directory named *Dir*.

Round(*x*)

Rounds the real number *x* to the nearest whole number.

RunError[[(*ErrorNum*)]]

Halts program execution with the run-time error number *ErrorNum*, or run-time error 0 if *ErrorNum* is not specified.

Seek(*FileVariable*, *Pos*)

Moves the current position of the file specified by *FileVariable* to the position *Pos*.

SeekEof [[(*FileVariable*)]]

Returns the end-of-file status for the text file specified by *FileVariable*, skipping blanks, tabs, and end-of-line markers. The actual file position does not change.

SeekEoln[[(*FileVariable*)]]

Returns the end-of-line status for the text file specified by *FileVariable*, skipping blanks and tabs.

Seg(*x*)

Returns the segment containing *x*, a variable, typed constant, procedure, or function name.

Self

Refers to the object that received a message, used by methods.

SET OF *OrdinalType*

Identifies *OrdinalType* as the base type for a set type, constant, or variable.

SetTextBuf(*FileVariable*, *Buffer* [[,*Size*]])

Assigns the text file *FileVariable* a buffer of *Size* bytes in memory starting at *Buffer*. If *Size* is omitted, *SizeOf* (*Buffer*) is assumed.

SHL

Acts as the bitwise shift-left operator. The expression *i* **SHL** *j* shifts the value of *i* to the left by *j* bits.

SHR

Acts as the bitwise shift-right operator. The expression *i* **SHR** *j* shifts the value of *i* to the right by *j* bits.

Sin(*x*)

Returns the sine of *x*.

SizeOf(*x*)

Returns the size of the variable, typed constant, or type *x* in bytes.

SPtr

Returns the value of the SP register, the current offset of the stack pointer in the stack segment.

Sqr(*x* **)**

Returns the square of *x*.

Sqrt(*x* **)**

Returns the square root of *x*, a positive integer or real number.

SSeg

Returns the value of the SS register, the stack segment address of the stack pointer.

Str(*Number*[[**:***Width*[[**:***Decimals*]]]], *String* **)**

Converts the numeric value *Number* to the string *String*. The arguments *Width* and *Decimals* specify the total width and number of decimal places that will appear in the string.

STRING[[[*Length*]]]

Defines a variable or constant as a series of up to 255 characters. The integer *Length* specifies the maximum length of the string.

Succ(*x* **)**

Returns the successor to *x* in the list of values of its ordinal type.

Swap(*x* **)**

Exchanges the high- and low-order bytes of *x*, an integer or word.

THEN *StatementBlock*

Used in the second half of an **IF...THEN** statement. If the condition in the **IF** portion of the statement is **True**, the statements in *StatementBlock* execute.

TO

Indicates that a **FOR** statement's ending value is greater than its starting value and that the control variable will be incremented by one.

Trunc(*x* **)**

Truncates a real value *x* to an integer.

Truncate(*FileVariable* **)**

Truncates the file specified by *FileVariable* at the current file position.

TYPE

Begins a type definition section.

UNIT *Identifier*

Identifies the code that follows as a unit and gives the unit the name *Identifier*.

UNTIL *BooleanExpression*

Terminates a **REPEAT** statement when *BooleanExpression* becomes **True**.

UpCase(*Char* **)**

Returns character *Char* in uppercase.

USES *Identifier*[[**,** *Identifier*]]...

Identifies units required by the program to resolve references to identifiers defined within the units.

Val(*String*, *Number*, *ErrorPosition* **)**

Converts the numeric string *String* to its numeric representation *Number*. If *String* does not represent a number, *ErrorPosition* returns the position of the first offending character.

VAR

Begins a variable declaration section or declares a variable parameter.

WHILE *BooleanExpression* **DO** *StatementBlock*

Executes *StatementBlock* as long as *BooleanExpression* remains **True**. When *BooleanExpression* becomes **False**, control passes to the statement following *StatementBlock*.

WITH *RecordName1* [[**,** *RecordName2* **...]]DO** *StatementBlock*

Allows statements within *StatementBlock* to refer to the fields of one or more records without specifying the names of the records (*RecordName1*, *RecordName2*...).

Write([[*FileVariable* **,]]***Var1*[[**,***Var2***...]]** **)**

Writes one or more values to the standard output device or to the file specified by *FileVariable*.

Writeln([[*FileVariable* **,]]***Var1*[[**,***Var2***...]]** **)**

Executes the **Write** procedure, then sends an end-of-line marker to the standard output device or the file specified by a *FileVariable*.

XOR

Acts as the logical or bitwise exclusive-or operator.

D.2 Crt Procedures and Functions

AssignCrt(*FileVariable* **)**

Associates a **Text** file variable *FileVariable* with the CRT device (screen).

ClrEol

Clears a line from the cursor to the end of the line.

ClrScr

Clears the window and moves the cursor to the upper left corner.

Delay(*Microseconds* **)**

Pauses program execution for a specified length of time.

DelLine

Deletes the line at the current cursor location.

GotoXY(*x*,*y* **)**

Moves the cursor to designated column and row.

HighVideo

Turns on high-intensity video for the current foreground color.

InsLine

Inserts a blank line at the current cursor location.

KeyPressed

Returns **True** if the keyboard buffer contains a character.

LowVideo

Turns off high-intensity video for the current foreground color.

NormVideo

Restores the text colors and attributes that were in effect at program start-up.

NoSound

Turns off the computer's speaker.

ReadKey

Returns one character from the keyboard buffer but does not echo character on the screen.

Sound(*Frequency* **)**

Generates a tone from the computer's speaker at the specified frequency.

TextBackground(*Color* **)**

Sets the background color for character output.

TextColor(*Color* **)**

Sets the foreground color and blinking attribute for character output.

TextMode(*Mode* **)**

Sets the display to the specified text mode.

WhereX

Returns the current x-coordinate of the text cursor.

WhereY

Returns the current y-coordinate of the text cursor.

Window(*x1*, *y1*, *x2*, *y2* **)**

Defines a text display window. The coordinates give the upper left and lower right corners of the window.

D.3 Dos Procedures and Functions

DiskFree(*DriveNumber* **)**

Returns the number of bytes of free space on the specified drive.

DiskSize(*DriveNumber* **)**

Returns the total capacity in bytes of the specified drive.

DosExitCode

Returns the exit code from a child process.

DosVersion

Returns the version number of the operating system.

EnvCount

Returns the number of variables defined in the DOS environment.

EnvStr(*EnvironmentStringIndex* **)**

Returns the value of a variable from the DOS environment.

Exec(*ProgramPath* , *CommandLine* **)**

Loads and runs a child process while suspending parent process.

FExpand(*FilePath* **)**

Expands a name to a fully qualified DOS path name.

FindFirst(*SearchPattern* , *Attributes*, *Matched*)

Searches the specified directory for the first file matching the given *SearchPattern* and set of attributes.

FindNext(*Matched*)

Searches the specified directory for the next file matching the *SearchPattern* and attributes specified in a previous call to **FindFirst**.

FSearch(*FilePath* , *DirectoryList*)

Searches for a file in a list of directories.

FSplit(*FilePath* , *Directory* , *FileName* , *Extension*)

Separates a path name into its directory, basename, and extension parts.

GetCBreak(*Breaking*)

Gets the current state of DOS CTRL+BREAK checking.

GetDate(*Year*, *Month*, *Day*, *DayOfWeek*)

Gets the current system date.

GetEnv(*EnvironmentStringLabel*)

Returns the current value of a DOS environment variable.

GetFAttr(*FileVariable*, *Attribute*)

Gets a file's attributes.

GetFTime(*FileVariable*, *TimeStamp*)

Gets the **LongInt** representing a file's date and time of modification.

GetIntVec(*InterruptNumber*, *Vector*)

Gets the vector address for a given interrupt number.

GetTime(*Hour*, *Minute*, *Second*, *Sec100*)

Gets the current system time.

GetVerify(*Verifying*)

Gets the current state of the DOS verify flag.

Intr(*InterruptNumber*, *RegisterValues*)

Calls a software interrupt, loading and returning register values.

Keep(*ExitCode*)

Terminates a program but keeps it resident in memory.

MsDos(*RegisterValues*)

Calls DOS interrupt $21.

PackTime(*DateTime*, *TimeStamp* **)**

Converts an unpacked *DateTime* record to a packed **LongInt** *TimeStamp*

SetCBreak(*Breaking* **)**

Turns DOS CTRL+BREAK checking on or off.

SetDate(*Year*, *Month*, *Day* **)**

Sets the current system date.

SetFAttr(*FileVariable*, *Attribute* **)**

Sets a file's attributes.

SetFTime(*FileVariable*, *TimeStamp* **)**

Sets a file's date and time of file modification record.

SetIntVec(*InterruptNumber*, *Vector* **)**

Installs a new interrupt handler. If a program changes an interrupt vector, it must restore it before terminating.

SetTime(*Hour*, *Minute*, *Second*, *Sec100* **)**

Sets the system time.

SetVerify(*Verifying* **)**

Sets the state of the DOS verify flag.

SwapVectors

Swaps interrupt vectors with previously saved values.

UnpackTime(*TimeStamp* , *DateTime* **)**

Converts the **LongInt** *TimeStamp* argument to an unpacked *DateTime* record.

D.4 Printer Unit Interface

The printer unit does not contain any procedures or functions. It connects the file variable Lst with the printer port. Using Lst in a **Write** or **Writeln** statement sends the text to the printer:

```
Write( Lst, 'This text goes to the printer.' );
```

Lst is a text variable assigned to the file variable **PRN** and preconnected to the LPT1 printer port.

D.5 MSGraph Procedures and Functions

_Arc(*x1, y1, x2, y2, x3, y3, x4, y4* **)**

Draws an arc given the bounding rectangle and beginning and ending points in viewport coordinates.

_Arc_wxy(*wxy1, wxy2, wxy3, wxy4* **)**

Draws an arc given the bounding rectangle and beginning and ending points in window coordinates in **_WXYCoord** records.

_ClearScreen(*Area* **)**

Clears the specified area of the screen.

_DisplayCursor(*Toggle* **)**

Specifies whether to turn the cursor back on or leave it off after executing graphics routines.

_Ellipse(*Control, x1, y1, x2, y2* **)**

Draws an ellipse given the fill control and bounding rectangle in viewport coordinates.

_Ellipse_w(*Control, wx1, wy1, wx2, wy2* **)**

Draws an ellipse given the fill control and bounding rectangle in window coordinates.

_Ellipse_wxy(*Control, wxy1, wxy2* **)**

Draws an ellipse given the fill control and bounding rectangle in window coordinates in the **_WXYCoord** records *wxy1* and *wxy2*.

_FloodFill(*x, y, Boundary* **)**

Fills an area with the current color and fill mask. If the point (*x*, *y*) lies inside the figure, it fills the interior; if (*x*, *y*) lies outside the figure, it fills the background. *x* and *y* are given in viewport coordinates.

_FloodFill_w(*wx, wy, Boundary* **)**

Fills an area with the current color and fill mask. If the point (*x*, *y*) lies inside the figure, it fills the interior; if (*x*, *y*) lies outside the figure, it fills the background. *wx* and *wy* are given in window coordinates.

_GetActivePage

Returns the current active page number.

_GetArcInfo(*Start, End, Paint* **)**

Returns information about the most recently drawn **_Arc** or **_Pie**. The *Start* point, *End* point, and *Paint* point are returned in **_XYCoord** records.

_GetBkColor

Returns the current background color.

_GetColor

Returns the current color index.

_GetCurrentPosition(*xy* **)**

Returns the current graphics cursor position in viewport coordinates in the **_XYCoord** record *xy*.

_GetCurrentPosition_wxy(*wxy* **)**

Returns the current graphics cursor position in window coordinates in the **_WXYCoord** record *wxy*.

_GetFillMask(*Mask* **)**

Returns the current fill mask in *mask*, if one is defined.

_GetFontInfo(*FInfo* **)**

Gets the current font characteristics and returns them in the **_FontInfo** record *FInfo*.

_GetGTextExtent(*TextString* **)**

Returns the pixel width required to print the string *TextString* in the current font with the **_OutGText** function.

_GetGTextVector(*Vector* **)**

Returns the current rotation vector that is applied to font-based text output in the **_XYCoord** record *Vector*. The default is (1,0).

_GetImage(*x1, y1, x2, y2, Image* **)**

Stores the screen image inside the bounding rectangle specified in viewport coordinates. Stores the image in the buffer *Image*. The rectangle must be completely within the current clipping region.

_GetImage_w(*wx1, wy1, wx2, wy2, Image* **)**

Stores the screen image inside the bounding rectangle specified in window coordinates. Stores the image in the buffer *Image*. The rectangle must be completely within the current clipping region.

_GetImage_wxy(*wxy1*, *wxy2*, *Image* **)**

Stores the screen image inside the bounding rectangle specified in window coordinates in the **_WXYCoord** records *wxy1* and *wxy2*. Stores the image in the buffer *Image*. The rectangle must be completely within the current clipping region.

_GetLineStyle

Returns the current line-style mask.

_GetPhysCoord(*x*, *y* , *xy* **)**

Translates the viewport coordinates (*x*, *y*) into physical screen coordinates and returns them in the **_XYCoord** record *xy*.

_GetPixel(*x*, *y* **)**

Returns the pixel value (color index) at the location specified in viewport coordinates (*x*, *y*).

_GetPixel_w(*wx*, *wy* **)**

Returns the pixel value (color index) at the location specified in window coordinates (*wx*, *wy*).

_GetTextColor

Returns the color index (attribute) of the current text color.

_GetTextCursor

Returns the current cursor attribute (shape) in text modes.

_GetTextPosition(*r*, *c* **)**

Returns the current row and column position of the text cursor.

_GetTextWindow(*r1*, *c1*, *r2*, *c2* **)**

Returns the boundaries of the current text window in row and column coordinates.

_GetVideoConfig(*vc* **)**

Returns the current graphics environment configuration in the **_VideoConfig** record *vc*.

_GetViewCoord(*x*, *y*, *xy* **)**

Translates physical coordinates (*x*, *y*) into viewport coordinates, returning the viewport coordinates in the **_XYCoord** record *xy*.

_GetViewCoord_w(*wx*, *wy*, *xy* **)**

Translates window coordinates (*wx*, *wy*) into viewport coordinates, returning the viewport coordinates in the **_XYCoord** record *xy*.

_GetViewCoord_wxy(*wxy, xy* **)**

Translates window coordinates given in the **_WXYCoord** record *wxy* into viewport coordinates, returning the viewport coordinates in the **_XYCoord** record *xy*.

_GetVisualPage

Returns the current visual page number.

_GetWindowCoord(*x, y, wxy* **)**

Translates viewport coordinates (*x*, *y*) into window coordinates and returns them in the **_WXYCoord** record *wxy*.

_ImageSize(*x1, y1, x2, y2* **)**

Returns the number of bytes needed to store the image inside the bounding rectangle specified by the viewport coordinates (*x1*, *y1*), (*x2*, *y2*). This function returns a **LongInt**.

_ImageSize_w(*wx1, wy1, wx2, wy2* **)**

Returns the number of bytes needed to store the image inside the bounding rectangle specified by the window coordinates (*wx1*, *wy1*), (*wx2*, *wy2*). This function returns a **LongInt**.

_ImageSize_wxy(*wxy1, wxy2* **)**

Returns the number of bytes needed to store the image inside the bounding rectangle specified by the window coordinates in the **_WXYCoord** records *wxy1* and *wxy2*. This function returns a **LongInt**.

_LineTo(*x, y* **)**

Draws a line from the current position to the point specified in viewport coordinates, using the current color, line-style mask, and logical write mode.

_LineTo_w(*wx, wy* **)**

Draws a line from the current position to the point specified in window coordinates, using the current color, line-style mask, and logical write mode.

_MoveTo(*x, y* **)**

Moves the graphics cursor to the point specified by the viewport coordinates (*x*, *y*).

_MoveTo_w(*wx, wy* **)**

Moves the graphics cursor to the point specified by the window coordinates (*wx*, *wy*).

_OutGText(*TextString* **)**

Prints *TextString* on the screen using the current font and graphics color at the current graphics cursor position.

_OutMem(*TextString, Length* **)**

Prints *TextString* on the screen at the current text cursor position. This procedure treats ASCII 0, 10, and 13 as graphics characters. Formatting is not supported.

_OutText(*TextString* **)**

Prints *TextString* on the screen at the current text cursor position. Formatting is not supported, except for carriage return and line feed.

_Pie(*Control, x1, y1, x2, y2, x3, y3, x4, y4* **)**

Draws a pie-shaped wedge given the fill control, bounding rectangle, starting point, and ending point. All coordinates are given in viewport coordinates.

_Pie_wxy(*Control, wxy1, wxy2, wxy3, wxy4* **)**

Draws a pie-shaped wedge given the fill control, bounding rectangle, starting point, and ending point. All coordinates are given in **_WXYCoord** records.

_PutImage(*x, y, Image, Action* **)**

Transfers the image stored in the buffer *Image* to the screen (using the logical operator *Action*) with the upper left corner at the specified viewport coordinates. If the image does not completely fit in the current clipping region, the image is not transferred.

_PutImage_w(*wx, wy, Image, Action* **)**

Transfers the image stored in the buffer *Image* to the screen (using the logical operator *Action*) with the upper left corner at the specified window coordinates. If the image does not completely fit in the current clipping region, the image is not transferred.

_Rectangle(*Control, x1, y1, x2, y2* **)**

Draws a rectangle given the fill control and bounding rectangle in viewport coordinates, using the current color, line-style mask, and logical write mode.

_Rectangle_w(*Control, wx1, wy1, wx2, wy2* **)**

Draws a rectangle given the fill control and bounding rectangle in window coordinates, using the current color, line-style mask, and logical write mode.

_Rectangle_wxy(*Control , wxy1, wxy2* **)**

Draws a rectangle given the fill control and bounding rectangle in the window coordinates specified in the **_WXYCoord** records *wxy1* and *wxy2* and using the current color, line-style mask, and logical write mode.

_RegisterFonts(*PathName* **)**

Registers the fonts in the file given in *PathName*.

_RemapAllPalette(*NewPalette* **)**

Remaps the entire color palette simultaneously, given an array of color values.

_RemapPalette(*Index*, *Value* **)**

Remaps the color index *Index* to the color value *Value*.

_ScrollTextWindow(*Count* **)**

Scrolls the current text window by *Count* lines. If *Count* is positive, the text scrolls up; if negative, the text scrolls down.

_SelectPalette(*Number* **)**

Sets the active palette to palette number *Number* in CGA and Olivetti graphics modes.

_SetActivePage(*Page* **)**

Makes page number *Page* the active page for graphics output, available only for configurations that support multiple screen pages.

_SetBkColor(*Color* **)**

Sets the current background color to *Color*.

_SetClipRgn(*x1*, *y1*, *x2*, *y2* **)**

Limits the display of subsequent graphics and font text to the bounding rectangle, given in viewport coordinates.

_SetColor(*Color* **)**

Sets the current graphics color to color index *Color*.

_SetFillMask(*Mask* **)**

Defines the current fill mask.

_SetFont(*Options* **)**

Selects a new active font from one of the registered fonts. Font selection is based on the characteristics specified by *Options*.

_SetGTextVector(*xvect*, *yvect* **)**

Sets the current rotation vector that is applied to font-based text output. The default is horizontal text (1,0).

_SetLineStyle(*Style* **)**

Sets the current line-style mask to *Style*. The line-style mask affects the output of **_LineTo** and **_Rectangle**.

_SetPixel(*x*, *y* **)**

Changes the specified pixel to the current color. The point is specified in viewport coordinates.

_SetPixel_w(*wx*, *wy* **)**

Sets the specified pixel to the current color. The point is specified in window coordinates.

_SetTextColor(*Color* **)**

Sets the text color to color index *Color*. Subsequent text output from **_OutText** and **_OutMem** appears in the new color.

_SetTextCursor(*Attr* **)**

Sets the cursor shape in text mode.

_SetTextPosition(*r*, *c* **)**

Moves the text cursor position to the row and column specified, relative to the current text window.

_SetTextRows(*Rows* **)**

Sets the number of rows available for text modes.

_SetTextWindow(*r1*, *c1*, *r2*, *c2* **)**

Defines the upper left and lower right boundaries of the current text window. Output from **_OutText** and **_OutMem** is limited to this window.

_SetVideoMode(*Mode* **)**

Selects a screen mode for a particular hardware/display configuration.

_SetVideoModeRows(*Mode*, *Rows* **)**

Sets the screen mode and number of text rows for a particular hardware/display configuration.

_SetViewOrg(*x*, *y*, *Org* **)**

Moves the viewport origin to the physical coordinates (*x*, *y*), and returns the previous origin in the **_XYCoord** record *Org*.

_SetViewport(*x1*, *y1*, *x2*, *y2* **)**

Defines the graphics viewport (defines the clipping region and sets the viewport origin to the upper left corner of the region) given the bounding rectangle in physical coordinates.

_SetVisualPage(*Page* **)**

Makes *Page* the current visual page (requires a configuration that supports multiple screen pages).

_SetWindow(*FInvert* , *x1*, *y1*, *x2*, *y2*)

Creates a floating-point graphics window given a bounding rectangle in window coordinates. If *FInvert* is **True**, the window is inverted vertically.

_SetWriteMode(*WMode*)

Sets the logical operation applied to line-drawing output to one of the following: **_Gor**, **_Gand**, **_GPReset**, **_GPSet**, or **_Gxor**. Only **LineTo** and **_Rectangle** are affected.

_UnRegisterFonts

Disables fonts and frees memory previously allocated by **_RegisterFonts**.

_WrapOn(*Option*)

Controls text wrapping for **_OutText** and **_OutMem** when the output reaches the edge of the text window. By default, text wrapping is enabled.

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

MICROSOFT PRODUCT ASSISTANCE REQUEST

Microsoft Product Support Services - Phone (206) 454-2030

Instructions

When you need assistance with a Microsoft product, call our Product Support Services group at (206) 454-2030. So that we can answer your question as quickly as possible, please gather all information that applies to your problem. Note or print out any on-screen messages you get when the problem occurs. Have your manual and product disks close at hand and have all the information requested on this form available when you call.

Diagnosing a Problem

So that we can assist you more effectively, please be prepared to answer the following questions regarding your problem, your software, and your hardware.

1. Can you reproduce the problem?
   ❒ yes ❒ no

2. Does the problem occur with another copy of the original disk of your Microsoft Software?
   ❒ yes ❒ no

3. Does the problem occur with another system (if available)?
   ❒ yes ❒ no

4. If you were running other windowing or memory-resident software at the same time, does the problem also occur when you don't use the other software?
   ❒ yes ❒ no

Product

Product name

Version Number

Registration Number

Software

Operating System

Name/Version number

Windowing Environment

If you were running Microsoft Windows or another windowing environment, give name and number of windowing software:

CD ROM Software

Name/Version number

Other Software

Name/Version number of any other software you were running when problem occurred, including memory-resident software (such as keyboard enhancers or print spoolers):

Hardware

So that we can assist you more effectively, please be prepared to answer the following questions regarding your problem, your software, and your hardware.

Computer

Manufacturer/model _____ Total memory _____

Floppy-disk drives
Number: ❒ 1 ❒ 2 ❒ Other
Size: ❒ 3 1/2" ❒ 5 1/4"
Number of Sides: ❒ 1 ❒ 2
Density: ❒ Single ❒ Double ❒ Quad
Capacity:
5 1/4": ❒ 160K ❒ 360K ❒ 1.2 megabytes
3 1/2": ❒ 360K ❒ 400K ❒ 720K ❒ 800K
❒ 1.4 megabytes

System Memory

Manufacturer/model _____ Total memory _____
(If using DOS, you can run CHKDSK to determine the amount of memory available. If using Apple Macintosh Finder, select "About The Finder..." from the Apple menu to determine the amount of memory available.)

Peripherals

Hard Disk

Manufacturer/model _____ Capacity(megabyte) _____

Printer/Plotter

Manufacturer/model _____

❒ Serial ❒ Parallel

Printer peripherals, such as font cartridges, downloadable fonts, sheet feeders:

_____

_____

Mouse
Microsoft Mouse: ❒ Bus ❒ Serial ❒ InPort™ ❒ Other

Manufacturer/model _____

Boards
❒ Add-on RAM board

Manufacturer/model _____

❒ Graphics-adapter board

Manufacturer/model _____

❒ Other boards installed

Manufacturer/model _____

Modem

Manufacturer/model _____

CD ROM Player

Manufacturer/model _____

Version of Microsoft MS-DOS® CD ROM Extensions:

_____

Network

Is your system part of a network? ❒ Yes ❒ No

Manufacturer/model _____

What hardware and software does your network use?

_____

_____

_____

Documentation Feedback – Microsoft® QuickPascal

Help us improve our documentation. After you've become familiar with our product, please complete and return this form. Comments and suggestions become the property of Microsoft Corporation.

Which statement best describes your experience with Pascal?

____ I haven't had much programming experience in any language.

____ I have used other languages, but I'm new to Pascal.

____ I have used Pascal occasionally, but I'm still unfamiliar with many of its features.

____ I use Pascal regularly in my professional work, but I'm not a full-time programmer or developer.

____ I'm a full-time programmer or developer using Pascal regularly.

How long ago did you buy this QuickPascal package?

____ months

Have you read *Up and Running* all the way through?

____ I haven't used it at all.

____ I've read part of it. Which parts? _____

_____

____ I've read it all the way through.

Have you used the on-line tutorial, QP Express?

____ I haven't used it at all.

____ I've used part of it. Which parts? _____

_____

____ I've followed it all the way through.

Which statement best summarizes your response to the Pascal language information in *Pascal by Example*?

____ It's too simple; I need more in-depth information.

____ It's about right; I can usually understand it without much difficulty.

____ It's too technical; I find it hard to read and apply.

In this QuickPascal package, some information is provided on-line and some in book form. What's your opinion of this mix?

____ I wish more information were available on-line. Please specify. _____

_____

____ I wish more information were available in book form. Please specify. _____

_____

____ I feel the balance is about right.

Were there any topics you felt weren't covered well enough anywhere in the documentation? Please explain.

_____

_____

_____

_____

_____

Overall, how well does the QuickPascal documentation meet your needs? Rate each from 1 (does not meet your needs at all) to 5 (meets your needs perfectly).

____ *Up and Running* _____

_____

____ *Pascal by Example* _____

_____

____ QP Advisor (on-line help) _____

_____

____ QP Express (on-line tutorial) _____

_____

Now, please return to the question above and tell us, in the space after each item, the main reason for your rating.

Use the back of this form for other suggestions and comments. Please note any errors and special strengths or weaknesses in areas such as programming examples, indexing, overall organization. Which parts of the book do you refer back to most frequently?

(Over)

Name

Address

City/State/Zip

Phone ( ) — ( ) —
(home) (work)

Additional comments:

Please mail this form to:

Microsoft Corporation
Attn: Product Support
16011 NE 36th Way
Box 97017
Redmond, WA 98073-9717

Microsoft Corporation
16011 NE 36th Way
Box 97017
Redmond, WA 98073-9717

Microsoft®

0489 Part No. 06396