PASCAL FOR BEGINNERS

Peter M. L. Lottrup

A practical guide and tutorial to standard Pascal on any personal computer. Includes an introductory Pascal interpreter for the Commodore 64 and 128, ready to type in and use.

COMPUTE! LIBRARY SELECTION

PASCAL for Beginners

Peter M. L. Lottrup

COMPUTE! Publications, Inc.
Part of ABC Consumer Magazines, Inc.
One of the ABC Publishing Companies
Greensboro, North Carolina

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-87455-068-8

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is part of ABC Consumer Magazines, Inc., one of the ABC Publishing Companies, and is not associated with any manufacturer of personal computers. Commodore 64 and Commodore 128 are trademarks of Commodore Electronics Limited.

Contents

Foreword

Learning to program in a structured language like Pascal is easy when you have the right teacher. *COMPUTE!'s Pascal for Beginners* is a step-by-step guide to programming in Pascal on any computer with a standard Pascal compiler or interpreter.

COMPUTE!'s Pascal for Beginners utilizes dozens of special program examples to teach you everything you'll need to know to begin writing your own effective Pascal applications. If you're a beginner to programming, you'll find the explanations concise and easy to understand with straightforward examples that you can study and modify. Programmers experienced in other languages will discover concepts which are familiar as well as the features which make Pascal a powerful and popular structured programming environment.

Once you've learned the fundamentals of Pascal, *COMPUTE!'s Pascal for Beginners* takes you further, showing you how to use some of the more powerful features of Pascal, such as dynamic data structures, and files.

COMPUTE!'s Pascal for Beginners is for any implementation of Pascal that uses standard Pascal. For those with a Commodore 64 or 128, we've also included a Pascal interpreter. While not a full-featured Pascal interpreter, it offers all the power and interactivity necessary to benefit those who are learning, or want to get a feel for, the language. Read Appendix E before typing in the interpreter to see if it meets your needs.

If you're a Commodore 64 or 128 user and prefer not to type in the interpreter, you can purchase a disk that includes the interpreter and most of the Pascal programs from this book. Call COMPUTE! Publications at 1-800-346-6767 (in New York, 212-887-8525), or use the coupon found in the back of this book.

Introduction

Pascal was developed by Niklaus Wirth in Zurich, Switzerland, from the ALGOL 60 computer language. It is currently accepted as an excellent tool for teaching computer programming. Pascal, therefore, is especially popular in high schools and colleges.

Pascal is a high-level structured programming language. High-level languages use English-like words (such as IF-THEN and WHILE-DO) for commands. BASIC, C, COBOL, and FORTRAN are other examples of high-level computer languages. Programs written in a structured format follow a logical flow, which makes program execution easy to follow and understand.

Pascal offers several advantages over other computer languages. Its structure lends itself to modular programming. It's easy to give meaningful names to variables because it allows long variable names, usually up to eight characters. Other advantages include useful implementation of functions and subroutines; program flow that's easy to follow and debug; and constants and data types defined by the user.

Pascal is well-suited for team programming. Each module of a program can be written by a different individual; then, once all the modules have been completed, they can easily be combined to form one complete program.

This book will teach you most of Pascal's features and instructions by using plenty of program examples. The features explained are standard to most versions of Pascal; in the cases where this is not so, the differences are pointed out.

Most of the examples will work on any personal computer using either a Pascal compiler or interpreter. A small number of commercial implementations of Pascal may require minor changes to a few of the examples.

Commodore 64 and 128 readers can use the type-in interpreter found in the appendices.

Before attempting to type in any of the program examples in this book, you should become familiar with the implementation of Pascal that you are using. If you are using the interpreter in this book, read the section of the appendices which explains its use.

Interpreters and Compilers

To work in Pascal, you'll need either an interpreter or a compiler. Interpreters and compilers are programs which translate high-level language instructions into the native language of the computer.

An interpreter is a control program which reads one instruction at a time, translates it into information that the computer can understand, and immediately executes the instruction. This is what happens in BASIC: The BASIC interpreter reads each program instruction, one at a time; translates each instruction into machine-executable code; and then executes it. This method makes BASIC relatively slow, because this interpretation takes place while the program is running.

If a Pascal program is executed by using an interpreter, it will also tend to execute rather slowly. Even so, interpreters are generally more convenient to use when you're learning a language, and syntax errors are usually immediately and accurately reported by the interpreter.

A compiler, on the other hand, translates the whole program into machine-executable code first, before actually executing it. Thus, a compiled program runs much faster than an interpreted program, just as compiled BASIC programs execute faster than interpreted BASIC programs. Even so, a compiler is more complicated to use than an interpreter and is not really necessary when you're learning to program in a new language.

Using This Book

This book is aimed at anyone who wants to learn to program in Pascal. Although you don't need a computer to use the book (you could follow the example programs on paper), a computer will make learning much easier by allowing you to try the example programs and experiment with them. It will also allow you to write and test your own programs.

The book takes you on a step-by-step learning process, introducing you to structured programming and the Pascal language. Many comparisons to BASIC are given for those who are familiar with BASIC.

Beginners should read the first seven chapters in order. These chapters cover such topics as variables, statements to read values and to print values, loops, comparison statements, data types, and sets and arrays. Later chapters deal with functions and procedures, record variables, files, recursion, and other more advanced topics.

Chapter 14 helps you write your own Pascal programs—including correctly planning the different structures and variables that will be used. It also describes several extensions to standard Pascal.

In the appendices is a summary of all Pascal keywords, structures, and so forth, that are covered in the book as well as a list of different Pascal commercial packages on the market and other information.

You'll also find in the appendices a Pascal interpreter for the Commodore 64 and 128, ready to type in and use; it includes all necessary instructions.

Keywords and Identifiers

Throughout the text you'll see references to keywords. A *keyword* is a reserved word in Pascal, a word which cannot be changed and cannot be written with embedded spaces. You'll find a list of keywords in Appendix A. Keywords can be used only for the purpose for which Pascal defines them. The only exceptions are when they are used between quotation marks or in a comment line (like a REMark line in BASIC), which the interpreter or compiler ignores.

Identifiers are words chosen by the programmer for constant values, variables, subroutine names, and so forth. The first character of an identifier must be alphabetic; the others may be any combination of letters and digits (without embedded spaces). The identifier cannot be a reserved Pascal keyword. These rules are the same as those for BASIC variable names. Most standard Pascal interpreters and compilers recognize only the first eight characters of identifiers.

Some identifiers are defined by Pascal itself, and you can use them as they are or redefine their functions. You'll see how to do this in a later chapter. A list of standard identifiers, defined by Pascal, can be found in Appendix B.

Standard Pascal does not differentiate between upper- and lowercase letters. In other words, these variable identifiers are considered the same:

HeLLo
HELLO
hello

Doing Without Line Numbers

How can you write a program without using any line numbers? Most BASIC programmers are accustomed to using line numbers to transfer control from one program statement to another. All but the most recent implementations of BASIC frequently use line numbers in conjunction with statements such as GOTO and GOSUB.

Pascal programs do not use line numbers; as a matter of fact, even though the GOTO statement exists in Pascal, it's rarely, if ever, used in programs and should be avoided as much as possible. Pascal is a structured programming language because it doesn't need GOTO to control the flow of the program. It does not transfer control erratically throughout the program.

At the moment, you may find it difficult to imagine how a program can execute without jumps, but that will become clear as you start programming. You will see line numbers in many of the examples. These are not necessary for the Pascal program to operate, but they are there to help you enter the listings into your computer. They also help in identifying lines in program explanations. *The line numbers do not form part of the program itself.*

Comments in Program Listings

You may add comments to your Pascal program listings just as you do when you use REM in BASIC. A comment in Pascal must be enclosed by the symbols **(*** and ***)** in the following way:

```
(* comment *)
```

The interpreter or compiler will ignore whatever is written inside the comment symbols, and remarks will help make your program clear.

Note: Although the symbols (* and *) are used to enclose comments, it's also common to see the remarks enclosed in braces—{ }. Because this symbol is not found on most computer keyboards and character sets, the alternative is just as common.

About the Interpreter

The Pascal interpreter included in the Appendix is for the Commodore 64 and 128 running in 64 mode. It is *not* a full-featured Pascal interpreter—rather it's intended for those Commodore owners who do not own a compiler or interpreter, and who wish to learn Pascal before investing in a full-featured Pascal. Read Appendix E before typing in the interpreter to see if it meets your needs.

CHAPTER 1

First Programs

CHAPTER 1
First Programs

Every Pascal program consists of two parts. The first part, the *declarative* section, is where information like the constants and variables used, subroutines used, and so forth, is stated. The second part is the actual program. This division makes a Pascal listing easy to read, understand, and debug. All variables, constants, and subroutines used throughout the program are declared at the beginning.

Program Headers

When you write a program in Pascal, you must start off by stating its name and the external files that will be used (more on files in a later chapter). This program identification is done by using the keyword **PROGRAM**:

```
PROGRAM HELLO(INPUT,OUTPUT);
```

This program header tells us that the program will be called HELLO and that the external files used by the program are INPUT and OUTPUT. Though we will cover this in greater detail later on, basically, INPUT means the computer will be reading information from an external source other than the program itself (such as from the keyboard, entered by the user), and OUTPUT means the program will also be sending information (printing results on the screen). The external files used are always stated in parentheses after the program name. INPUT and OUTPUT are the program parameters. Here is another example header:

```
PROGRAM TEST(OUTPUT);
```

In this example, the program is called TEST, and no external information will be read by the computer (hence, the absence of the INPUT file). Data will be output by the computer; therefore, OUTPUT is stated.

Note the semicolon at the end of each of our example statements. These are called *separator symbols* and are used to separate individual Pascal statements. The semicolon is added at the end of *every* Pascal statement to separate

one statement from the next, except in certain cases which will be discussed later.

Declaring Constants and Variables

Most programs use some values that are constant throughout the program; others use values that do change, that are *variable.* Pascal lets you define what constants and what variables you'll be using in the program, and it gives these values identification symbols. These constant and variable definitions are included in the declarative part of any Pascal listing.

Constants are defined in Pascal following the keyword **CONST**. Consider the following example:

```
PROGRAM DECLARE(INPUT,OUTPUT);
CONST
  A=200;
  NAME='JOHN';
```

The program name in this example is DECLARE, and it will be using standard INPUT and OUTPUT. After the program name, two constants are declared. To mark the beginning of the section of constant definitions is the keyword CONST. In this example, A is defined as 200, and NAME as JOHN. There are several things to note about constant declarations:

- To define the value of a constant, use an equal sign (=).
- Constants cannot be used in the same way as variables during program execution. A statement such as A = A + 1 *or* NAME = 'FRED' are not allowed if A and NAME have been defined as constants.
- Constants are used when you want to use the same string or number repeatedly throughout a program, and you want to represent it by a constant name. Using a constant rather than the actual string or value can sometimes make a program clearer and more legible.
- String constants are enclosed within single quotation marks ('). Double quotation marks (") are not used.
- Each definition must end with a semicolon.
- The keyword CONST does not have a semicolon after it. This is because the semicolon after the definition of the constant A is used for the CONST keyword, too. The above declaration could be written as

```
CONST A=200;
  NAME='JOHN';
```

Even so, the first example is preferred because it makes reading the program listing easier. Also note the way in which the constant definitions have

been written, leaving a greater margin than the CONST keyword. This indention is added for greater reading clarity (though it isn't necessary). As programs become more complex, you'll see the difference it makes.

If you want to include a single quotation mark within a constant text assignment, you must include two single quotation marks; do not use a double quotation mark. For example, if the defined string constant is

```
A='JOHN''S BOOK'
```

and this constant is printed, the result on the screen will be

```
JOHN'S BOOK
```

All variable assignments beginning with the keyword CONST up to the next declarative section—or the beginning of the program itself—are defined as constants in the program.

VARiable Declarations

Variables are defined after the keyword **VAR**. While CONST is used to make permanent assignments, VAR is used to reserve memory for variables.

A variable can belong to one of the following four basic types: INTEGER, REAL, BOOLEAN, or CHAR (there are actually more than these four; they'll be discussed later).

If a variable is defined as an **INTEGER**, it can be used only to store integer values (numbers without decimals).

The **REAL** variables, as the name implies, are used to store real numbers (numbers with decimals). If a REAL variable is assigned an integer value, the integer is treated as a real number. In other words, 79 would be stored as a REAL variable: 79.0. However, if an integer variable is assigned a real value, an error occurs. For example, if i is an integer variable, the statement $i = 1.5$ is illegal because Pascal doesn't know what to do with the .5.

A variable defined as **BOOLEAN** can have only one of two values: TRUE or FALSE.

A variable defined as **CHAR** is used to store single characters.

We'll discuss variable types in greater detail later in the book. Here's an example of variable declarations in a program:

```
VAR
  A:INTEGER;
  LETTER,SYMBOL:CHAR;
  NUMBER:REAL;
  CODE,ID:INTEGER;
  ANSWER:BOOLEAN;
```

Here, the keyword VAR indicates the start of the variable declarations in the program. A is declared as an integer variable; CODE and ID are also integer variables; NUMBER is a real variable; LETTER and SYMBOL are character variables; and ANSWER is a Boolean variable.

A *colon* is used to assign the variable type, and not the equal sign used for constants. Also, if you want to assign several different variables the same variable type, you can just separate the variables by commas.

The VAR section is finished when the next part of the program is found. Variables declared in the VAR section are used in the same way as BASIC variables. Modifying their values during the execution of the program is permitted (unlike constants), but mixing certain variable types is not allowed. For example, if a variable is declared as CHAR, you cannot use that variable to store anything other than characters.

Constant and Variable Identifiers

Most standard Pascal interpreters and compilers differentiate only the first eight characters of constant and variable identifiers. In other words, these two identifiers are different because the eighth characters in the names are different:

```
FIRSTVA1 and FIRSTVA2
```

But the identifiers NUMBERVA1 and NUMBERVA2 are considered the same because their first eight characters are the same. (Check your manual to see how many significant characters your implementation of Pascal allows.)

Also, remember that—similar to variables in BASIC—a variable or constant name can be defined with any set of alphanumeric characters, as long as the first character is alphabetic and the name chosen is not a reserved Pascal keyword.

These are valid identifiers:

```
NAME1       REPLY1       EXAMPLEINPUT       Q1X
```

But these identifiers are not valid:

1NUM	Names must start with a character
ANSWER.2	A punctuation mark in a name is not permitted
TEXT WORD	Spaces may not be embedded in identifiers

Note: Some versions of Pascal allow the underline character (_) to be used in identifiers to improve readability; again check your manual.

Writing the Main Program

Once all the declarations in the program have been made, the actual program is written. The main program always starts with the keyword **BEGIN** and ends

with the keyword **END** followed by a period. END followed by a period *must* be the last statement found in a Pascal program. The period is necessary to distinguish this use of the keyword END from uses of END without a period in other parts of the program (for example, to mark the end of compound statements).

The keyword BEGIN, which states the beginning of the Pascal program, does not have a semicolon after it.

In summary, a simple Pascal program is structured like this:

```
Program Header
Constant Declarations
Variable Declarations
BEGIN
  Main Program
END.
```

Assigning a Value to a Variable

As stated above, constant values are assigned in the declarative part of the program. Variables, on the other hand, only have their types declared, but no actual values are defined. This means that the program will need to assign values to some or all of the variables you have previously declared. In BASIC, you might write A = 1 or C = B + A. In Pascal, the symbol := (a colon followed by an equal sign) is used to assign values to variables. Thus, assuming that the variables have been correctly declared, you might write

```
A:=1
```

or

```
C:=B+A
```

Aside from the different symbols used, these Pascal assignments operate in the same manner as in BASIC. In the first example, the value 1 is assigned to the variable A, and in the second, the value obtained by adding B and A is stored in C. *You must not leave spaces between the colon and the equal sign.* This statement is incorrect: C: =B+A.

Characters are assigned to character variables in the same way:

```
LETTER:='P'
```

Note that character variables are not the same as string variables in BASIC; only one character can be assigned to the variable, not a string of characters. This will be discussed further in Chapters 3 and 7.

Printing on the Screen

Before we look at a complete example program, let's see how to print text on the screen. Two similar statements can be used: **WRITE** and **WRITELN**.

The WRITE statement prints text, constant, or variable values on the screen (or other output files, as you'll later see) and assumes that the next position to be written on the screen is the one immediately following the text printed. This is similar to the BASIC PRINT statement followed by a semicolon. The following two statements produce the same result:

BASIC: `PRINT"HELLO";`

Pascal: `WRITE('HELLO');`

The WRITE and WRITELN instructions require that the text or variables to be printed are enclosed in parentheses. Note that when you want to print a string on the screen, you enclose it in single quotation marks and include the text in parentheses next to the WRITE statement.

If you want to print something on the screen and have the cursor proceed to the next line, as a normal PRINT does in BASIC, use WRITELN. The following two statements obtain the same results on the screen:

BASIC: `PRINT"HELLO"`

Pascal: `WRITELN('HELLO');`

Here are examples and explanations of the WRITE and WRITELN statements:

```
WRITE('PASCAL FOR BEGINNERS');
WRITELN(A+B);
WRITELN;
```

The first example prints the text *PASCAL FOR BEGINNERS* on the screen. A WRITE executed after that would cause the text to be printed after the word *BEGINNERS,* because WRITE was used instead of WRITELN.

The next example prints the value obtained by adding the values of A and B; it causes the next print position to be on the line following that where A+B is printed.

The third example simply prints a blank line. It's the same as a single PRINT would be in BASIC.

The WRITE and WRITELN statements can also print several different items in one statement. You could print text, the value of a variable, and then another text item with just one WRITELN statement. In BASIC, this is done by using semicolons:

```
PRINT "THE ANSWER IS:";A;" METERS."
```

In Pascal, each string, variable, or constant printed is separated by a comma, so you would write

```
WRITELN('THE ANSWER IS: ',A,' METERS.');
```

The way the comma separates elements to be printed varies according to the Pascal implementation being used. Usually, no space—or only one blank space—is added between printed items; the comma does not separate elements in columns as it does in BASIC (the interpreter included with this book does not leave any blank spaces between items when these are separated by commas in the WRITE or WRITELN statements).

Writing a Program

Below is the first complete Pascal program. It uses most of what has been discussed in this chapter and introduces some new things.

Read through the listing first; try to understand as much as possible of the program before you read the explanations. You'll see how the declarations and meaningful identifiers make understanding the program easy.

In this example, the program will assign values to constants and variables, perform certain mathematical operations, and then print the results on the screen.

```
PROGRAM NUMBERS(OUTPUT);
CONST
  A=25;
  TEXT='MATHEMATICAL OPERATIONS';

VAR
  NUM,OP1,OP2,RESULT:INTEGER;
  SQROOT:REAL;
BEGIN
  OP1:=A;
  OP2:=A-15;
  RESULT:=OP1*OP2;
  NUM:=10;
  SQROOT:=SQRT(NUM);
  WRITELN(TEXT);
  WRITELN;
  WRITE(OP1,'*',OP2,'=');
  WRITELN(RESULT);
  WRITELN('THE SQUARE OF ',NUM,' IS ',SQR(NUM));
  WRITELN('THE SQUARE ROOT OF ',NUM,' IS ',SQROOT)
END.
```

The program header gives the program name and says that no information will be input by the program, but that information will be output.

Two constants are declared: A is 25 and TEXT is equal to MATHEMATICAL OPERATIONS. This means that whenever TEXT is printed, the phrase will be printed instead; whenever A is used, the value 25 will be substituted.

Variables are declared after the constant declarations; NUM, OP1, OP2, and RESULT will be used to store integer numbers, and SQROOT will be used for real numbers.

The actual program starts with the keyword BEGIN. Note that following BEGIN, each instruction is indented. This is a good programming practice which will be used throughout this book. Indention doesn't mean anything to the compiler or interpreter, but it helps make the program easier for you, the programmer, to read.

The program is very simple. To start, the variable OP1 is made equal to the value of the constant A (25), and OP2 is made equal to A−15 (10).

Next, the product of OP1 and OP2 is calculated, using the multiplication sign—an asterisk (*) as in BASIC. The result is stored in the variable RESULT.

Then NUM is made equal to 10, and the program calculates the square root of NUM using the **SQRT** routine (discussed in the next chapter). This result is stored in the variable SQROOT.

Once all these calculations have been made, the program prints five lines: the text constant defined at the beginning of the program, a blank line, and three lines with the answers of the operations.

The program prints OP1, the multiplication sign, OP2, and the equal sign. This is done by using the WRITE statement because, when the answer (RESULT) is printed, we want it printed beside the equal sign. The same method is used to print the square and square root of NUM. The program then ends.

To print the square of NUM, you must include the operation (**SQR**) directly in the WRITELN statement.

Notice that the statement before the END. instruction does not have an ending semicolon. This is because the statement prior to any END keyword doesn't need a separator symbol.

The results of this very simple program will look like this on the screen:

```
MATHEMATICAL OPERATIONS
25*10=250
THE SQUARE OF 10 IS 100
THE SQUARE ROOT OF 10 IS 3.16228000E+00
```

Note that real numbers are expressed in scientific notation. You can avoid this in several ways, which will be discussed in the following chapter.

CHAPTER 2

Integer and Real Variables

CHAPTER 2
Integer and Real Variables

Two important variable types, introduced in the previous chapter, are INTEGER and REAL. INTEGER type variables are used to store whole numbers. REAL variables are used to store real numbers, numbers which may contain decimals.

Integer Numbers

Integer numbers are stored in variables declared as INTEGER. They are often used as counters in program loops, to count the repetitions of any certain event, or simply when integer math is preferred (when you want to avoid the decimal parts of numbers). Integer values may not contain embedded blanks or commas.

Although they vary from computer to computer, the numbers which can be stored in an INTEGER variable usually range from −32767 to 32767. On most Pascal systems, the constant MAXINT is defined as the maximum integer value which can be stored in an integer variable. The following statement will print that value:

```
WRITELN(MAXINT)
```

In other words, integer values must always be in the range of −(MAXINT+1) to MAXINT.

Try this short program to see how MAXINT works:

```
PROGRAM MAXEXAMPLE(OUTPUT);
CONST
  TEXT='MAXIMUM INTEGER IS  ';
VAR
  NUM:INTEGER;
BEGIN
  WRITELN(TEXT,MAXINT);
  NUM:=-(MAXINT);
  WRITELN('NEGATIVE  ',TEXT,NUM)
END.
```

A result which is not in the specified range will cause an overflow error. Try adding this line just after `NUM:=-MAXINT;` :

```
NUM:=MAXINT+1;
```

On most systems this line will produce an out-of-range error (the exact error message will vary).

Negative integer numbers are preceded by a minus (−) sign. The plus sign can be omitted from positive numbers.

Integer Operations

Here are the five basic operations used in integer mathematics:

+	Addition
−	Subtraction
*	Multiplication
DIV	Integer division
MOD	Remainder in integer division

The addition, subtraction, and multiplication operations work just as they do in BASIC, always dealing with integer numbers as operands. With integer division, the result will be calculated without decimal values. For example,

```
7 DIV 3 = 2
13 DIV 3 = 4
```

The second value must not be negative when DIV is used.

Use the MOD function to find the remainder of integer division:

```
13 MOD 3 = 1
```

Other functions which can be applied to integer arithmetic include:

ABS	Returns the absolute value (positive value) of a number
SQR	Returns the square of a given number
ODD	Returns either TRUE if the number is odd or FALSE if the number is even
PRED	Returns the previous integer number of a given integer value
SUCC	Returns the following integer number of a given integer value

Here are some examples:

```
ABS(-7)    = 7
ABS(7)     = 7
SQR(3)     = 9
SQR(12)    = 144
SQR(-7)    = 49
ODD(3)     = TRUE
ODD(8)     = FALSE
SUCC(2)    = 3
SUCC(-3)   = -2
PRED(97)   = 96
PRED(-4)   = -5
```

The functions PRED(−MAXINT) and SUCC(MAXINT) are undefined and cause execution errors.

The following program calculates the square of a given integer number:

```
PROGRAM SQUARES(OUTPUT);
VAR
  X:INTEGER;
BEGIN
  X:=3;
  WRITELN('THE SQUARE OF ',X,' IS ',SQR(X))
END.
```

The result of this program would be

```
THE SQUARE OF 3 IS 9
```

A Short Quiz

For practice, fill in the output that will appear on the screen as a result of each of these statements. The results can be found later in this chapter.

	Statement	Result
1.	WRITELN(ODD(3))	_____
2.	WRITELN(ODD(4−2))	_____
3.	WRITELN(ABS(−794))	_____
4.	WRITELN(SQR(2))	_____
5.	WRITELN(5 DIV 2)	_____
6.	WRITELN(5 MOD 2)	_____

Combining Integers and Real Numbers

If a constant is defined as an integer number, it can be combined with integer operations. For example,

```
CONST
  VALUE=10;
VAR
  NUM1,NUM2:INTEGER;
```

The following operations are valid:

```
NUM1+VALUE
(NUM1-NUM2) DIV VALUE
NUM1 MOD NUM2 - 3
```

Operations are evaluated in the same order as in BASIC. Multiplication, division, and remainder operations are calculated before addition and subtraction. You can alter this evaluating order by using parentheses.

Another symbol for division is the slash (/). The slash is used for the division of real numbers, as in BASIC, but it can also be used with integer numbers. Whenever the slash is used, whether with an integer or a real number, the result will always be considered a real number by the computer. In other words,

5 DIV 2 = 2 **5/2 = 2.5**
1 DIV 2 = 0 **1/2 = 0.5**
4 DIV 2 = 2 **4/2 = 2.0**

An integer operation can be assigned to a real variable. If you make these assignments:

```
VAR
  RESULT:REAL;
  NUM1,NUM2:INTEGER
```

the following operation is valid:

```
RESULT:=NUM1*NUM2
```

Pascal changes the integer result of the operation (NUM1*NUM2) into a real number and then assigns this value to the variable RESULT.

Other functions which can be used with integer values, but which produce real values as results, are LN, EXP, SQRT, SIN, COS, and ARCTAN. Each of these functions is described below.

Results of quiz: (1) TRUE (2) FALSE (3) 794 (4) 4 (5) 2 (6) 1

Real Numbers

Real numbers are used in the same way as numbers are used in BASIC (and as real numbers are used in mathematics). Real numbers are represented by the computer in scientific notation. To indicate scientific notation, the computer uses an E followed by a positive or negative integer number. The E should be read as "times 10 to the power of...."

Depending on the computer used, the maximum and minimum real numbers which can be represented will vary—as will the way the real numbers are displayed by default on the screen. Generally, the maximum real number that can be represented ranges from aproximately 10 to the power of 30 on some computers to 10 to the power of 70 on others. The number of decimals used also varies according to the computer and Pascal implementation used. Care must be taken because of this. In certain real operations, there might be a considerable loss of precision.

Real numbers may not contain any embedded blanks. For example,

1.3 = 1.3*10⁰ = 1.30000E+00

0.00791 = 7.91*10⁻³ = 7.91000E−03

273.00139 = 2.73001E+02

Some precision is lost when numbers have several decimal digits. In the last example, the digits 3 and 9 are lost. Also, as with integer numbers, negative numbers are preceded by a minus sign. The computer always represents real numbers in the same way and in the same number of positions. In the first two examples, the numbers are completed with zeros so that every number has six digits plus the exponent value. You can change this by stating a field size. (This is discussed later in the chapter.)

These are the basic operator symbols used in real arithmetic:

+	Addition
−	Subtraction
*	Multiplication
/	Division

When integer numbers are used in real arithmetic, they are converted to real numbers whenever a real number is involved in the operation. If RES is a real variable, and you write RES:=3, the computer will make the following assignment: RES:=3.0.

In this example,

```
(7 + 3) / (5.1 - 5)
```

the computer does the following evaluation:

```
7 + 3 = 10   (integer result)
5.1 - 5 = 5.1 - 5.0 = 0.1   (real result)
```

It then calculates the real result:

```
10 / 0.1 = 10.0 / 0.1 = 100.0
```

If a real number is used in an arithmetic operation, the result is always treated as a real number, even if the number can be expressed as an integer value. When the / operator is used to perform a division, the result is always a real number, even if both operands are integer values.

Trying to assign a real result to an integer variable produces an error condition.

Math Functions

The following list of functions can be used when working with real numbers (or integer values with real results):

SIN(X)	Calculates the sine of X, where X is an angle expressed in radians
COS(X)	Calculates the cosine of X, where X is an angle expressed in radians
ARCTAN(X)	Calculates the arctangent, expressed in radians, of X
LN(X)	Returns the logarithm of X in base *e*
SQR(X)	Returns the square of the number X
SQRT(X)	Returns the square root of the number X
EXP(X)	Calculates the value of *e* to the power of X
TRUNC(X)	Returns the integer part of X
ROUND(X)	Rounds a number X to the closest integer number

The functions TRUNC and ROUND are used to convert REAL numbers into INTEGER numbers. There is an important difference between the functions TRUNC and ROUND. TRUNC finds the corresponding integer number by ignoring the fractional part of the number. ROUND approximates the number to the nearest integer by rounding the number. If the converted real number cannot be represented in integer format, an error occurs.

Here are some examples:

```
TRUNC(2.3) = 2
ROUND(2.3) = 2
TRUNC(3.719) = 3
ROUND(3.719) = 4
TRUNC(-23.6) = -23
ROUND(-23.6) = -24
```

This program illustrates the use of TRUNC and ROUND. Try changing the value of Y to see different results.

```
PROGRAM ROUNDING(OUTPUT);
VAR
  X:REAL
  Y:REAL
BEGIN
  Y:=5.8;
  X:=TRUNC(Y);
  WRITELN('TRUNC(',Y,') = ',X);
  X:=ROUND(Y);
  WRITELN('ROUND(',Y,') = ',X)
END.
```

Formatting the Output

When you're printing integer and real numbers with the WRITE and WRITELN statements, you can decide how you want them printed. Do this by declaring the field size of the output. To declare the field size, put a colon after the value to be printed and either an integer value, constant, or variable identifier, which represents the size of the field. This statement reserves eight positions in which to print the value of NUM:

```
WRITELN(NUM:8);
```

If the value to be printed has more digits than the positions stated in the field, the field declaration is ignored and the number will be printed anyway. If the number has fewer digits than the specified field, the number is preceded by blanks; this aligns the number to the right side of the specified field. This is useful in formatting the output of a list of integer numbers. The default field size varies according to the Pascal implementation.

Try this simple example. Test the program with different values of X:

```
PROGRAM INVENTORY(OUTPUT);
VAR
  X:INTEGER;
  Y:INTEGER;
BEGIN
  X:=10;
  WRITELN('INVENTORY LIST':2*X);
  WRITELN;
  WRITE('PRODUCT':X);
  WRITELN('NUMBER':X+1);
  WRITE('PAPER':X);
  Y:=250;
  WRITELN(Y:X);
  WRITE('PENCILS':X);
  Y:=250;
  WRITELN(Y:X);
  WRITE('PENS':X);
  Y:=250;
  WRITELN(Y:X)
END.
```

You can also print real numbers by using the format option. Specifying the size of field for real numbers is accomplished in the same manner as for integers: Follow the value or variable to be printed with a colon and an integer number, constant, or variable. The value is printed in scientific notation in the specified field. But you can also specify how many decimal digits you wish to print by following the field size with another colon and another integer value or identifier. For example,

```
A:=10;
WRITELN(VALUE:A:3)
```

This statement will print the real number VALUE in a field with ten positions, leaving three digits after the decimal point. Doing this also avoids the real results printed in scientific notation—useful in many applications. If VALUE is equal to 2.34, the statement will produce two blank spaces followed by 2.340.

In the statement

```
WRITELN(2.34:6:3)
```

the computer will print the number 2.34 in a field size 6, leaving three decimal digits. Since the number is only three digits, two blanks are added before the number (on some systems, a zero is added to the decimal part of the number to obtain the three decimal places). Here's another example:

```
WRITELN(2134.5393:10:1);
```

The output here would be

```
    2134.5
```

In this example, there are five spaces and only one decimal value printed, and the number is printed in a field size 10.

(Some compilers and interpreters count the decimal point as one position in the field. Also, the way real numbers are displayed on the screen varies according to the computer and the Pascal implementation used.)

If the field specified is not large enough to print the number, the field specifications are ignored (but the decimal part specifications are still taken into account).

The next section includes an example program which illustrates some of the concepts discussed in this chapter.

Area and Perimeter of a Circle

The following program calculates the area and perimeter of a circle. The radius must be defined as a constant in the declaration section. The results are first printed as they are output by the computer by default; then they are printed a second time in fields specified by the program.

```
PROGRAM CIRCLE(OUTPUT);
CONST
  RADIUS=10;
  PI=3.14159;
VAR
  AREA,PERIMETER:REAL;
BEGIN
  AREA:=PI*SQR(RADIUS);
  PERIMETER:=2*PI*RADIUS;
  WRITELN(RADIUS,AREA,PERIMETER);
  WRITELN(RADIUS,AREA:10:1,PERIMETER:10:1)
END.
```

When you run the program, the following two sets of values will be printed:

```
10 3.14159E+02 6.28318E+01
10       314.2          62.8
```

(The exact output of these lines will vary depending on the computer and the Pascal version.)

The first three numbers are printed without any specified fields. The first

number is integer, so it is printed in normal notation (to declare the constant RADIUS as real, you would have to write RADIUS=10.0). The other two numbers are real results and are printed in scientific notation. The second set of numbers is printed with specified fields for the real results. The results are printed in field size 10, with only one decimal value.

Notice that you can add multiple items in a WRITELN and WRITE statement even if you include field delimiters.

CHAPTER 3

More About Variables

CHAPTER 3
More About Variables

So far, all the example programs have used values assigned to variables in the program itself. Most application programs require the user to enter information while the program is running. Pascal has two statements which read data from an external device: READ and READLN.

Reading Values in a Program

The **READ** and **READLN** statements are used to enter values. The effects of READ and READLN vary depending on the type of variable. READ and READLN are different in the same way that WRITE and WRITELN are different. After reading the value entered, READ causes the position of the next read or write to be immediately to the right of whatever was just read. READLN, on the other hand, causes the next read or write to be executed on the following line.

When a READ or READLN statement is encountered, program execution stops and waits for an input from the user (ended by pressing RETURN). In contrast to BASIC, no prompt is printed. If you want a prompt, you'll have to print it before the read is executed.

Here are some examples of the READ and READLN statements:

```
READ(X);
READLN(X);
```

In both cases, the value read is assigned to the variable X. The value or expression entered by the user must coincide with the variable type for X that was defined in the first part of the program, or, in most cases, an error will occur. You can also read several values together, with one READ statement, separating the variables inside the READ parentheses by commas:

```
READ(X,NAME,ANS);
READLN(NUM,OP);
```

If the variable in the READ statement is declared as integer, and the value entered is real, an error occurs, just as when you assign an incorrect value to an integer variable. If, on the other hand, the number entered is integer and the variable real, the integer value is changed into a real value and assigned to the variable.

When reading integer or real numbers, READ and READLN work in the same way that INPUT works in BASIC. The user types in a number, which is assigned to the corresponding variable when he or she presses RETURN. The computer will then read the value keyed in and assign it to the variable. Blank spaces typed before the number are ignored. If two or more numbers are required by the READ statement, you can either press RETURN after each number entered or write them one after the other, separating them with one or more blank spaces. The computer will assign each number to its corresponding variable.

You cannot read the value of Boolean variables. READ and READLN can only be used for integer, real, or character variables. Even so, it's easy to write a routine that reads a character variable (T or F for TRUE or FALSE) and changes that value to its corresponding Boolean value. An example of this is shown at the end of Chapter 5.

Be careful when you read a character variable—only one character will be read and assigned by the READ statement because character variables may have only one character assigned. For example, if the computer encounters the following READLN statement in a program:

```
VAR
  X:CHAR;

BEGIN
  READLN(X);
```

and you type

```
HELLO <RETURN>
```

the variable X will be equal to H.

If the computer executed the statement (assuming X, Y, and Z are declared as CHAR)

```
READ(X,Y,Z);
```

and you type

```
HELLO <RETURN>
```

the computer will make the following assignments:

```
X:='H'      Y:='E'      Z:='L'
```

In other words, Pascal does not have string variables; you cannot read and assign a complete string to a variable directly (this is done with a loop; see Chapter 7).

READ and READLN do not skip blank spaces that may be found before the character entered by the user, as they do when reading integer and real numbers. Blank spaces are read as characters.

Here's a sample program that uses READ and READLN:

```
PROGRAM BIRTHDAY(INPUT,OUTPUT);
VAR
  X,Y:INTEGER;
BEGIN
  WRITE('WHAT YEAR IS IT?  ');
  READ(Y);
  WRITELN('  THANKS');
  WRITE('ENTER YOUR AGE');
  WRITE('  NO FRACTIONS PLEASE  ');
  READLN(X);
  WRITELN('THANKS');
  WRITELN('IF IT IS PAST YOUR BIRTHDAY THIS YEAR');
  WRITELN('THEN YOU WERE BORN IN  ',Y-X);
  WRITELN('OTHERWISE YOU WERE BORN IN  ',Y-X-1)
END.
```

Boolean Variables and Operators

As mentioned earlier, a constant or variable declared as BOOLEAN may either be TRUE or FALSE and no other value. Therefore, Boolean variables are most useful in conditional loops (see Chapter 4) and in IF-THEN statements where certain conditions must be met.

Just as in BASIC, in Pascal you can use the Boolean operators AND, OR, and NOT when executing comparisons or assignments. AND results in TRUE only if both operators are TRUE. OR results in TRUE if any (or both) operators are TRUE. NOT changes the current state (TRUE or FALSE) of a variable to the opposite value. For instance, if

```
OPER1:=TRUE
OPER2:=FALSE
```

the following operations would have these results:

```
OPER1 AND OPER2  = FALSE
OPER1 OR OPER2   = TRUE
NOT(OPER1)       = FALSE
```

In the first example, OPER1 is TRUE, but OPER2 is FALSE. The AND operator implies that both operators must be TRUE for the result of the operation to be TRUE. OPER2 is FALSE, so the result of the AND operation is FALSE.

In the second example, the OR operator means that the result of the operation will be TRUE if either of the operators is TRUE. OPER1 is TRUE, so the result is TRUE.

The NOT operator simply changes the current state of the variable. If the variable is TRUE, NOT makes it false. If it is FALSE, NOT makes it TRUE.

Boolean operators can be used with other types of variables. Consider the following example:

```
PROGRAM BOOL(INPUT,OUTPUT);
VAR
  RESULT,OPER1,OPER2:BOOLEAN;
  A:INTEGER;
BEGIN
  OPER1:=TRUE;
  OPER2:=FALSE;
  A:=77;
  RESULT:= (A=77) AND (OPER1) AND NOT(OPER2);
  WRITELN('THE RESULT IS   ',RESULT)
END.
```

This program shows how Boolean operators are used together with Boolean and integer variables to produce one result. The program defines OPER1 as TRUE and OPER2 as FALSE. Variable A is assigned the number 77. The condition (TRUE or FALSE) of RESULT is then found. The condition A=77 is TRUE and OPER1 is TRUE. OPER2 is FALSE, but when NOT is applied to it, it changes to TRUE. In other words, the assignment made to RESULT is

RESULT:=TRUE AND TRUE AND TRUE

Thus, RESULT is TRUE. WRITELN is used to print the value of a Boolean operator.

Comparisons are made with the relational operators; they are the same as in BASIC:

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

Be careful with assignments such as

```
RESULT:= (N=2) AND (S>8);
```

If the parentheses were not used in this example, an error would occur.

Character Variables

Character variables are defined in a program by using the keyword **CHAR**. Once a variable has been declared a CHAR variable, you can assign it a value by enclosing a character within single quotation marks:

```
LETTER:='X'
```

Characters can be compared by using the relational operators. Their ASCII codes (or corresponding code, according to the computer used) are used for the comparison. In other words,

'B' < 'C'
'J' > 'D'
'3' > '1'

In all Pascal implementations, the digits 0–9 are in order and contiguous. The letters *A*–Z are also in order, but are not necessarily contiguous, according to the character code set used.

Two functions that can be applied to character variables are **ORD** and **CHR**. ORD returns the ASCII code of a character, just as ASC does in BASIC. If your computer doesn't use ASCII codes for the characters, it will return whatever codes the computer uses. As most computers use ASCII, we'll use ASCII in the next examples.

Suppose you write

```
WRITELN(ORD('1'));
```

The result printed will be 49, the ASCII code of 1.

If you wanted to change a numeric digit stored as a character in a character variable to its equivalent decimal digit (stored in NUM), you would do the following:

```
X:=ORD(NUM)-ORD('0');
```

Or, directly for the ASCII character set:

```
X:=ORD(NUM)-48
```

That is, you subtract the character code of zero from the character code of your number to arrive at its equivalent decimal digit.

The CHR function has the opposite effect; it's similar to the BASIC function CHR$.

For example, if you write

```
X:=42;
WRITELN(CHR(X));
```

the result will be a printed asterisk (ASCII code 42).

The expressions PRED(CHR(0)) and SUCC(CHR(255)) are not defined. Also, using the expression CHR(X), where X is smaller than 0 or larger than 255, produces an error.

Example Programs

```
PROGRAM LETNUM(INPUT,OUTPUT);
VAR
  NUMBER:REAL;
  LETTER:CHAR;
  POSIT:INTEGER;
BEGIN
  WRITE('ENTER ANY POSITIVE NUMBER:');
  READLN(NUMBER);
  WRITE('ENTER ANY LETTER:');
  READLN(LETTER);
  WRITELN;
  WRITE('THE LETTER ENTERED IS NUMBER  ');
  POSIT:=ORD(LETTER)-ORD('A')+1;
  WRITELN(POSIT,'  IN THE ALPHABET');
  WRITELN;
  WRITE('THE SQUARE ROOT OF  ',NUMBER,'  IS  ');
  WRITELN(SQRT(NUMBER))
END.
```

LETNUM uses no constants and three variables: NUMBER (REAL), LETTER (CHAR), and POSIT (INTEGER). First, the user is asked to input two values: any positive number and a letter. Notice how WRITE, instead of WRITELN, is used for the prompts so that, when the user types in the answer, it will be printed to the right of the prompt instead of beneath it.

Once the user has typed in the number and letter, the program finds the

position occupied by the letter in the alphabet (by subtracting the ASCII code of the letter *A* from the ASCII code of the entered letter and then adding one). This result is then stored in the integer variable POSIT (the position is an integer number) and printed. Finally, the square root of the number that was entered is printed. The program limits the number of decimal places to four.

Note that if you enter a negative number, an error occurs because the square root of a negative number is undefined when working with real numbers. Also, entering a nonalphabetic character causes a mistake in the printed result. These errors can be avoided in several ways, which will be discussed later.

```
PROGRAM BOOL2(OUTPUT);
VAR
  RES1,RES2:BOOLEAN;
  NUM1,NUM2:INTEGER;
BEGIN
  RES1:=TRUE;
  RES2:=FALSE;
  NUM1:=5;
  NUM2:=7;
  WRITELN(NUM1,'=',NUM2,'  IS  ',NUM1=NUM2);
  WRITELN(RES1,'=',RES2,'  IS  ',RES1=RES2);
  WRITELN((5=5) AND (NUM1=(NUM2-2)),NOT(RES2));
  WRITELN(ODD(3),ODD(8))
END.
```

This program demonstrates the results of different relations using Boolean operators.

First, RES1 is assigned TRUE, and RES2, FALSE; NUM1 is assigned 5, and NUM2, 7. Then, different relations are printed out, using WRITELN statements. As NUM1<>NUM2, the first WRITELN statement prints

```
5=7 IS FALSE
```

Since RES1<>RES2, the next WRITELN statement prints

```
TRUE=FALSE IS FALSE
```

And since 5 = 5, 5 = 7 − 2, and NOT(FALSE) is TRUE, the third WRITELN prints

```
TRUE TRUE
```

Finally, because 3 is odd and 8 is not, the last WRITELN prints

```
TRUE FALSE
```

Be careful to differentiate between the symbols = and := . The first is used when you compare values, like IF A=B THEN..., which gives a TRUE or FALSE Boolean result. The second is used for assignments like A:=A+1. In Boolean expressions, both symbols might be used. For example,

```
A:=B AND (C=3);
```

which assigns A the result of B AND (C=3).

CHAPTER 4

Program Loops

CHAPTER 4
Program Loops

For you to be able to write more complex and useful Pascal programs, two very important types of structures are still missing: loops and conditional branches. Loops are discussed in this chapter, and conditional branches will be treated in Chapter 5. Once you understand how to use loops and branches effectively, you will be able to write more sophiscated Pascal programs.

While BASIC limits you mainly to FOR-NEXT loops when repeating program sections, Pascal offers a variety of loop structures that you can use to repeat a statement or group of statements a certain number of times.

Pascal includes loops that are similar to the BASIC FOR-NEXT loop, loops that execute *while* a certain condition is met, and loops that execute *until* a certain condition is met. In this chapter, three different types of Pascal loops will be explained: **WHILE-DO**, **REPEAT-UNTIL**, and **FOR-DO** loops.

Using WHILE-DO

The WHILE loop is a very useful Pascal structure. As long as the condition following the WHILE statement is TRUE, the statement following WHILE will be executed. In a WHILE loop, the condition is evaluated before the loop itself. The general structure of a WHILE loop is

WHILE *condition* **DO**
  statement

In other words, these are the steps followed in the evaluation of a WHILE loop:

1. Evaluate the condition in the WHILE statement. If it is FALSE, jump to step 4 and continue program execution.
2. Execute the program instructions in the loop (the condition for the loop was found to be TRUE).
3. Return to step 1.
4. Loop is finished. Continue with program execution.

Consider the following example:

```
WHILE X<>'*' DO
  READ(X);
```

This statement reads characters from the keyboard (using READ) until an asterisk is entered.

In the previous example, only one statement is executed while the condition is true. This is useful in certain cases, like the one above, where characters are being read and certain characters have to be ignored. But in most cases, you'll want to execute a set of instructions while the condition is met. This can easily be done, as long as the instructions to be executed in the WHILE loop are enclosed by the keywords BEGIN and END in the following way:

WHILE ***condition*** **DO**
  BEGIN
  ... program statements
  END

Here's an example:

```
PROGRAM WHILELOOP1(OUTPUT);
VAR
  A:INTEGER;

BEGIN
  A:=0;
  WHILE A<20 DO
    BEGIN
    A:=A+1;
    WRITELN(A,'   HELLO')
    END;
  WRITELN('BYE')
END.
```

In this example, the word HELLO will be printed 20 times on the screen because the variable A will vary from 1 to 20. Note where semicolons are not necessary.

Let's see another example:

```
PROGRAM WHILELOOP2(OUTPUT);
VAR
  X:INTEGER;
  COND:BOOLEAN;

BEGIN
  X:=10;
  COND:=TRUE;
  WHILE (X<20) AND (COND) DO
    BEGIN
    WRITELN(X,COND);
```

```
    X:=X+1;
    END;
  WRITELN('X IS NOW EQUAL TO   ',X)
END.
```

This example uses two conditions to determine whether the loop will be executed or not. Since the program uses the Boolean operator AND, both conditions must be TRUE for the loop to be executed. Thus, X must be less than 20, and COND must be TRUE. Though you can do so if you like, it's not necessary to ask for the validity of COND with COND=TRUE. It's assumed that this is the case by just stating ...AND COND. If X<20 and COND is TRUE, you can interpret the condition stated as WHILE TRUE AND TRUE DO....

Note that in both examples, the WHILE statement does not end with a semicolon. A semicolon is not necessary because, even if BEGIN is on the next line, BEGIN is implied at the end of the WHILE line. The WHILE statement above could be written as

```
WHILE (X<20) AND COND DO BEGIN
```

REPEAT–UNTIL

Although REPEAT–UNTIL loops are not used as frequently as WHILE-DO loops, they are handy in certain circumstances. This loop repeats all instructions enclosed within the keywords **REPEAT** and **UNTIL,** until a stated condition is met.

The loop starts with the keyword REPEAT and finishes with the keyword UNTIL, where the condition that determines repetition is stated. Different from the WHILE loop, the repetition condition is evaluated at the end of the loop. The loop is always executed at least once. Since the keywords REPEAT and UNTIL enclose the statements to be repeated, you don't enclose them with BEGIN and END as you do with the WHILE loop. The general syntax for a REPEAT loop is

REPEAT;
...(program statements)
UNTIL (*condition*)

The primary difference between this type of loop and the WHILE loop is that the statements in a REPEAT loop are always executed at least once. When you're using a WHILE loop, it's possible that the statements in the loop might not even be executed once.

Here are the steps followed in a REPEAT loop:

1. Execute the instructions found between the keywords REPEAT and UNTIL.
2. Evaluate the repetition condition (at the UNTIL statement). If it's FALSE, return to step 1.
3. The condition is TRUE. Leave the loop and continue program execution.

This is an example of a REPEAT–UNTIL loop:

```
PROGRAM REPEATUNTIL(OUTPUT);
VAR
  A:INTEGER;
BEGIN
  A:=0;
  WRITELN('THE FIRST TEN NUMBERS ARE:');
  REPEAT
    WRITELN(A);
    A:=A+1
  UNTIL A>9;
END.
```

This program, using a REPEAT loop, prints the numbers 0–9 on the screen.

The REPEAT and UNTIL keywords act like the BEGIN and END keywords in other cases. Therefore, a semicolon is not used after the keyword REPEAT, nor is one used with the statement prior to the keyword UNTIL. Note that, as with BEGIN and END, whatever is enclosed within REPEAT and UNTIL is indented for greater clarity in reading or debugging the program. In this way, it's easy to see what statements are executed in the REPEAT loop.

The WHILE loop is, in most cases, superior to the REPEAT loop because it takes into consideration the fact that the loop may not need to be executed even once during the program. On the other hand, use REPEAT for certain cases when you want the loop to be executed at least once. When in doubt as to which of the two structures to use, try WHILE first.

FOR-DO Loops

This type of loop will be the most familiar to BASIC programmers. It works very much like the FOR-NEXT loop in BASIC. The FOR-DO loop is used when you know how many times you want to repeat a loop—that is, when the number of repetitions of the loop is not determined by the effect of certain instructions within the loop itself. A control variable of any ordinal type (which means that it can be any type which has the SUCC function defined) is used in

the beginning of the loop. The most common control variable is defined as INTEGER. For example,

```
FOR I:=1 TO 20 DO
  WRITELN('EXAMPLE');
```

This statement prints the word EXAMPLE on the screen 20 times. Each time the loop is executed, the computer increases the value of the variable I by one—making the assignment I=SUCC(I)—and the loop is repeated until the variable's value is 20. The first time the variable is equal to 1, the second time to 2, and so on. Note that the := sign is used in the loop, not a simple equal sign. This is because the values 1–20 are being *assigned* to the variable (not compared). The value of the control variable when you leave the loop depends on the Pascal compiler or interpreter that you're using. It's best to consider it undefined and assign a new, specific value to it if the variable is needed later. Also, do not modify the value of the control variable inside the program loop.

Here's another example:

```
PROGRAM FORDO(OUTPUT);
VAR
I,MAXNUM:INTEGER;

BEGIN
  WRITE('ENTER THE NUMBER OF TIMES TO REPEAT   ');
  READLN(MAXNUM);
  FOR I:=1 TO MAXNUM DO
    WRITELN(I)
END.
```

In this example, a number is read and a loop is executed that number of times, printing the value of the control variable in each repetition. If you enter a 1 at the prompt, the loop is repeated once and a 1 is printed. If you enter a number smaller than 1, the loop is not executed at all, just as in BASIC.

The FOR-DO loop can also be used to repeat a number of statements, as long as these are enclosed by the keywords BEGIN and END and use the following general syntax:

```
FOR variable:=val1 TO val2 DO
  BEGIN
  ...program statements
  END
```

For example,

```
PROGRAM SUMPROD(INPUT,OUTPUT);
VAR
  PROD,SUM,NUM,I:INTEGER;
BEGIN
  WRITELN('ENTER A POSITIVE WHOLE NUMBER LESS THAN 8');
  PROD:=1;
  SUM:=0;
  READLN(NUM);
  FOR I:=1 TO NUM DO
    BEGIN
    PROD:=PROD*I;
    SUM:=SUM+I
    END;
  WRITE('THE PRODUCT OF THE NUMBERS BETWEEN');
  WRITELN('  1 AND  ',NUM,'  IS  ',PROD);
  WRITELN('AND THE SUM IS  ',SUM)
END.
```

This program reads a positive integer and calculates the product and sum of the numbers between 1 and the number entered. Because more than one operation has to be executed inside the FOR-DO loop, these statements are enclosed between the BEGIN and END keywords.

Since all the variables are defined as integer, if you enter a number greater than 7, an overflow error occurs because the product of the numbers exceeds the maximum integer value allowed. A number smaller than 1 causes incorrect results. By changing the variable PROD to real and specifying the number of decimal places in the WRITE statement, you can enable the program to handle numbers up to 33 before an overflow error occurs. This is the revised program:

```
PROGRAM SUMPROD(INPUT,OUTPUT);
VAR
  PROD: REAL;
SUM,NUM,I:INTEGER;
BEGIN
  WRITELN('ENTER A POSITIVE WHOLE NUMBER LESS THAN 34');
  PROD:=1.0;
  SUM:=0;
  READLN(NUM);
  FOR I:=1 TO NUM DO
    BEGIN
```

```
    PROD:=PROD*I;
    SUM:=SUM+I
    END;
  WRITE('THE PRODUCT OF THE NUMBERS BETWEEN');
  WRITELN('  1 AND  ',NUM,'  IS  ',PROD:1:1);
  WRITELN('AND THE SUM IS  ',SUM)
END.
```

Pascal's FOR-DO has nothing similar to BASIC's STEP statement to increment the loop in steps other than 1. Pascal does have a substitute for BASIC's STEP −1. To count down from a larger value to a smaller value, use the keyword **DOWNTO**:

```
FOR A:=30 DOWNTO 10 DO...
```

In this example, A will first be 30, then 29, and so forth, down to 10, where the loop will end. When you use DOWNTO, the assignment A:=PRED(I) is used to find the next value of A.

Sample Programs

Finding an average. This first program will ask the user to input some real numbers ending in a zero. The program will then calculate and print the average of these numbers. A WHILE loop will be used in the program to read the numbers from the keyboard.

```
PROGRAM AVERAGE(INPUT,OUTPUT);
VAR
  VALREAD,SUM,AVERAGE:REAL;
  NUM:INTEGER;
BEGIN
  SUM:=0.0;
  NUM:=0;
  WRITELN('PLEASE INPUT REAL NUMBERS');
  WRITELN('ENTER A ZERO WHEN FINISHED');
  WRITELN;
  READ(VALREAD);
  WHILE VALREAD<>0.0 DO
    BEGIN
    SUM:=SUM+VALREAD;
    NUM:=NUM+1;
    READ(VALREAD)
    END; {OF WHILE}
  AVERAGE:=SUM/NUM;
```

```
  WRITE('THE AVERAGE IS   ');
  WRITELN(AVERAGE)
END.
```

This program requires a variable to store each number read, another for the addition of the numbers, and a third for the average. All these are declared as real numbers. Also needed is a variable to keep count of how many numbers have been read: an integer, NUM.

After the declarative part of the program has been written, the main program begins. The first step is to make SUM and NUM equal to zero. Then the prompt is printed. Before the WHILE loop can start, the first input must be read. Next, the WHILE loop starts, checking to see whether the number read (VALREAD) is zero. If it is, execution leaves the loop. If VALREAD is not zero, the number read is added to the total value in SUM, and the total number of values entered is increased by one. A new number must be read inside the loop since, when the loop has been executed, control returns to the WHILE statement which checks the value of VALREAD.

Once all the numbers have been read, the average is found by dividing SUM by NUM, and finally the average is printed.

Take care when you run the program, because no checks are made for incorrect entries. If a character other than a number is entered, the program will stop with an error. You'll also get an error if the first number entered is zero (division by zero is not allowed). You can easily avoid these kinds of errors by using an IF-THEN statement, which is discussed in the next chapter.

Multiplication tables. The next program prints the multiplication table for any given number. It uses a FOR-DO loop to print the table.

```
PROGRAM TABLES1(INPUT,OUTPUT);
CONST
  TEXT='INPUT ANY INTEGER NUMBER (0 TO END):';
VAR
  NUM,CTRL:INTEGER;
BEGIN
  WRITE(TEXT);
  READLN(NUM);
  WHILE NUM<>0 DO
    BEGIN
    FOR CTRL:=1 TO 12 DO
      WRITELN(NUM,'  *  ',CTRL,'  =  ',NUM*CTRL);
    WRITE(TEXT);
    READLN(NUM)
```

```
    END;
  WRITELN('PROGRAM EXECUTION FINISHED.')
END.
```

The multiplication table for any number you enter is printed (as long as it doesn't cause an overflow). Enter zero to end the program.

The main program is very simple. First, you enter a number. Next, the WHILE loop repeats the program execution until you enter a zero. Once in the WHILE loop, the FOR loop prints the table.

An interesting point in this example is the way that most of the program is enclosed in a WHILE loop. This allows the repeat of the program's main segment as many times as desired without its having to jump back in the program with a GOTO or similar statement, which would be necessary in BASIC.

Multiplication tables 2. This program extends the previous one and prints out the multiplication tables from 1 through 12, each table separated by a blank line. The program is written with two nested FOR-DO loops.

```
PROGRAM TABLES2(INPUT,OUTPUT);
CONST
  NUM=12;
VAR
  CTRL1,CTRL2,RES:INTEGER;
BEGIN
  FOR CTRL1:=1 TO NUM DO
    BEGIN
    FOR CTRL2:=1 TO NUM DO
     BEGIN
     RES:=CTRL1*CTRL2;
     WRITELN(CTRL1,' * ',CTRL2,' = ',RES)
     END;
    WRITELN
    END
END.
```

Sum and product of numbers. In the fourth example, two REPEAT–UNTIL loops are used to print the product and addition of all numbers between 1 and the number entered. The operation and result will be printed out. The comment lines are included in the listing to help you understand it; you don't need to type in these lines.

```
PROGRAM SUMPROD(INPUT,OUTPUT);
VAR
  NUM,SUM,PROD,I:INTEGER;
BEGIN
```

```
  SUM:=0;
  PROD:=1;
  I:=1;
  WRITE('INPUT ANY POSITIVE INTEGER NUMBER:  ');
  READLN(NUM);
(* ADD ALL THE NUMBERS *)
  REPEAT
    SUM:=SUM+I;
    WRITE(I,'  +  ');
    I:=I+1
  UNTIL I>NUM;
(* PRINT RESULT OF ADDITION *)
  WRITELN('  =  ',SUM);
  I:=1;
(* MULTIPLY ALL THE NUMBERS *)
  REPEAT
    PROD:=PROD*I;
    WRITE(I,'  *  ');
    I:=I+1
  UNTIL I>NUM;
(* PRINT RESULT OF PRODUCT *)
  WRITELN('  =  ',PROD)
END.
```

The main program asks for a positive integer number (1–7 only). Using two REPEAT loops, the program adds all the numbers from 1 to the number entered and then finds the product of all the numbers from 1 to the given integer. Note that the way this program is written you'll get extra plus and equal signs.

CHAPTER 5

Comparisons and Branches

CHAPTER 5
Comparisons and Branches

A very important function of any computer language is its ability to make decisions. Most programs compare two or more values and, according to the result of the comparison, execute one routine or another. The statement in Pascal which allows you to do this—which BASIC programmers will find familiar—is the IF-THEN statement.

IF-THEN

The Pascal IF-THEN structure has the following format:

IF ***condition***
  THEN ***statement***

For example, the line

```
IF A=B
  THEN WRITELN(A)
```

will print the value of A on the screen if the condition A=B is found to be TRUE. Otherwise, control passes to the next statement.

You might also use the IF statement to check for a certain character when reading values:

```
IF LETTER='S'
  THEN ...
```

Let's see how to write a short program to read a word entered by the user and count the number of *A*'s found in the word:

```
PROGRAM COUNT(INPUT,OUTPUT);
VAR
  LETTER:CHAR;
  COUNT:INTEGER;
BEGIN
  COUNT:=0;
```

```
  READ(LETTER);
  WHILE LETTER<>'  ' DO
  BEGIN
    IF LETTER='A'
      THEN COUNT:=COUNT+1;
    READ(LETTER)
  END;
  WRITELN('THE LETTER "A" WAS FOUND  ',COUNT,'  TIMES')
END.
```

Briefly, this program reads one character at a time, using a WHILE loop until a space character is found. In the loop, each character is compared to the letter *A*, and each time it is found, the counter increases by one.

The IF-THEN statement works the same way that it does in BASIC. But it also lets you do much more. You can use it to execute (or not) a group of statements enclosed within the keywords BEGIN and END. The structure is

IF ***condition***
  THEN
    BEGIN
    ... ***(statements)***
    END

In the next example, if the variable X is equal to 1, the following three statements enclosed within the BEGIN and END keywords will be executed. Otherwise, they will not be executed.

```
IF X=1 THEN
     BEGIN
     X:=X+1;
     WRITELN(X);
     Y:=X
     END
```

Adding the ELSE Condition

The IF-THEN statement can be even more useful if you add instructions to be executed if the condition stated is not TRUE. To do that, use the ELSE keyword:

IF ***condition***
  THEN ***statement1***
  ELSE ***statement2***

Either or both of the two parts can have several statements as long as they are enclosed by the keywords BEGIN and END.

The statement prior to an ELSE statement should not terminate with a semicolon; nor is there a semicolon following the ELSE. Here's an example:

```
IF NUM=10
  THEN WRITELN(NUM)
  ELSE NUM:=NUM+1;
```

The value of NUM is printed on the screen if NUM=10. Otherwise, one is added to the value of the variable. There is no semicolon following the statement WRITELN(NUM) because the keyword ELSE follows. Here's another example:

```
IF NUM>10 THEN
     BEGIN
     WRITELN(NUM);
     NUM:=NUM+1;
     COPY:=NUM
     END
  ELSE
     BEGIN
     NUM:=NUM+1;
     COPY:=NUM
     END;
```

In this program segment, several statements are used in both branches of the condition. If NUM is greater than 10, the value is printed, increased by one, and copied in the variable COPY. Otherwise, NUM is increased by one and copied, but not printed. Note that the END keyword before ELSE does not have a semicolon following it.

IF-THEN statements are also very useful to avoid errors in certain operations. Here is a routine to print the square root of a number:

```
PROGRAM SQROOT(INPUT,OUTPUT);
VAR
  NUM:REAL;
BEGIN
  WRITE('INPUT ANY POSITIVE NUMBER:');
```

```
  READLN(NUM);
  IF NUM<0
  THEN WRITELN('ONLY POSITVE NUMBERS')
  ELSE WRITELN('THE SQUARE ROOT IS   ',SQRT(NUM))
END.
```

This program checks to see whether the number entered by the user is negative. If it is not, the square root is calculated and printed. If it is, a warning message is printed.

When using several IF-THEN-ELSE structures together, you must be careful not to get confused as to which ELSE belongs to which IF. The ELSE keyword always belongs to the nearest IF statement which does not have an ELSE keyword. Indenting each nested loop also helps provide greater clarity in reading the program:

IF *condition1*
  THEN IF *condition2*
    THEN IF *condition3*
      THEN *statement1*
      ELSE *statement2*
    ELSE IF *condition4*
      THEN *statement3*

Try to avoid using nested loops that are too complex for someone reading your program to understand.

The CASE Statement

Often, the IF-THEN statement is not a practical solution to a problem. Consider the following program segment that counts the vowels found in a text entered by the user:

```
IF LETTER='A'
  THEN A:=A+1
  ELSE IF LETTER='E'
    THEN E:=E+1
    ELSE IF LETTER='I'
      THEN I:=I+1
      ELSE IF LETTER='O'
        THEN O:=O+1
        ELSE IF LETTER='U'
          THEN U:=U+1;
```

This is a clumsy way to write the program. The Pascal instruction **CASE** lets you separate program execution into as many branches as you need, similar

to BASIC's ON statement. CASE jumps to each separate routine according to the value of a variable or expression. The general structure of CASE is

CASE ***expression*** **OF**
  label1: statement1
  label2: statement2
  .
  .
  .
  label*n*: statement*n*
END

When a CASE statement is encountered in the program, the computer evaluates the expression and transfers control to the corresponding statement. Each statement is identified by a label (which can be thought of as a line number), which is one of the possible values of the variable in the CASE statement. The CASE expression must be a scalar type, and it may not be real. Each label in the CASE statement must be unique. To clarify this, let's rewrite our previous program segment:

```
CASE LETTER OF
  'A': A:=A+1;
  'E': E:=E+1;
  'I': I:=I+1;
  'O': O:=O+1;
  'U': U:=U+1
END;
```

As you can see from this example, the CASE statement must be ended by the END keyword to close the CASE section (CASE acts like BEGIN). Every line inside the CASE is indented for greater clarity. The computer finds the correct line by evaluating the variable (LETTER) and jumps to the line identified by the value found. For example, if LETTER is equal to I, the program will jump to execute the line labeled with the I. Two comments must be made about labels: First, character variable labels (as in the example) are enclosed in single quotation marks (other variable types are not). Second, with some implementations, if the CASE variable is assigned a value which has no matching label and the CASE statement is executed, a CASE ERROR will occur. Other versions of Pascal use the OTHERWISE statement, which allows the execution of the program to fall through to the next statement when no matching label is found for the value in the CASE variable.

Here is an example where the CASE statement branches according to the value of an integer variable:

```
CASE NUM OF
  1:WRITELN('ONE');
  2:WRITELN('TWO');
  3:WRITELN('THREE')
END
```

One general result might be needed for several labels. If this is the case, just combine the labels on one or more lines, separated by commas:

```
CASE NUM OF
  1,3,5,7,9:WRITELN('THE NUMBER IS ODD');
  2,4,6,8:WRITELN('THE NUMBER IS EVEN');
  0:WRITELN('THE NUMBER IS ZERO')
END
```

The following program example works like the first program in this chapter. The program counts the vowels in the word entered. When the space bar is pressed, a message tells how many vowels have been typed. If your implementation of Pascal does not have the OTHERWISE keyword, this program will not function properly.

```
PROGRAM VOWELS(INPUT,OUTPUT);
VAR
  LETTER:CHAR;
  COUNT:INTEGER;

BEGIN
  COUNT:=0;
  READ(LETTER);
  WHILE LETTER<>'  ' DO
  BEGIN
    CASE LETTER OF
      'A','E','I','O','U':COUNT:=COUNT+1;
      OTHERWISE
    END; (* of case *)
    READ(LETTER)
  END; (* of while *)
  WRITELN;
  WRITELN('YOUR WORD HAS  ',COUNT,'  VOWELS')
END. (* of vowels *)
```

Finally, as with the IF-THEN statement, you can combine several instructions in each CASE branch by enclosing them within the keywords BEGIN and END as in the following example:

```
CASE variable OF
  label1:BEGIN
      statements
    END;
  label2:BEGIN
      statements
    END;
  label3:BEGIN
      statements
    END;
END;
```

In this example, the CASE separates execution into three parts, according to the value of the variable. Remember to close the CASE statement with its own END keyword.

Example Programs

The following programs will help you with several of the concepts discussed in this and previous chapters. The first program asks the user to input an integer and then prints the factors of that integer. The MOD function makes this task easy.

```
PROGRAM FACTORS(INPUT,OUTPUT);
VAR
  I,NUM,FACTOR:INTEGER;
BEGIN
  WRITELN('INPUT A POSITIVE INTEGER NUMBER');
  READLN(NUM);
  IF (NUM>=0)
  THEN
    BEGIN
      WRITELN('THE FACTORS OF ',NUM,' ARE:');
      FOR I:=1 TO NUM DO
        BEGIN
         FACTOR:=NUM MOD I;
         IF FACTOR=0
            THEN WRITE(I,'   ')
        END
    END
  ELSE WRITELN('THE NUMBER IS NOT VALID.')
END.
```

Notice how the program checks to see whether the number read by the

program is valid: IF (NUM>=0) THEN. If it is not, the error message is printed in an ELSE statement.

Let's write a program that will count the number of words in a sentence entered by the user. Besides the period to mark the end of the sentence, the sentence can include only the comma as a punctuation mark. (To find the end of a word, the program checks for a comma or a space only.) The variable LE will be used to store each character entered by the user, one at a time. COUNT will be used to count the words in the text. A WHILE loop will be used to read letters.

```
PROGRAM COUNTER(INPUT,OUTPUT);
CONST
  PERIOD='.';
  COMMA=',';
  SPACE=' ';
VAR
  LE:CHAR;
  COUNT:INTEGER;
BEGIN
  WRITELN('ENTER A SENTENCE:');
  WRITELN;
  COUNT:=0;
  READ(LE);
  WHILE LE<>PERIOD DO
    BEGIN
      WHILE LE=SPACE DO
        READ(LE);
      WHILE (LE<>PERIOD) AND (LE<>COMMA) AND (LE<>SPACE) DO
        READ(LE);
      COUNT:=COUNT+1;
      IF LE<>PERIOD
        THEN READ(LE)
    END;
  WRITELN;
  WRITELN(COUNT,'  WORDS WERE FOUND.')
END.
```

In this program, we use a main loop to continue reading text as long as LE is not a period, indicating the end of the sentence. The statement

```
WHILE LE=SPACE DO READ(LE);
```

skips all spaces until the next word in the sentence. And

```
WHILE (LE<>PERIOD) AND (LE<>COMMA) AND (LE<>SPACE) DO
  READ(LE);
```

skips all the letters of the word, reading characters until a comma, space, or period is reached. When one of these characters is found, another word is added: COUNT. Then, if the character found is not a period, the next character is read, and the program returns to repeat the WHILE loop. Otherwise, control is returned to the WHILE loop, but LE will be equal to the period and control will leave the loop. Finally, the number of words found is printed.

Reading a Boolean Variable

In Chapter 3, we mentioned that the values of Boolean variables could not be read by using the READ or READLN statement. With the control structures covered so far, this can be done very easily. The following example asks you to input either a T for TRUE or an F for FALSE. Your reply will be stored in a character variable. Then, using an IF-THEN-ELSE statement, the program assigns the value of the Boolean variable REPLY accordingly.

```
PROGRAM BOOLREAD(INPUT,OUTPUT);
VAR
  REPLY:BOOLEAN;
  ANSWER:CHAR;
BEGIN
  WRITE('PLEASE REPLY (T)RUE OR (F)ALSE:  ');
  READLN(ANSWER);
  IF ANSWER='T' THEN REPLY:=TRUE
  ELSE REPLY:=FALSE
  WRITELN;
  WRITELN(REPLY)
END.
```

CHAPTER 6

Defining Data Types

CHAPTER 6
Defining Data Types

In addition to the four standard data types (INTEGER, REAL, BOOLEAN, and CHAR), you can define your own enumerated data types. This means that you can decide what possible values your variables may be assigned. An example will clarify this: Suppose you want to use a variable to store the different types of fruit that have been eaten by a person on each day of the week. To define your variable type, use the keyword **TYPE**:

```
TYPE
  FRUIT=(BANANA,ORANGE,APPLE,GRAPE)
```

Now, any constant or variable declared as FRUIT will be able to have only a value (fruit name) belonging to the list declared in the TYPE section. For example, you might have

```
CONST
  MEAL=ORANGE;
VAR
  DAY1,DAY2:FRUIT;
```

And you can assign in the program

```
DAY1:=APPLE;
DAY2:=BANANA
```

Remember, FRUIT is a *variable type,* just like INTEGER or BOOLEAN, and DAY1 and DAY2 are *variables* belonging to type FRUIT. BANANA, ORANGE, APPLE, and GRAPE are *values,* not variables. When you store the value BANANA in DAY1, DAY1 does not contain the string 'BANANA'; it has the value BANANA.

As with VAR and CONST, TYPE is not terminated with a semicolon.

TYPE declarations are included in the declarative section of a program, following the CONSTant declarations and before the VARiable declarations. After the keyword TYPE, you declare the different definitions, separating the variable type identifier from the list of possible values with the equal sign, just

as with constant definitions. The possible values that a variable defined as that type may have are enclosed within parentheses and are separated by commas.

Let's see another example. You might want to use a variable to store the days of the week. Here, you could use a user-defined type:

```
TYPE
  DAYS=(MON,TUE,WED,THU,FRI,SAT,SUN)
```

Variables would be declared as follows:

```
VAR
  FIRSTDAY,LASTDAY:DAYS
```

You might have noticed that the BOOLEAN variable type which you have been using is really the following declared type:

```
TYPE
  BOOLEAN=(TRUE,FALSE)
```

This declaration is done implicitly by the computer.

Many Pascal implementations let you make the type declarations directly in the VARiable declarations. Consider the WEEKDAYS example above. The variables FIRSTDAY and LASTDAY could be declared as

```
VAR
  FIRSTDAY,LASTDAY:(MON,TUE,WED,THU,FRI,SAT,SUN)
```

In this way, you omit the explicit TYPE declaration in the program. In most cases, it's not a good programming practice to declare variables like this. It's best to keep the TYPE declarations separate from the VAR declarations.

You may *not* assign the same possible variable value to two different TYPE declarations as in the following declaration:

```
TYPE
  FRUIT=(APPLES,ORANGES,BANANAS);
  DESSERTS=(ICE-CREAM,CAKE,ORANGES)
```

Here, ORANGES has been used in the declarations of two different variable types.

When you want to assign a value to a variable which has a defined data type, you do not need to enclose it within single quotation marks. Returning to the first example, see how the data type values are assigned to the corresponding variables:

```
TYPE
  FRUIT=(BANANA,ORANGE,APPLE,GRAPE);
VAR
  DAY1,DAY2:FRUIT;
```

```
BEGIN
  DAY1:=ORANGE;
  DAY2:=BANANA
```

Just as with the four standard data types we have been using so far, you may not mix variable assignments. This program segment would cause an error:

```
TYPE
  FRUIT=(APPLES,ORANGES,BANANAS);
  DAYS=(MONDAY,TUESDAY,WEDNESDAY);
VAR
  X:FRUIT;
  Y:DAYS;
BEGIN
  X:=MONDAY
```

Operators and Functions with Data Types

You may use only relational operators with variables belonging to the same data list. When comparing variables of the same group, the result will be either TRUE or FALSE. The order of the values in a defined data type is the order in which they have been declared in the TYPE data list. If you declare

```
TYPE
  SUBJECTS=(MATH,ENGLISH,FRENCH,GEOGRAPHY)
```

then these statements will return a value of TRUE:

```
MATH < ENGLISH
FRENCH > MATH
GEOGRAPHY > ENGLISH
```

Using PRED, SUCC, and ORD

You can use the functions PRED, SUCC, and ORD with enumerated TYPE declarations. ORD returns you the position of the value in the TYPE declaration list with the first element being in position 0. If the list is

```
TYPE
  COUNTRY=(GERMANY,FRANCE,BRAZIL,PANAMA);
```

then

```
ORD(GERMANY)=0
ORD(BRAZIL)=2
```

To move around in the list of values, you can use the SUCC and PRED functions. SUCC will return the following element, and PRED will return the previous element in the list. In other words,

```
PRED(FRANCE)=GERMANY
PRED(PANAMA)=BRAZIL
SUCC(FRANCE)=BRAZIL
SUCC(GERMANY)=FRANCE
```

Note that PRED is not defined for the first element in the list, and SUCC is not defined for the last element in the list.

Reporting Results

Using defined data types does have some limitations. The most important is that you may not print the value of a variable belonging to a defined data type by using WRITE or WRITELN. The following program segment is *incorrect:*

```
TYPE
  FRUIT=(BANANA,APPLE,ORANGE);
VAR
  X:FRUIT;
BEGIN
  X:=APPLE;
  WRITELN(X)
```

The program would not print the value of X when the WRITELN statement was encountered. Standard Pascal cannot print the value of defined data types. What you can do instead is to use a CASE statement to print the value of the variable. Correcting the previous example:

```
BEGIN
  X:=APPLE;
  CASE X OF
    APPLE:WRITELN('APPLE');
    BANANA:WRITELN('BANANA');
    ORANGE:WRITELN('ORANGE')
  END
```

Variable Subranges

Another kind of TYPE declaration is in using subranges. Subranges are used when you want the value of a variable to vary between a determined minimum and maximum value. You may define subranges over any scalar type except REAL. Consider this example:

```
TYPE
  RANGE=1..30;
VAR
  X:RANGE
```

Here, X will be able to vary only between 1 and 30. In other words, you are restricting the possible value of the variable X. Note that the possible minimum and maximum values of the subrange are separated by two consecutive periods.

You can also make the subrange declaration directly in the variable declaration section:

```
VAR
  X:1..20
```

Still, as we mentioned before, try to keep the TYPE and the VARiable declarations separate.

Subranges can be used to vary from one integer value to another, as in the previous examples:

```
TYPE
  RANGES=5..27
```

A subrange can also vary between characters and defined elements (the ORD is taken into account):

```
TYPE
  LETTERS='A'..'Z'
```

A declaration where the second limit is smaller than the first limit of a subrange is wrong. For example, this is *incorrect*:

```
TYPE
  RANGES=20..7
```

Once the subrange has been declared in the TYPE section, the corresponding variables belonging to that type are defined as usual:

```
TYPE
  RANGE=1..20;
  LETTER='A'..'Z';
VAR
  COUNTER:RANGE;
  VALREAD:LETTER
```

If you have two subranges of the same variable type, you can mix the corresponding variables in the different operations. Suppose you declare the following two integer subranges:

```
TYPE
  RANGE1=1..50;
  RANGE2=10..25
```

And you declare the variables:

```
VAR
  OPER1:RANGE1;
  OPER2:RANGE2;
  RESULT:INTEGER
```

Expressions such as

```
  OPER1+OPER2
  OPER2 DIV OPER1
  RESULT:=OPER1*OPER2
```

are correct, as long as you do not attempt to assign a value to a subrange variable, where the value does not belong to that variable's subrange.

Why would you want to use subranges when, in appearance, they are really not necessary? After all, it is essentially the same to declare a variable to vary, for example, from 1 to 20, or to declare it simply INTEGER. This is true, but subranges, mainly INTEGER subranges, should be used as often as possible. The use of subranges makes a program much easier to interpret and follow. By seeing the range of values between which each variable may vary, it's easier to see what that variable is used for in the program.

CHAPTER 7

Arrays, Strings, and Sets

CHAPTER 7
Arrays, Strings, and Sets

Arrays allow you to store a list of elements belonging to the same variable type. Before you can use an array, it must have been declared in the variable declaration section of the program.

Declaring an Array

The declaration of an array includes the variable name, array dimension(s), and type of variable. An array's index type and component type must be specified. The *component* type (the type to which all the elements of the array belong) may be any valid data type, including other arrays, that allows multi-dimensional arrays. The array *index* type may be any scalar or subrange type (it may not be REAL). A ten-element integer array named BOOKS would be declared as

```
VAR
  BOOKS:ARRAY[1..10] OF INTEGER
```

Note the syntax of the declaration. Here, the array is being declared in the variable section of the program (it could also be declared as a TYPE), with the array variable name, BOOKS, followed by a colon. Next, the keyword **ARRAY** (to make BOOKS an array variable) is used. The dimension of the array is inside square brackets, with the minimum and maximum values separated by two periods. Whenever you're using array variables, enclose the index with square brackets. Finally, the dimension is followed by the keyword **OF** and the variable type—in this case, INTEGER.

Let's take a look at another array declaration:

```
PEOPLE:ARRAY[1..12,1..3] OF REAL;
```

Here, the array name is PEOPLE, a two-dimensional array varying between 1 and 12 for the first dimension and between 1 and 3 for the second dimension. The array is used to store REAL values. For every extra dimension used, follow the range declaration with a comma and the following size dec-

laration. The number of dimensions allowed varies according to the Pascal implementation and computer being used. This declaration is in reality the same as declaring

```
PEOPLE:ARRAY[1..12] OF ARRAY[1..3] OF REAL;
```

Both notations are valid. The first example (the abbreviated declaration) is the most common.

Here are some other array declaration examples:

```
ADDCODES:ARRAY['A'..'Z'] OF 0..MAXINT;
YEARTEMPS:ARRAY[JAN..DEC] OF REAL;
ANSWERS:ARRAY[1..10] OF BOOLEAN;
```

The first declaration has an index ranging from the letter *A* to Z. Each array element may have an integer value from zero to MAXINT.

The second declaration has an index ranging from JANuary to DECember, which is a data type that must have been declared beforehand. For example,

```
TYPE
  MONTHS=(JAN,FEB,MAR,APR,MAY,JUN,JUL,AUG,SEP,OCT,NOV,DEC)
```

You could also write directly:

```
NUMCODES:ARRAY[MONTHS] OF REAL
```

Each array element can store a REAL value. This array could be used, for example, to store the average temperature for each month in a year.

The third example is a declaration of a Boolean array, where each of the elements may either be TRUE or FALSE.

Another way to declare an array is to define the array structure in the TYPE section of the program and then later assign the variable to that type:

```
TYPE
  LETTERS=ARRAY[1..28] OF CHAR;
VAR
  LET1,LET2:LETTERS
```

This is the same as explicitly declaring

```
VAR
  LET1,LET2:ARRAY[1..28] OF CHAR
```

(declaring two variables, LET1 and LET2, as character array variables).

Let's look at another possible array declaration. It's the same as any of the arrays we have seen so far, except the array is declared as **PACKED**:

```
NAME:PACKED ARRAY[1..30] OF CHAR
```

The difference between a PACKED and a normal array is mainly the memory usage of each structure. A PACKED array uses less memory. This makes the program more memory efficient, but access to each individual array component is slower. The computer and Pascal implementation that you are using will store the array in memory in a default way; so, to the Pascal programmer, there is no great difference in programming with packed or normal arrays.

But there is one exception. Packed arrays are used when you're working with character strings. This will be discussed later in the chapter.

Using Array Variables

Once an array has been declared in your program, you may use array variables just as in BASIC, except the index is always enclosed in square brackets. For example, if A is a one-dimensional integer array, you will assign a value to the third element like this:

```
A[3]:=12
```

This assigns a value of 12 to the third element of array A. The index may also be a variable or expression:

```
IF A[VAL]=A[VAL+1] THEN ...
```

Elements of arrays declared in the previous section can be accessed as follows:

```
YEARTEMPS[JUL]:=82   making 82 the average temperature of July
ANSWER[3]:=FALSE     making FALSE the third answer
```

An element from a two-dimensional (or larger) array may be accessed as

```
A[X,Y]:=value
```

where X and Y are the indexes, separated by commas. Again, this is the abbreviated format for the element A[X][Y].

Reading an Array

As we've already mentioned, to read a name or some other expression requires an array. Arrays are required because a variable declared as CHAR can be used to store only one character.

The following program uses a CHAR array to read a sentence without

any intermediate spaces. The sentence must be finished with a period. (The sentence here is read as an array of characters, not as a string, because our array variable is not declared as PACKED. The next section will discuss strings.)

```
PROGRAM SENTENCE(INPUT,OUTPUT);
VAR
  SENT:ARRAY[1..30] OF CHAR;
  LP,INDEX:INTEGER;
BEGIN
  INDEX:=1;
  WRITELN('PLEASE ENTER A SENTENCE');
  READ(SENT[INDEX]);
  WHILE (SENT[INDEX]<>'.') AND (INDEX<30) DO
    BEGIN
      INDEX:=INDEX+1;
      READ(SENT[INDEX])
    END;
  WRITELN;
  WRITELN;
  FOR LP:=1 TO INDEX DO
    WRITE(SENT[LP])
END.
```

The program uses a one-dimensional, 30-element character array to store the sentence. The array is called SENT. Variable INDEX will be used to access the array elements, and it is assigned an integer subrange between 1 and 31. When actual program execution begins, the user enters a sentence; this is done in a WHILE loop. The program reads characters as long as a period is not found and while there are fewer than 30 characters in the sentence. Each character read is stored in array SENT.

When the sentence has been read, printing is done by using a FOR loop. The FOR loop will print all characters of the sentence. Another variable in the FOR loop, LP, is used to cycle from 1 to INDEX (INDEX stores the number of characters in the sentence).

Comparing and Copying Arrays

Two array elements may be compared in the usual way, specifying the index of each element.

```
IF A[2]=A[4] THEN ...
```

compares the second and fourth element of array A.

The elements compared must be of a compatible type. To copy two complete arrays, both arrays must have identical declarations (type and size). If A

and B have been declared identically, this expression will copy every element of B into A:

```
A:=B
```

Searching an Array

Searching an array is a simple task if you use a WHILE loop, illustrated by the following program segment. The routine searches an integer number array (NUMTABLE) for a specific value (INTNUM). The number of elements in the array is stored in LENGTH.

```
FOUND:=FALSE;
READLN(INTNUM);
POSIT:=1;
WHILE (FOUND=FALSE) AND (POSIT<LENGTH+1) DO
  BEGIN
    IF NUMTABLE[POSIT]=INTNUM
      THEN FOUND:=TRUE
      ELSE POSIT:=SUCC(POSIT)
  END;
```

Upon exit from this loop, either the element is found (FOUND=TRUE) or is not found (FOUND=FALSE). This program segment reports these results:

```
IF FOUND
  THEN WRITELN('FOUND IN   ',POSIT)
  ELSE WRITELN('NOT FOUND')
```

Multidimensional arrays are searched in a way similar to simple arrays, with the array indexes varying through all the possible combinations. In this example, a WHILE loop was preferred to a FOR-DO loop because a FOR loop cycles through all the array elements no matter at what point the match is found (if it is). The WHILE loop in the above segment will stop searching once a match is found.

All searches can be performed with any type of array, varying the index through all possible combinations. If you want to search through an array with the index type being a series of months (JAN, FEB, MAR, and so forth), just make the index variable cycle through JANuary to DECember.

```
FOR MONTH:=JAN TO DEC DO
  READLN(YEARTEMPS[MONTH]);
```

Array Limitations

Arrays are useful for storing tables of data. But they have one severe limitation: They have a fixed size, or dimension. In other words, if you dimension an

array of size 50, and you use fewer than 50 elements, you will be wasting memory. Using more than 50 elements will cause the program to abort with an execution error. These same limitations apply to most versions of BASIC, where an array variable must be dimensioned and then may not be redimensioned. Pascal offers an alternative: pointers and dynamic data storage. Chapter 12 will explain pointers and dynamic storage.

Strings in Pascal

String values in Pascal are stored in packed character arrays. To store four-character strings, you would declare the following structure:

```
NAME:PACKED ARRAY[1..4] OF CHAR;
```

(NAME is the variable name.)

You can then assign a value to string variable NAME like this:

```
NAME:='JOHN'
```

String values assigned to the string variable must be of the exact length assigned to the packed array in the array declaration. If the string variable to be assigned is shorter than the size of the packed array, pad it with spaces:

```
NAME:='JOE  '
```

The following assignments are *incorrect* if NAME is declared as above:

```
NAME:='BO'      (only two characters)
NAME:='SUSAN'   (four characters maximum)
```

Compared with other programming languages, string manipulation is limited in Pascal. It's limited because string lengths are declared at the beginning of the program and may not vary during program execution.

String variables (packed character arrays) are used in the same way as normal array variables. They may be copied and compared using the array name, as long as the respective lengths are the same.

Strings of the same length may also be compared to determine their order. The computer compares the two strings, one character at a time, until it finds two characters that are different. The character with the greater ASCII code is the larger of the two characters. For example,

```
'DOG'>'CAT'                  because D > C
'MARY      '>'MARIANNE'      because Y > I
```

You cannot use READ or READLN to read a string value directly from the keyboard (though some Pascal implementations let you do this in a way similar to the operation of BASIC's INPUT statement: If the number of charac-

ters entered by the user is greater than the parameters specified, the extra characters are ignored). To read a string, you must use a loop.

This program reads a string variable and then prints it in reverse:

```
PROGRAM STR(INPUT,OUTPUT);
VAR
  STRVAR:ARRAY[1..10] OF CHAR;
  I:INTEGER;
BEGIN
  WRITELN('ENTER A TEN CHARACTER STRING');
  FOR I:=1 TO 10 DO
    READ(STRVAR[I]);
  WRITELN;
  WRITE('THE REVERSE STRING IS:');
  FOR I:=10 DOWNTO 1 DO
    WRITE(STRVAR[I])
END.
```

In this program, two loops do the job easily. The array variable STRVAR is used to store the ten-character string read. The variable I is the array index variable which ranges between 1 and 10. When the program begins, a FOR loop is used to read a string from the keyboard. The loop expects ten characters to be input. If fewer are entered, the computer waits for more input. If more are entered, the overflow characters are ignored. Another FOR loop is used to count down from 10 to 1 to print the string in STRVAR in reverse.

You may use the standard function **EOLN** to test for the end of the user's input. In this way, when the program detects the RETURN key, it knows that the user's input is ready, without having to read a specific number of characters (like 10 in the example). We'll discuss this in an example later in this chapter, and we'll formally introduce it when dealing with files in Chapter 10.

Formatted String Output

If you output a string constant by using WRITE, you may follow the constant name or string itself by a colon and an integer value:

```
WRITE('JOHN':3);
```

If the value following the string is smaller than the length of the string, only that number of characters, starting from the left of the string, is printed. The above statement will print JOH (only the three left characters of the string will be printed). If the statement is

```
WRITELN('MARY SMITH':6);
```

the printed result will be MARY S .

If the number following the expression is equal to the number of characters in the string, the string will be printed with no modifications.

Finally, if the number following the colon is greater than the number of characters in the string, *N* spaces will be printed before the string, where *N* is the value after the colon minus the length of the string. For example,

```
WRITELN('SUSAN':8)
```

prints three blanks and then SUSAN:

```
   SUSAN
```

The statement WRITE(' ':N) is very useful in Pascal because it will tab *N* spaces on the current line. The lines

```
WRITE(' ':20);
WRITELN('HELLO');
```

will print HELLO starting in column 21 of the current line.

Try this short demostration program:

```
PROGRAM TABIT(OUTPUT);
CONST
  START='START IN COLUMN  ';
VAR
  I:INTEGER;
BEGIN
  FOR I:= 1 TO 20 DO
  BEGIN
    WRITE(' ':I);
    WRITELN(START,I+1:2)
  END;
END.
```

Pack and Unpack

Pascal allows you to **PACK** a normal array into a PACKED array variable, or to **UNPACK** a PACKED array into a normal array. This operation may help you gain some speed which is lost when gaining access to individual elements in packed arrays. You might, for example, read a name as a normal array—with fewer problems than using a packed array. Then, when you want to save the name to disk, PACK it into a packed array and store it using less disk space. You can also do this in reverse, reading a packed array from disk and UNPACKing it into a normal array for use in the program. The array into which you're PACKing or UNPACKing a string must be at least the same length as the string you're packing (unpacking) or a greater length and of the same type.

If you define the arrays

```
SENTENCE1:ARRAY[1..30] OF CHAR;
SENTENCE2:PACKED ARRAY[1..30] OF CHAR;
```

you could use SENTENCE1 while working in the program and PACK it into SENTENCE2 when you're about to save the variable to disk.

The statement

```
PACK (SENTENCE1,1,SENTENCE2);
```

copies every element of SENTENCE1 into SENTENCE2. This Pascal statement is equal to

```
FOR I:=1 TO 30 DO
  SENTENCE2[I]:=SENTENCE1[I];
```

To transfer a packed array to a normal array, use the following statement:

```
UNPACK(SENTENCE2,SENTENCE1,1);
```

This statement copies every element from the packed array, SENTENCE2, into the corresponding position in the unpacked array, SENTENCE1, and is equivalent to the statement

```
FOR I:=1 TO 30 DO
  SENTENCE1[I]:=SENTENCE2[I];
```

The general format for PACK and UNPACK is

```
PACK(ARRAY1,INDEX,ARRAY2)
```

where ARRAY1 is the unpacked array; ARRAY2 is the packed array of the same or greater dimension than ARRAY1, with the same component type as ARRAY1; and INDEX is the index type of ARRAY1. This procedure copies ARRAY1[INDEX], ARRAY1[SUCC(INDEX)], and so on, into the corresponding elements of ARRAY2:

```
UNPACK(ARRAY2,ARRAY1,INDEX)
```

where ARRAY2 is the packed array; ARRAY1 is the unpacked array of equal or greater dimension than ARRAY2, with the same component type; and INDEX is the index type of ARRAY1. This procedure copies all the elements from ARRAY2 into ARRAY1[INDEX], ARRAY1[SUCC(INDEX)], and so forth.

The PACK and UNPACK operations are necessary only when you're working with large amounts of data which you know will save a lot of memory space when PACKed into a packed array. The memory saved also depends on

the computer and Pascal implementation you are using. Otherwise, these procedures needn't be used.

Declaring a Set

Pascal offers a very useful programming structure called a *set*. As in mathematics, a set in Pascal is a group of elements of the same type, which are treated as a group, but which have no particular order. An element either belongs or does not belong to a set. That is, the set of characters (A, B, C) is the same as (B, A, C), (C, C, B, A), (A, A, B, C, B, C), and so forth. Just as in mathematics, Pascal lets you perform such operations as joining two sets and intersecting two sets. Let's start by seeing how to define a set in a program.

You declare a set by making a list of its possible elements in a TYPE declaration, declaring a set structure using that data list, and then declaring set variables. The type of the set can be any scalar type except REAL. An example of a set of fruit will clarify this. First, we enumerate the list of all possible fruit in the set in our TYPE section:

```
TYPE
  FRUIT=(BANANA,ORANGE,APPLE);
```

Now we define our set structure with the keywords **SET** and **OF**. This definition is also included in the TYPE declarations:

```
SET1=SET OF FRUIT;
```

In other words, the set structure SET1 is a set with three possible elements: BANANA, ORANGE, and APPLE. Now let's define our set variable(s) in the variable section of the program:

```
VAR
  VAR1,VAR2:SET1
```

And where do all these declarations lead us? First of all, SET1 is a set type associated with another type called FRUIT. Any constant or variable belonging to type SET1 must have a value belonging to the list of FRUIT.

Alternatively, you can declare the set directly in the VARiable section, but you should try to keep declarations separate:

```
VAR
  VAR1,VAR2:SET OF (BANANA,ORANGE,APPLE)
```

Placing Values in a Set

Our set variables have now been defined. No value has been assigned to them yet. The elements which may be assigned to the set must belong to the base

type of that set (FRUIT, in the example). You could then write

```
VAR1:=[BANANA];
VAR2:=[APPLE,ORANGE]
```

Note that a set is represented by square brackets. Within the brackets, you write the set elements, separating them by commas if there is more than one element. In the first example, VAR1 is a set with only one element (BANANA). The second set, VAR2, has two elements (APPLE and ORANGE).

If your set is composed of elements which are consecutive in your type declaration for that set, you need specify only the first and last elements in the list (as in the following examples). First, declare the set list, set structure, and set variable:

```
TYPE
  TOOLS=(HAMMER,SAW,SCREWDRIVER,WRENCH,DRILL,PLIERS);
  TOOLSET=SET OF TOOLS;
VAR
  TOOLBOX:TOOLSET
```

The variable assignment

```
TOOLBOX:=[SAW..WRENCH]
```

is equal to

```
TOOLBOX:=[SAW,SCREWDRIVER,WRENCH]
```

Or the assignment

```
TOOLBOX:=[SCREWDRIVER..PLIERS]
```

is equal to

```
TOOLBOX:=[SCREWDRIVER,WRENCH,DRILL,PLIERS]
```

This can be done only if the elements of the set assignment are consecutive in the type declaration list.

An empty set is declared by assigning a pair of square brackets without any element in them. For example,

```
TOOLBOX:=[ ]
```

declares TOOLBOX as an empty set (with no elements).

Set Operations

All set operations performed in normal arithmetic can be executed in Pascal with sets that have the same base type, that are subranges of the same base

type, or that have one base type a subrange of the other. These set operations will be described in this section.

The *union* of two sets means that you find a set which results from joining two sets together. Any elements common to both sets are copied only once into the new set, which is called the *union* of the two sets. In mathematics, this operation is represented by the U symbol. In Pascal, the plus sign (+) is used:

```
[1,7,9] + [2,3] = [1,2,3,7,9]
[MIKE,ANN,JENNY] + [NICK,ANN] = [MIKE,JENNY,NICK,ANN]
[A,B] + [ ] = [A,B]
```

Remember that the elements in a set have no particular order, so the sets

```
[1,2,3]=[2,1,3]=[1,3,2]
```

are equal.

Intersecting two sets means a new set is formed from the elements which are *common* to both sets that you are operating with. The mathematical symbol for intersection is ∩. In Pascal, the operator used is the asterisk (*). Using Pascal symbology, here are the results of intersecting a few sets:

```
[1,3,7,2] * [1,7,5] = [1,7]
[RED,BLUE,YELLOW,GREEN] * [BLUE,WHITE] = [BLUE]
[A,B,H,D,J] * [K,C] = [ ]
```

Finding the *difference* between two sets means finding all the elements which belong to the first set, but do not belong to the second set. The symbol for difference is the minus sign. This statement returns all the elements in set A which do not exist in set B:

```
A-B
```

Here is another difference example:

```
[BLUE,RED,ORANGE] - [RED,GREEN] = [BLUE,ORANGE]
```

Whole sets with compatible base types can also be compared with the same symbology used elsewhere.

`A=B` Sets A and B are equal.

`A<>B` Sets A and B are not equal.

`A<=B` Set A is included in set B (all the elements in set A exist in set B).

`A>=B` Set B is included in set A (all elements in set B exist in set A).

These examples use this symbology:

```
[A,B,C]=[B,A,C]     TRUE
[A]<=[A,B,C]        TRUE
[A,B]>=[C]          FALSE
[1,2,3]<>[3,2,5]    TRUE
[A,B]<=[B,A,C,L]    TRUE
```

An important question that will eventually arise is, Does an element belong to a certain set? In mathematics, to check whether an element belongs to a set, the ∈ symbol is used. Pascal lets you check this with the **IN** keyword. For example, the expression

```
RED  IN [BLUE,GREY,RED]  is TRUE
1    IN [3,4,7,9,2]      is FALSE
```

Set Program Example

Let's write a simple program that uses a set. With this program, when a user enters a word, the program will count the number of vowels in that word. Without sets, determining whether a certain letter is a vowel would require five different IF-THEN statements or a loop to check a letter against a table of vowels stored in an array. Sets greatly simplify this process, since you need only to define a set of vowels and then check within the program:

IF *variable* IN VOWELSET THEN ...

The EOLN function is used to find the end of the input from the user. EOLN is discussed briefly in the program explanation and will be discussed more fully in Chapter 10.

```
PROGRAM VOWELS(INPUT,OUTPUT);
TYPE
  VSET=SET OF CHAR;
VAR
  VOWELSET:VSET;
  LETTER:CHAR;
  COUNT:INTEGER;
BEGIN
  COUNT:=0;
  VOWELSET:=['A','E','I','O','U'];
  WRITELN('INPUT A WORD:  ');
  READ(LETTER);
  WHILE NOT(EOLN) DO
    BEGIN
```

```
      IF LETTER IN VOWELSET
        THEN COUNT:=COUNT+1;
      READ(LETTER)
    END;
  WRITELN(COUNT,'  VOWELS WERE FOUND.')
END.
```

In the TYPE declarations of the program, we define a set of vowels called VSET, and the base type for the set is defined as CHAR. In the variable section of the program, we declare the set variable, VOWELSET; the variable to store each character read, LETTER; and the variable to store the number of vowels in the word, COUNT.

COUNT is set to zero, and VOWELSET is assigned as the complete set of vowels. The input prompt for the word is printed, and a loop to read the word begins.

In the WHILE statement of this loop is an EOLN (end-of-line) function. This function will return a value of TRUE until the end of the input line has been found. Otherwise, EOLN will return FALSE, thus staying in the WHILE loop. (More on EOLN later.) This means that the WHILE loop will continue reading characters until the user enters RETURN. In the WHILE loop, each character entered is stored in LETTER and is checked to see whether it belongs to the set of vowels, VOWELSET. If it does, COUNT is increased by one. Finally, when execution leaves the WHILE loop, the number of vowels found—stored in COUNT—is printed.

Changing Base Example

This example program uses sets to convert a number in base 16 into its corresponding value in base 10. Two sets are used: One for the letters *A–F* and another for the digits 0–9. First, the constants, data types, and variables are declared:

```
PROGRAM CONVERT(INPUT,OUTPUT);
CONST
  BASE=16;
  SPACE=' ';
TYPE
  SETCHAR=SET OF CHAR;
VAR
  DATA:CHAR;
  NUM,ELEM:INTEGER;
  LETTER,DIGIT:SETCHAR;
```

Variable DATA will store each character read, one at a time. NUM will be used as an accumulator of the decimal number, and ELEM will be used in the conversion operations. LETTER and DIGIT are two sets, one of letters *(A–F)* and one of digits (0–9).

```
BEGIN
  NUM:=0;
  LETTER:=['A','B','C','D','E','F'];
  DIGIT:=['0'..'9'];
  READ(DATA);
```

In this segment, the accumulator, NUM, is set to zero, and the two sets are initialized. Then, the first character is read from the hexadecimal value input by the user in DATA. Next, the conversion is performed:

```
  WHILE DATA<>SPACE DO
  BEGIN
    IF DATA IN DIGIT THEN ELEM:=ORD(DATA)-ORD('0')
    ELSE IF DATA IN LETTER THEN ELEM:=ORD(DATA)-ORD('A')+10
    ELSE WRITELN('WRONG INPUT:',DATA);
    NUM:=NUM*BASE+ELEM;
    READ(DATA)
  END;
  WRITELN('THE DECIMAL EQUIVALENT IS:',NUM)
END.
```

Here, the decimal value is calculated, using a WHILE loop to read hexadecimal digits until a space character is found. (End your input with a space followed by a RETURN. On some systems, pressing RETURN without a space is sufficient.)

The hexadecimal digit is then checked. If it's a number (IN DIGIT), its decimal value is calculated by subtracting the ASCII code of zero from the digit's ASCII code. If the character is a letter *(A–F)*, its decimal value is its ASCII code minus the ASCII value of the letter *A* plus 10. Otherwise, the character is wrong, and a message is printed. (Although the operation will still be calculated, the result will be incorrect. You could use a BOOLEAN flag to end the operation.) The accumulator is increased by multiplying itself by the base (16) and adding the new value, ELEM. Then a new DATA is read and checked once again in the WHILE statement. When control leaves the WHILE loop, the value of NUM is printed.

On most computers when you run this program, the maximum hexadecimal value that can be entered is 7FFF. Otherwise, an INTEGER OVERFLOW error will occur because the value of MAXINT has been exceeded. Remember, MAXINT is 32767 in base 10 and 7FFF in base 16.

CHAPTER 8

Functions and Procedures

CHAPTER 8
Functions and Procedures

You use subroutines in BASIC when you need a certain group of statements to be executed repeatedly in various parts of your program, or when you want to make your program clearer by separating it into different execution modules. Pascal lets you use structures that are similar to subroutines, but are much richer. Pascal has two types of subprograms: *functions* and *procedures.*

Programs Within a Program

Whenever you need a program segment to be repeated at various points in your program, you may use a procedure. All procedures used in a Pascal program must be defined at the beginning of the program, before the actual Pascal program begins. Using the keyword **PROCEDURE**, you state the name of the routine and optional parameters. A procedure is called by the name you have given it. The parameters are values or expressions which are passed from the main program to the procedure. Let's look at a simple procedure that prints three lines of text each time it is called. The procedure is declared as follows:

```
PROCEDURE WRITETEXT;
BEGIN
  WRITELN('SENTENCE #1');
  WRITELN('SENTENCE #2');
  WRITELN('SENTENCE #3')
END;
```

The procedure is called WRITETEXT, and no parameters are passed to it. Note that the structure of a procedure is similar to that of a program. All instructions belonging to the routine are enclosed by the keywords BEGIN and END, which mark all the statements belonging to it. Thus, it is not necessary to end the procedure with any particular word (like RETURN in BASIC) because, when the computer finds the END corresponding to the first BEGIN of the subprogram, it knows that the end of the routine has been found. This END has a semicolon after it. Control then returns to the calling program, which can be the main program or another procedure.

All procedures are placed after variable declarations and the BEGINning of the program. Here is a complete program that uses the above procedure. The procedure is called twice: Six sentences are printed on the screen.

```
PROGRAM EXAMPLE1(INPUT,OUTPUT);
PROCEDURE WRITETEXT;
BEGIN           {of the procedure}
  WRITELN('SENTENCE #1');
  WRITELN('SENTENCE #2');
  WRITELN('SENTENCE #3')
END;            {of the procedure}

BEGIN           {of the main program}
  WRITETEXT;    {calls the procedure}
  WRITETEXT     {calls the procedure again}
END.            {of the main program}
```

Global Parameters

Procedures are similar to small Pascal programs in and of themselves. As well as having their own names, procedures may have their own constants, types, variables, and subprograms defined.

Identifiers (constants, types, variables, and subprograms) are defined or not defined in a subprogram according to where in the program they have been declared. The procedure can use all the declarations that are defined in the main program or in the subprogram which calls it. These declarations that are not passed to the procedure through its parameter list are called *global* values. In the case of the BASIC subroutine structure, all variables in the subroutine are shared with the main program and vice versa. That is, you don't have variables which are specific to the subroutine. In Pascal, the procedure may use all the variables general to the program—all the identifiers defined before the subprogram is called. If any of the global variables are modified in the procedure, they will contain the modified value upon return to the calling program, as in the following case:

```
PROGRAM GLOBAL(INPUT,OUTPUT);
VAR
  X:INTEGER;
PROCEDURE ADDTHREE;
BEGIN
  X:=X+3
END;
```

```
BEGIN
  X:=1;
  ADDTHREE;
  ADDTHREE;
  ADDTHREE;
  WRITELN(X)
END.
```

The procedure ADDTHREE simply adds the value 3 to the current value of variable X. Since no variable X is specifically defined for the procedure, the variable X of the calling program is used, and the value of X is modified in the main program by the procedure. Thus, the computer will perform the following calculation:

X=1+3+3+3

The resulting value printed will be 10. In this example, variable X is a *global* variable, declared in the main program and valid in all program procedures at different execution levels *unless specifically declared for that subroutine.*

Local Parameters

If you declare a CONSTant, TYPE, or VARiable in the procedure itself, you're creating identifiers specific to that procedure, values which the calling program won't know exist. These values are called *local* because they exist only in the subroutine itself and are no longer valid upon return from that subroutine. Here's an example:

```
PROGRAM LOCAL(INPUT,OUTPUT);
VAR
  NUMBER:REAL;
PROCEDURE ONLYHERE;
VAR
  NUMBER:REAL;
BEGIN            {of the procedure}
  NUMBER:=27.3
END;             {of the procedure}
BEGIN            {of the main program}
  NUMBER:=0.0;
  ONLYHERE;
  WRITELN(NUMBER)
END.
```

First, NUMBER is defined as a REAL variable. Next is the procedure ONLYHERE, which declares a REAL variable, also called NUMBER. The procedure assigns a value of 27.3 to the variable NUMBER and then ends.

The main program assigns a value of zero to NUMBER, calls the procedure ONLYHERE, and prints the value of NUMBER. Can you guess the value printed? If you think 27.3 will be printed, read this section again.

The program will print a zero because the variable NUMBER declared in the main program is not the same variable (even though the name is the same) as the variable NUMBER specifically declared in the procedure. While ONLYHERE is executing, NUMBER is a global variable until declared a local variable for the procedure ONLYHERE in the procedure declarations. If you wanted the procedure to change the value of NUMBER, you would not declare it in the procedure by leaving out the fifth and sixth lines of the program (the variable declaration for the procedure).

Passing Parameters

Using global variables is poor structured programming because the variable may be changed from anywhere within the program. In good structured programming, a variable is changed only in the section of the program in which it is declared or in a subprogram to which its value has been passed. If you want a procedure to use values from the main program, without being forced to use global variables, the solution is to pass these values between the different program sections by using the *procedure header.* This information is enclosed within parentheses, *both* in the procedure call and in the procedure header. You may pass constants or variables to the subprogram, with certain restrictions. Parameters must correspond in number, position, and type in the procedure call and in the procedure header. The parameters may be completed with dummy parameters if necessary. Here's an example of a procedure header with parameters:

```
PROCEDURE HELLO(X:INTEGER;Y,Z:REAL)
```

A call to this procedure might look like this:

```
HELLO(N,Y,A)
```

The assignment of values when control is passed to the procedure is

```
X:=N     Y:=Y     Z:=A
```

Let's take a closer look at the syntax used for the procedure header parameters. Identifiers must be declared with their corresponding types, and the types have to match the types of the values passed as parameters in the procedure call.

In the example, the procedure's parameters are X (INTEGER), Y, and Z (REAL). The corresponding variables N, Y, and A in the calling program must also be INTEGER, REAL, and REAL, respectively. When the procedure is called, a copy of each parameter, X, Y, and Z, is stored in the variables N, Y, and A. Though the procedure will use the values of X, Y, and Z to work with, these values will not be modified in the calling program because the procedure is using its own working copies, N, Y, and A. These parameters are said to be passed by value because only their values are used in the subprogram, while the values in the calling program remain unchanged. In other words, the variables N, Y, and A in the procedure are different from the variables of the calling program.

You might imagine N, Y, and A as if they had been declared in a VAR declaration in the procedure—the only difference being that their values are already assigned (by the values of X, Y, and Z). When returning to the main program, X, Y, and Z will still have the same values they had when the routine was called. Here's another example:

```
PROGRAM PARAMETERS(INPUT,OUTPUT);
VAR
  NUM1,NUM2:INTEGER;
PROCEDURE PASSVALS(COPY1,COPY2:INTEGER;SYMBOL:CHAR);
BEGIN
  WRITE(COPY1,SYMBOL,COPY2,'=');
  WRITELN(COPY1*COPY2)
END;
BEGIN
  NUM1:=2;
  NUM2:=3;
  PASSVALS(NUM1,NUM2,'*')
END.
```

In this example, two variable values are passed to the procedure (NUM1, NUM2), which will be called COPY1 and COPY2 in the procedure. The values of NUM1 and NUM2 will remain unchanged throughout the program. A third parameter is passed, but it is not a variable. It is a character, and it is assigned directly to variable SYMBOL in the procedure.

The procedure then uses these values to print an operation and return. Remember, the values of the variables used to pass the values to the procedure do not change.

Passing Values Back to the Calling Program

You may use another syntax to pass the parameters to the procedure, which will cause the variables in the calling program to be updated together with the corresponding variables in the procedure. Thus, the subprogram is allowed to pass results back to the calling program through the parameter list. You do this by using the keyword VAR before variable declarations in the procedure header. These parameters are said to be passed by reference because the address, not the value, of the variables is passed to the procedure. Constants and expressions may not be passed by reference because their value may not be changed by the procedure. Here's a procedure header with one pass-by-reference parameter (and no pass-by-value parameters):

```
PROCEDURE DOESCHANGE(VAR X:INTEGER);
```

A call to this procedure could be

```
DOESCHANGE(NUM);
```

where NUM must be an INTEGER variable.

What happens with the values of the variables involved in this case? The value of NUM will always be equal to the value of X because, as far as the machine is concerned, NUM and X are the same variable. If X is changed in the procedure, the value of NUM will be updated accordingly. This imposes one restriction on the procedure call: A constant or expression, like a number or character, may not be passed directly to the procedure, as in the previous example, where the character * is passed and then stored in the variable SYMBOL.

A Quick Review

Here's a brief summary of the different procedure types and the different ways of handling variables in each one as well as any restrictions:

Procedure call: NAME;
Procedure header: PROCEDURE NAME;
No declarations in the procedure.

In this case, the procedure can use only the values defined in the main program. Any values changed by the procedure will also be changed in the calling program. Identifiers are the same in all parts of the program.

Procedure call: NAME;
Procedure header: PROCEDURE NAME;
Declarations in the procedure.

When CONST/TYPE/VAR declarations exist in the procedure, these values exist only in the procedure. Local declarations for the procedure do not

exist upon return to the calling program. Global values can still be changed in the main program by the procedure if they haven't been declared with the same name in the PROCEDURE declarations.

Procedure call: NAME(VAR1,VAR2);
Procedure header: PROCEDURE NAME(X,Y:INTEGER);
Declarations specific to the procedure may or may not exist.

When parameters in the procedure header are not preceded by the keyword VAR, the values of the corresponding identifiers of the calling statement will remain unchanged in the calling program. Constants, types, variables, and expressions may be passed directly to the procedure. These are pass-by-value parameters.

Procedure call: NAME(VAR1,VAR2);
Procedure header: PROCEDURE NAME(VAR X,Y:INTEGER);
Declarations specific to the procedure may or may not exist.

By preceding the parameter list in the procedure header with the keyword VAR, you will update the values of the corresponding variables in the calling program together with the variables in the procedure. Constants or expressions may not be passed directly to the subprogram, because no variable would exist to be updated in the procedure call. Only variables may be used. In this case, the parameters are passed by reference.

More About Procedures

You can pass arrays and records between calling programs and subprograms by using the general identifier. When the arrays and records are large, it's best to pass them by reference; then the computer will not need to make a copy of all the values in the array or record.

Pass-by-value and pass-by-reference parameters may be combined in the same parameter list. For example,

```
PROCEDURE BOTH(X,Y:INTEGER;VAR FLAG:INTEGER);
...
...
BOTH(NUM1,NUM2,RES);
```

In this procedure header and call, X and Y are pass-by-value parameters which will use the values of NUM1 and NUM2 to operate, without modifying their values. FLAG is a pass-by-reference parameter corresponding with RES. When RES is modified in the subprogram, so is the value of FLAG.

A procedure may call another procedure which has been declared in the same program. So any program section may call the use of any routine, as long

as that routine is defined before the calling section. But in many cases, a subprogram is used only by another procedure and not by the program or any other routine. If that's the case, you may use *nested procedures*—that is, declare a procedure in the declarative section of another procedure. Here's such an example:

```
PROCEDURE OUTSIDE(X:INTEGER);
  VAR
  SYMBOL:CHAR;
  PROCEDURE INSIDE(SYM:CHAR);
...
...
```

In this program segment, the procedure OUTSIDE could be called directly, but procedure INSIDE can be accessed only by procedure OUTSIDE. INSIDE is specific to OUTSIDE. Think of OUTSIDE as the main program and INSIDE as a procedure of that program.

When you use nested procedures, be careful at all moments to keep control of what identifiers exist, are accessible, or are being changed; thus, you can avoid any mysterious effects upon return to the main calling program.

Evaluating a Procedure

The computer follows these steps when a procedure call is found in a program:

1. The parameters in the procedure are aligned on a one-to-one basis with the parameter list in the procedure header. There must be the same number of parameters, of corresponding types, in the procedure call and procedure header.
2. For the pass-by-value parameters (without VAR), positions in memory are created for the local parameters. The values of the identifiers are already stored in memory.

   For the pass-by-reference parameters (with VAR), the address of the parameters in the procedure call is assigned to each parameter in the procedure header, so both values (in the calling program and procedure) will be updated together.
3. Places in memory are created according to local declarations with their values still undefined.
4. The procedure is executed, with pass-by-value parameters acting as local parameters and pass-by-reference parameters being updated in the calling program together with the procedure.
5. Upon return from the procedure, all local memory used by the pass-by-value parameters and local variables is freed.

Procedure Example Program

The following simple program will help you visualize the parameter relations when you're using procedures. The program searches a three-element integer array (TABLE) for an integer value entered by the user.

```
PROGRAM FIND(INPUT,OUTPUT);
TYPE
  ARTABLE=ARRAY[1..3] OF INTEGER;
VAR
  TABLE:ARTABLE;
  DATA:INTEGER;
  RESULT:BOOLEAN;
PROCEDURE SEARCH(INTTABLE:ARTABLE;VAR FLAG:BOOLEAN);
VAR
  I:1..3;
BEGIN
  FLAG:=FALSE;
  FOR I:=1 TO 3 DO
  BEGIN
    IF INTTABLE[I]=DATA THEN FLAG:=TRUE
  END
END;
BEGIN
  TABLE[1]:=82;
  TABLE[2]:=103;
  TABLE[3]:=7;
  READLN(DATA);
  SEARCH(TABLE,RESULT);
  WRITELN(RESULT)
END.
```

In the first part of the program, an array (TABLE) of three integer elements is declared. The array type is declared in the TYPE section as ARTABLE because it will be needed again in the procedure parameter declarations. DATA is an INTEGER variable where the value to be searched for will be stored. RESULT is a Boolean flag to show whether the element has been found or not.

The procedure used to search for the element in the array is called SEARCH. The array is passed to the procedure as a pass-by-value parameter (INTTABLE). FLAG is used as a pass-by-reference parameter. If it's changed in the procedure, the corresponding variable (RESULT) is updated accordingly. A local variable to the subprogram is I.

The procedure searches through the array for the value entered (DATA). Note that DATA is used as a global variable. It is not passed specifically to the

procedure. The variable I is used to cycle through the array. If the element is found, FLAG (RESULT) will be TRUE. Otherwise, it will be FALSE.

The main program simply initializes the array with three values, reads an integer number, and calls the procedure to search for the element; then it prints the result of the call.

From Procedures to Functions

As you can see from the previous descriptions of procedures, Pascal offers you a very powerful way to write subprograms to be used by the main program. Values may or not be changed in the main program, they may be passed directly to the procedure, or they may be declared in the procedure itself, and so forth. But, often, you'll want only to calculate a value with a subroutine. If that's the case, using a function may be a better choice.

When you want to calculate the square root of a number in Pascal, you can use the standard function SQRT. Assuming X and Y are variables, you can write

```
X:=SQRT(Y)
```

and the square root of Y will be calculated and stored in X. In this example, we've used the standard Pascal function called SQRT. It's standard because it comes defined by Pascal itself. Several of these predefined functions exist—SIN, SUCC, and ROUND, among others. These functions always involve some kind of parameter and return a value as a result. This value is returned stored in the function name. For instance,

```
Y:=SUCC(X)
```

Here, SUCC(X) will return the corresponding value. The difference between a function and a procedure is that, while a procedure achieves certain results when called, a function always returns a value stored in the function name. In this function example, three function calls are made on the same line:

```
RESULT:=SIN(SQR(A)+SQRT(B))
```

Now that you've seen how standard functions are called and used (we discussed most of the standard functions in previous chapters), let's write our own function. Our first example will use an integer variable (A) as a parameter. Our function will be called POWER, and it will use two parameters. Both parameters must be integer values (the second positive), and the result of the function will be another integer number, the value resulting from raising the first number to the power of the second. The function header is very similar to a procedure header, but the type of the function itself must also be declared, since the value resulting from the operation will return in the function name.

The **FUNCTION** keyword is used to denote a function declaration:

```
FUNCTION POWER(NUM,ELEV:INTEGER):INTEGER;
```

This declaration might look a bit peculiar, but the syntax is quite straightforward. The name of our function is POWER, and the parameters are specified in parentheses after the function name—just like a procedure header. The function header ends with a colon and the variable type of the function itself. The resulting value of POWER will also be an INTEGER number.

The function is called later in some kind of expression—not simply by its name as with procedures. If you want to store in variable C the result of raising A to the power of B, you can call the function POWER as follows:

```
C:=POWER(A,B)
```

Here's the complete function:

```
FUNCTION POWER(NUM,ELEV:INTEGER):INTEGER;
VAR
  I:1..MAXINT;
  ACUM:INTEGER;
BEGIN
  ACUM:=1;
  FOR I:=1 TO ELEV DO
    ACUM:=ACUM*NUM;
  POWER:=ACUM
END;
```

The structure of the function itself is very simple and consists of a loop which multiplies NUM by itself an ELEV number of times. The result of each multiplication is stored successively in ACUM (which in the end will be the result of the operation). Finally, ACUM is copied into POWER, because upon return to the calling program, the result must be stored in the function name. Just as with procedures, functions may have their own local declarations.

It's not possible to use the function name POWER directly in the operations because it is not a variable—though it may look like one. In general, the function name may appear on the left side of an assignment, but not on the right side, where it would be interpreted as a new call to the function. Thus,

```
POWER:=23          is correct
```

But

```
POWER:=POWER*2     is wrong
```

In the second illustration, the computer will think you're trying to call function POWER once again, without any parameters.

Using the function POWER, you could calculate

POWER(3,4)=3*3*3*3(4 times)=81

Remember that the name of the function (POWER) must be used to store the result of the operation; it's in the function name where the result of the operation will be placed. Upon return from the function, the only value which will be remembered by the computer will be the one stored in the function's name. Aside from using POWER in this way, the rest of the subprogram has the same structure as a procedure, with local declarations (variable I used for the loop index variable, and ACUM used in the operations) and the statements of the function enclosed inside the general BEGIN and END.

Here's a complete program that uses the POWER function:

```
PROGRAM EXTRAFCTN(INPUT,OUTPUT);
VAR
  NUM1,NUM2:INTEGER;
FUNCTION POWER(NUM,ELEV:INTEGER):INTEGER;
VAR
  I:1..MAXINT;
  ACUM:INTEGER;
BEGIN
  ACUM:=1;
  FOR I:=1 TO ELEV DO
    ACUM:=ACUM*NUM;
  POWER:=ACUM
END;
BEGIN
  WRITE('ENTER AN INTEGER NUMBER:');
  READLN(NUM1);
  WRITE('TO WHAT POWER SHOULD IT BE RAISED?  ');
  READLN(NUM2);
  WRITE(NUM1,'  TO THE POWER OF  ', NUM2,'  IS  ');
  WRITELN(POWER(NUM1,NUM2))
END.
```

This program simply reads a number and the power to which to raise it; then it prints the operation and the result using our defined POWER function.

Function Evaluation Steps

Aside from replacing the word *PROCEDURE* with *FUNCTION* and declaring the data type of the function itself, both functions and procedures share the same structures and identifier specifications.

Most Pascal systems do not allow you to pass standard function values to a defined function in the function call. Check the Pascal implementation being used for reference.

Briefly, these are the steps followed by the computer when a function is encountered:

1. A position in memory is created to store the result of the function (still undefined), according to the type assigned to the function.
2. The parameters in the function call are assigned to the corresponding parameters in the function header. These parameters must correspond on a one-to-one basis with corresponding types. New positions in memory are created to store the function parameters, since this is local data. The corresponding values, passed from the calling program, are stored in this memory area.
3. Memory is reserved for all local declarations, with values still undefined.
4. All the instructions of the function are executed.
5. All memory used by the function is cleared, and only the resulting value of the function is stored so that it can be accessed by the calling program through the function name.

The EVEN Function

Here's a program that includes a simple Boolean defined function called EVEN which returns TRUE if the integer parameter is an even number or FALSE if it is not (a function similiar to the standard function ODD):

```
PROGRAM FUNCTION2(INPUT, OUTPUT);
VAR I:INTEGER;

FUNCTION EVEN(NUM:INTEGER):BOOLEAN;
BEGIN
  EVEN:=FALSE;
  IF (NUM MOD 2)=0
    THEN EVEN:=TRUE
END;
BEGIN
  WRITELN('ENTER A NUMBER');
  READLN(I);
  WRITELN('THE NUMBER IS   ',EVEN(I))
END.
```

Advantages of Using Subprograms

Functions and procedures help you to split a program into individual, self-contained parts, which can easily be modified and analyzed independently from the program as a whole. These subprograms can be used from any point

in the main program, allowing you to use the same routine at several different points of your program.

In addition, the memory used by all local variables is freed upon return from the routine, thus permitting it to be used by other variables in the program.

Independent program modules also promote teamwork. Different programmers can deal with different program segments individually, without worrying about identifier or memory conflicts. These program modules are then easy to put together to form a complete program.

CHAPTER 9

Using Record Variables

CHAPTER 9
Using Record Variables

Just as an array variable is used to store a number of elements which belong to the same variable type, a *record variable* is used to store a number of elements belonging to different variable types. A record variable differs from an array variable in another aspect: You don't access each individual element with an index value, but rather by using variable names. In other words, a record structure has a general name for all variables in that record structure and an individual name for each element in the record.

Defining a Record Structure

Let's prepare a record to store information about a certain person. We'll store the person's name, age, and sex. We'll use a packed array to store the name, an integer subrange of 1–120 for the age, and a character variable for the sex (M or F). The record variable containing these three items of data will be called PERSON, and it will have the following structure:

	NAME	**(string:packed array variable)**
PERSON	**AGE**	**(integer subrange variable)**
	SEX	**(character variable)**

The record structure is declared in the TYPE declaration by declaring the record name, its components, and the variable type of each element in the record. Returning to our previous example, PERSON would be declared like this in a program:

```
TYPE
  PERSON=RECORD
  NAME:PACKED ARRAY[1..15] OF CHAR;
  AGE:1..120;
  SEX:CHAR
  END;
```

The RECORD keyword acts like a BEGIN, so each record declaration needs an END. The layout and indention of the record declaration varies; it's also common to write it like this:

```
TYPE
  PERSON=
    RECORD
     NAME:PACKED ARRAY[1..15] OF CHAR;
     AGE:1..120;
     SEX:CHAR
END
```

Both examples are equivalents. In each case, that particular record structure is called PERSON, with a 15-element packed array (string)—NAME—to store the person's name, an integer subrange—AGE—to store the age, and a character variable—SEX—to indicate the sex.

Declaring and Using Record Variables

Once the structure of the record has been defined, the next step is to declare the variables which will have the structure. This is done in the VARiable declaration section in the usual way. Simply use the name given to the record structure as the variable type:

```
VAR
  FIRST,LAST:PERSON;
```

Here, variables FIRST and LAST will have the record structure defined as PERSON. To access an element of the record, write the name of the record variable being used (FIRST or LAST), followed by a period and the component in the record structure definition (NAME, AGE, or SEX) which you wish to access. Here are some assignments you might use in our previous example:

```
FIRST.NAME:='JACK JONES      '
FIRST.AGE:=25
FIRST.SEX:='M'
```

Or, using the record variable LAST:

```
LAST.NAME:='MARY SMITH       '
LAST.AGE:=30
LAST.SEX:='F'
```

Remember, as defined, the NAME field must have 15 characters.

Thanks to the use of a record, you can store all the information for one person in a single variable (FIRST or LAST). Other advantages and uses of record variables will become clear later.

As with other data structures, a record may be declared directly in the VARiable section of the program:

```
VAR
  FIRST,LAST:RECORD
        NAME:PACK ARRAY [1..15] OF CHAR;
        AGE:1..120;
        SEX:CHAR
    END;
```

Copying Values

Here is one way to assign the values to another variable:

```
SECOND.NAME:=FIRST.NAME;
SECOND.AGE:=FIRST.AGE;
SECOND.SEX:=FIRST.SEX;
```

This is an inefficient way to copy a record variable, especially if the record structure has several elements. It's similar to copying an array element by element.

To copy a record variable, just use the record identifier:

```
SECOND:=FIRST;
```

In this example each element of the record variable FIRST will be copied to the corresponding element of the record variable SECOND.

When copying records, you must declare both records with the same base type declarations. If the declarations are made separately (even if they are the same in structure), an assignment statement will be invalid.

Combining Arrays and Records

Arrays and records can be combined. You can define an array of records. That is, each element of the array has a record structure. Continuing with our original record definition, you can declare a 30-element array to store the data about 30 different people:

```
TYPE
  PERSON=
    RECORD
     NAME:PACKED ARRAY[1..15] OF CHAR;
     AGE:1..120;
```

```
    SEX:CHAR
  END;
VAR
  PEOPLE:ARRAY[1..30] OF PERSON
```

Here, PEOPLE is defined as an array variable of 30 elements, where each element has the record structure defined as PERSON. You may then write, for example,

```
PEOPLE[1].NAME:='JOHNNY LONG      '
PEOPLE[1].AGE:=20
PEOPLE[1].SEX:='M'
PEOPLE[2].NAME:='LAURA WILSON     '
...
...
...
```

The WITH Keyword

Often, you'll be working with one element of a record structure repeatedly or with several elements of the record in a small part of your program. For example, using one person's data, you might write the following line:

```
IF PEOPLE[1].AGE<30 AND PEOPLE[1].SEX='M' AND
  PEOPLE[1].NAME='SMITH' THEN...
```

Here, all three conditions must be true before the statement will be executed. All variables in the conditions belong to a record named PEOPLE[1]. You can make the program easier to read and understand by using the **WITH** keyword in the following way:

```
WITH PEOPLE[1] DO
  IF AGE<30 AND SEX='M' AND NAME='SMITH' THEN...
```

The WITH keyword indicates that the variables of a certain record in the following statement(s) will be written omitting the record name. The general syntax is

WITH *record name* **DO**
  statement

Multiple statements can be included in the WITH structure by enclosing them within the keywords BEGIN and END. In that case, the structure for WITH is

WITH *record name* **DO**
BEGIN
  statements
END

Using Variable Records

In the record examples discussed so far, the structure of the records defined is always rigid; they're always the same wherever they're used in the program. A variable record structure, on the other hand, allows certain flexibility in a record's format. Let's illustrate this point.

Our example program asks the new members of a computer club a series of questions. The first five questions are general to all the users. The sixth question is for tape users only, and questions 7, 8, and 9 are for disk users only (the program assumes members either use disk or tape, but not both). These are the questions that will be asked:

1. NAME?
2. COMPUTER USED?
3. STANDARD RESIDENT MEMORY?
4. PRINTER? (Y/N)
5. CASSETTE/DISK (C/D)
6. CASSETTE PLAYER MODEL? (only tape users)
7. MAXIMUM DISK MEMORY IN K (only disk users)
8. DUAL DRIVE? (Y/N) (only disk users)
9. DRIVE MODEL? (only disk users)

If a user has a disk drive, question 6 is unnecessary. And, if the user stores programs on tape, questions 7, 8, and 9 may be omitted. Although you could use a fixed record to store this information (ignoring certain fields depending on the storage device owned by the user), or you could use two separately structured records, a better technique is using a variable record. What is needed is a record that adopts one shape if the user saves programs on tape and another if disks are used. Now, look at the following record definition:

```
TYPE
  DEVICE=(CASSETTE,DISK);
  MEMBER=
    RECORD
      NAME:PACKED ARRAY[1..30] OF CHAR;
      COMPUTER:PACKED ARRAY[1..10] OF CHAR;
      MEMORY:1..512;
      PRINTER:CHAR;
      CASE STORAGE:DEVICE OF
          CASSETTE:(DEVNAME:PACKED ARRAY[1..15] OF CHAR);
```

```
    DISK:(DISKMEM:INTEGER;
      DUAL:CHAR;
      DMODEL:PACKED ARRAY[1..15] OF CHAR)
  END
```

Here's how this record declaration works. First, before declaring the record, a type list of possible devices (cassette or disk) is defined—DEVICE. This will be used in the record definition. Next, the record structure MEMBER is declared. The first four elements of the structure are fixed because they are questions asked of all new members.

Now look at the way the CASE selector is used in the next section of the record. In several ways, this CASE is similar in structure to the program CASE discussed in Chapter 5. It will branch to the declaration section of the record according to the value of the expression following the keyword CASE (STORAGE, in the example). Thus, the key to the use of variable records is the line

```
CASE STORAGE:DEVICE OF
```

When the computer comes across this line, it checks the value of PERSON.STORAGE and branches to the corresponding label (CASSETTE or DISK), just as in the program CASE. Note that STORAGE has been defined in the record structure directly in the CASE statement and that the value can only be CASSETTE or DISK, as specified in the TYPE definition of DEVICE.

The computer chooses the structure which will be given to the rest of the record according to the declarations following the label chosen by the CASE selector.

In the example, if the value of PERSON.STORAGE is CASSETTE, the program will add the following variable to the record:

```
DEVNAME:PACKED ARRAY[1..15] OF CHAR
```

On the other hand, if PERSON.STORAGE is DISK, the following three fields will be added to the record:

```
DISKMEM:INTEGER;
DUAL:CHAR;
DMODEL:PACKED ARRAY[1..15] OF CHAR
```

The CASE selector used in records is similar to the normal CASE statement, but it's not the same. It does not require a closing END statement to the CASE (because the END of the record also closes the CASE), and, following each label, all variable fields for that label are enclosed within parentheses, as in this example:

```
CASSETTE:(DEVNAME:PACKED ARRAY[1..15] OF CHAR);
DISK:(DISKMEM:INTEGER;
  DUAL:CHAR;
  DMODEL:PACKED ARRAY[1..15] OF CHAR)
```

All of the fixed part of a variable record must *always* come before the variable part. Any declaration after the CASE line will be considered part of the variable section of the record.

You may assign one record structure to several identifiers by separating the labels by commas, as with the normal CASE statement:

```
BLACK,BROWN,BLUE:(assignment)
```

If a label of the variable record CASE has no fields, just follow it by a pair of empty parentheses:

```
PRINTER:( )
```

Variable CASEs may be nested. This means that if you divide a record structure with a CASE statement, any of these subdivisions may be further divided by another CASE statement. You should try not to complicate the record structure too much, however.

```
CASE STORAGE:DEVICE OF
    {branch according to STORAGE}
CASSETTE:(CASE SPEED:SPDEF OF
    {branch according to SPEED}
TURBO:(...);
STANDARD:(...));
DISK:(.....)
```

Program Example

The following program offers you a list of these four operations:

1. Enter a name in a list
2. Search for a name in a list
3. List all entries
4. End

Each entry in the table has three data items, stored in a record structure: the name (a packed array), the age (an integer subrange), and the sex (a character variable).

The program uses two procedures to operate. One prints the option list and reads the choice made by the user; the second is used to read a string variable, the name in each entry.

The first variable in the main program is INDEX. It is initialized to 1, the first element to be entered in the table. The option list is then printed, and while the choice is not 4 (end), the CASE statement is used to execute one of the three main operations. Finally, the options are printed again, and control continues in the WHILE loop.

```
PROGRAM TABLES(INPUT,OUTPUT);
TYPE
  PERSON=RECORD
    NAME:PACKED ARRAY[1..20] OF CHAR;
    AGE:1..120;
    SEX:CHAR
  END;
VAR
  ENTRY:ARRAY[1..20] OF PERSON;
  I,INDEX:INTEGER;
  CHOICE:1..4;
  DATA:PACKED ARRAY[1..20] OF CHAR;
PROCEDURE OPTIONS(VAR CH:1..4);
BEGIN
  WRITELN('(1) ADD ENTRY');
  WRITELN('(2) FIND ENTRY');
  WRITELN('(3) LIST TABLE');
  WRITELN('(4) END');
  READLN(CH)
END;
PROCEDURE READNAME;
VAR
  J:1..21;
BEGIN
  J:=1;
  READ(DATA[J]);
  WHILE (J<21) AND (NOT(EOLN)) DO
  BEGIN
    J:=J+1;
    READ(DATA[J])
  END
END;
BEGIN
  INDEX:=1;
  OPTIONS(CHOICE);
  WHILE CHOICE<4 DO
  BEGIN
    CASE CHOICE OF
```

```
      1:BEGIN
        WRITE('ENTER NAME:');
        READNAME;
        ENTRY[INDEX].NAME:=DATA;
        WRITE('ENTER AGE:');
        READLN(ENTRY[INDEX].AGE);
        WRITE('ENTER SEX:');
        READLN(ENTRY[INDEX].SEX);
        INDEX:=INDEX+1
      END;
      2:BEGIN
        WRITE('NAME TO BE FOUND:');
        READNAME;
        WRITELN;
        FOR I:=1 TO INDEX-1 DO
        BEGIN
         IF DATA=ENTRY[I].NAME THEN
         BEGIN
         WRITELN('NAME:',ENTRY[I].NAME);
         WRITELN('AGE:',ENTRY[I].AGE);
         WRITELN('SEX:',ENTRY[I].SEX)
         END
        END
      END;
      3:BEGIN
        FOR I:=1 TO INDEX-1 DO
        BEGIN
         WRITELN('NAME:',ENTRY[I].NAME);
         WRITELN('AGE:',ENTRY[I].AGE);
         WRITELN('SEX:',ENTRY[I].SEX);
         WRITELN
        END
      END
    END;
    OPTIONS(CHOICE)
END
END.
```

Please note that 20 is the maximum number of entries. A routine to clear all packed arrays (to make them equal to a string of spaces of maximum length) should be added if the strings are not cleared automatically.

CHAPTER 10

Files

CHAPTER 10
Files

In all the examples so far, we have been using variables with values defined from within a program or read from the keyboard. In addition, all output has been sent to the screen. Standard Pascal permits the use of external files to store information, but it limits you to sequential files.

Declaring Your File

As with all Pascal structures, files must be declared in the first part of a program. All file variables are added in the program header, following the program name in parentheses. The files named INPUT and OUTPUT are default files which get data from the keyboard and send data to the screen. Any other files which will be used must have file variables in the program header. If your program uses a file named DOGDATA, the program header might look like this:

```
PROGRAM FILES(INPUT,OUTPUT,DOGDATA);
```

The program header tells Pascal what the name of the program is and how many channels will have to be opened while the program executes. DOGDATA's file type has not been specified; Pascal has simply been told that a file will have to be opened which will be known as DOGDATA. In standard Pascal, there is no way to tell Pascal what file you want to use, so it defaults to the name of the file variable. Therefore, the file variable must be the name of your file. This means that if you want to use more than one file, you must put a file variable in the program header for every file you access. This can be cumbersome if you're working with several files, but need only a few of them to be open at any one time. Several implementations of Pascal remedy this problem by allowing you to specify the filename in the RESET statement or in a separate OPEN statement. If your version of Pascal has this feature, all you need to do is put a file variable in your program header for each file that you will have open at one time.

The next step in the declarative process is to declare the file type by using the keywords **FILE OF**. The INPUT and OUTPUT files are the exception.

They are predefined by Pascal and are not included in the program declarations. Disk files may have any structure as their data type. Thus, you may have a

```
FILE OF INTEGER
```

or a

```
FILE OF PEOPLE
```

where PEOPLE is a previously declared data structure—for example, a record.

The following program segment shows how to declare a file with a record structure:

```
TYPE
  MEMBERS=
    RECORD
     NAME:PACKED ARRAY[1..30] OF CHAR;
     AGE:0..130
    END;
VAR
  MEMFILE:FILE OF MEMBERS
```

This program segment declares MEMFILE to be a file containing records of type MEMBERS. The file variable itself is declared in the VARiable section of the program declarations. The program header would have to include the file variable MEMFILE:

```
PROGRAM USERGROUP(INPUT,OUPUT,MEMFILE);
```

Once the file variable and its structure have been declared, you can use the file variable in the program.

Resetting and Reading from a File

You can prepare a file for reading and for writing with two commands: RESET and REWRITE, respectively.

RESET prepares a file to be read and places a pointer to the first element of that file. You must use RESET before you can read a file. It's similar to BASIC's OPEN statement. The following line resets the file ANIMALS, placing a pointer to the first element of the file:

```
RESET(ANIMALS)
```

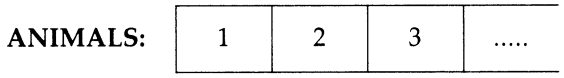
The file variable is in the parentheses following the RESET command.

To read the first element, use the familiar **READ** command, but specify

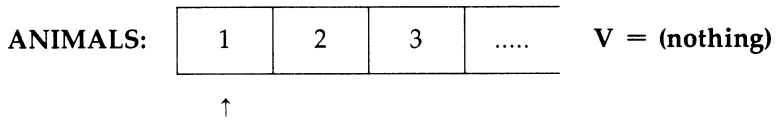
what file you will be reading from and the variable(s) where the values read must be stored:

```
READ(ANIMALS,V)
```

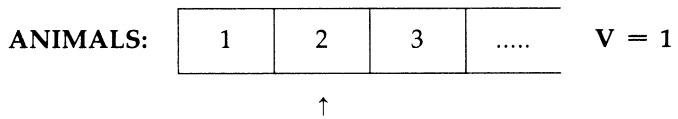
Variable V must be compatible with the data read from file ANIMALS, or an error will occur. READ assigns the value currently pointed to in the file to the corresponding variable and advances the pointer to the next element of the file. Graphically, here are the conditions of file ANIMALS after each command has executed:



RESET(ANIMALS) prepares the file to be read, placing a pointer to the first element to be read from the file.



Before READ(ANIMALS,V). The value of V is still not read; the pointer is placed on the element of the file which will be read next.



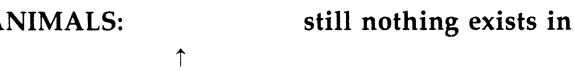
After READ(ANIMALS,V). The value of V has been read from the file, and the file pointer is advanced to the next element in the file (2).

Writing To a File

To prepare a file to be written to, use the REWRITE command. Specify the file inside the parentheses following REWRITE:

```
REWRITE(ANIMALS)
```

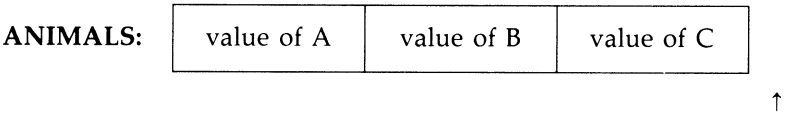
This line will set the pointer to the first position in the file. If the file exists already, any information in it will be written over. REWRITE does *not* append information (add information at the end of existing data) to the file. Here's how the file ANIMALS now looks:



To write a value to the file, use the WRITE command, specifying the file to be written to and the variable(s) to be written:

```
WRITE(ANIMALS,A,B,C)
```

This statement writes the values of variables A, B, and C to file ANIMALS. This is now the situation of file ANIMALS:



In other words, WRITE is used to write a value to a file at the current position of the file pointer; it then advances the pointer to the next position for the next element to be written.

A Simple Example

The following example will input three REAL values from the keyboard (INPUT) and store them to disk in a file called NUMBERS. The message *WRITING* is printed while this is being done. Then the message *READING* is printed, and the values are read back to memory from the file and are printed to the screen (OUTPUT).

```
PROGRAM FILES(INPUT,OUTPUT,NUMBERS);
VAR
  NUMBERS:FILE OF REAL;
  A,B,C:REAL;
BEGIN
  REWRITE(NUMBERS);
  WRITELN('ENTER THREE NUMBERS:');
  READ(A,B,C);
  WRITELN('WRITING...');
  WRITE(NUMBERS,A,B,C);
  WRITELN('READING...');
  RESET(NUMBERS);
  READ(NUMBERS,A,B,C);
  WRITELN(A,B,C)
END.
```

EOF, EOLN, and PAGE

When values are being read from a file, two things can be checked: EOF checks to see whether you've reached the end of the file, and EOLN checks for the end of a line of a text file. These two functions return a Boolean value according

to the current condition of the file being read. When a value has been read from a file, if there are no more elements left to be read in that file, the EOF condition returns the Boolean value TRUE. If there are still elements in the file, EOF returns FALSE. The EOF function uses the file variable of the file you're reading from as a parameter.

EOF(*file—variable*)=TRUE No more elements to be read in the *file*
EOF(*file—variable*)=FALSE More elements to be read in the *file*

The EOLN function tests the end of a line of text data in the same way that EOF tests the end of a file. (EOLN is used with text files, which are discussed in the next section.) When you're reading characters from a file and a RETURN character is found in the file, EOLN will return TRUE; otherwise, it will return FALSE. Just as with EOF, you must use the file variable of the file you're reading from as the parameter of the function.

One last subprogram affects a text file, but this time when it is being written to. **PAGE** sends a form-feed (skips to the start of a new "page" in the text file). PAGE must be followed by the file variable enclosed in parentheses as a parameter.

Text Files

Text files are special kinds of files in Pascal that consist purely of characters. To declare a text file, just specify the file type as the standard type **TEXT**:

```
FILEVAR:TEXT
```

Text files are essentially files of characters (type CHAR). Text is defined by Pascal as FILE OF CHAR. The standard files discussed so far, INPUT and OUTPUT, are text files.

READ, READLN, WRITE, and WRITELN are used to access text files. Even though text files are a series of characters and RETURNs, WRITE, WRITELN, READ, and READLN can directly write/read other variable types correctly. The following data types may be written to and read from a text file:

READ,READLN: CHAR, INTEGER, REAL
WRITE, WRITELN: CHAR, INTEGER, REAL, BOOLEAN, and strings

Here is an example which reads a word entered by the user, saves it in a text file, and then reads it and prints it again. The example is very simple, but it will help you to visualize the file process.

```
PROGRAM FILES(INPUT,OUTPUT,USERWORD);
VAR
  USERWORD:TEXT;
  LETTER:CHAR;
```

```
BEGIN
  REWRITE(USERWORD);
  WRITE('ENTER A WORD:');
  READ(LETTER);
  WHILE NOT(EOLN) DO
  BEGIN
    WRITE(USERWORD,LETTER);
    READ(LETTER)
  END;
```

At this point, the program has read the user's input, letter by letter, and has stored this data in a text file called USERWORD. Notice how the end of the user's input is found by checking the keyboard entry for an EOLN condition. The next step is to read the data in the text file USERWORD once again and print it on the screen:

```
  RESET(USERWORD);
  WRITELN;
  READ(USERWORD,LETTER);
  WHILE NOT(EOF(USERWORD)) DO
  BEGIN
    WRITE(LETTER);
    READ(USERWORD,LETTER)
  END
END.
```

The second part of the program prepares the file USERWORD so that it can now be read from. Then it enters a loop that reads characters from the file and prints them to the screen until the end of file USERWORD is found. Program execution then ends.

The INPUT and OUPUT Text Files

Two files automatically defined in Pascal are INPUT and OUTPUT; whenever a filename is omitted in a file instruction, INPUT or OUTPUT is assumed. Both INPUT and OUTPUT are text files. In other words, all the times you have used statements like

```
READLN(variable)
```

or

```
WRITE('HELLO')
```

the computer has assumed that the statements are

```
READLN(INPUT,variable)
```

and

```
WRITE(OUTPUT,'HELLO')
```

When you use EOF, EOLN, or PAGE without file parameters, they're directed to the INPUT file (EOF, EOLN) or the OUTPUT file (PAGE). This means that you can use EOLN to check for the end of the user's input (as we have done in some examples):

```
WHILE NOT(EOLN) DO READ(X)
```

This statement will read values for X until the user has finished input by pressing RETURN.

Finally, the command PAGE without parameters or PAGE(OUTPUT) will clear the screen.

GET and PUT: Using Buffers

When you use READ and WRITE to store and retrieve data information from a file, you're really executing two different operations. In the case of reading a file, you're retrieving the data currently pointed to by the file pointer and advancing the file pointer to the next data item to be read. When you're writing a value, a value is being written to the current position pointed to in the file, and the file pointer is advancing to the next WRITE position in the file. The READ and WRITE operations can be separated into two parts: reading (or writing) and moving the pointer. This is done by using the GET and PUT procedures and file buffers.

Every file has an associated buffer area of the same type as the base type of the file. Data read from or written to the file is temporarily stored in this buffer area. The area is set aside every time a file is opened for reading or writing, and the name of the buffer area is the same as the file variable followed by an up-arrow. Thus, if your file variable is CLUBDATA, the associated buffer created by the computer when the file is opened will be called *CLUBDATA*↑. The READ operation can be separated into two parts:

```
DATAVAR:=CLUBDATA↑
```

This line copies the current value in the buffer into the data variable, and

```
GET(CLUBDATA)
```

advances the file pointer and reads the next item of information from the file CLUBDATA into the associated buffer.

These statements are also valid the first time a file is opened for reading because, when a file is opened and the buffer area is prepared, the first data item from the file is already placed in the buffer. If a file is called WORD (a

text file) and has the word *HELLO* stored, then, when this file is prepared for reading with RESET(WORD), the letter *H* will be copied into the associated buffer WORD↑. The buffer can be copied into a variable. The file pointer is then advanced (thus reading the next file value into the buffer) with the statement GET(WORD).

`RESET(WORD);`	Prepares text file WORD for reading and places letter *H* into the buffer. If RESET is executed again on the same file without the file being closed first, the buffer pointer is RESET to the beginning of the file.
`MYVAR:=WORD↑;`	Copies *H* into variable MYVAR.
`GET(WORD);`	Moves the file pointer forward one position and copies the letter *E* into buffer WORD↑.

The READ operation joins the two steps (reading a value and moving the pointer) into one operation by reading the data from the buffer into the variable and advancing the pointer.

```
READ(WORD,MYVAR);
```

is equivalent to

```
MYVAR:=WORD↑;
GET(WORD);
```

Also, READLN(file) is equivalent to

```
WHILE NOT(EOLN(file)) DO
  GET(file);
```

In the same way, the WRITE operation can be divided into two steps: copying data into the file buffer and copying that buffer to the file, moving the file pointer to the next position in the file. Thus,

```
WRITE(WORD,MYVAR);
```

is equivalent to

```
WORD↑:=MYVAR;
PUT(WORD);
```

Though most of the time you'll be likely to use READ or WRITE for your file operations, GET and PUT will be useful, as in this example which reads a character from FILE1 and copies it into FILE2:

```
READ(FILE1,LETTER);
WRITE(FILE2,LETTER);
```

Using file buffers directly, you don't need the auxiliary variable LETTER to copy the file:

```
FILE2↑:=FILE↑1;
PUT(FILE2);
GET(FILE1);
```

If, after reading a value from a file, EOF is TRUE (the end-of-file is reached), the value stored in the file buffer will be undefined. If EOLN is TRUE, the character in the buffer will be a space.

Buffers will be useful when the READ or WRITE operation has to be divided into two different steps as described above.

Extensions

By using the instructions discussed in this chapter, you can access most external devices. According to the implementation used, RESET, REWRITE, and so forth, let you access other devices, such as a printer. Because access varies widely according to the Pascal system and computer that you're working with, you'll need to check the specific material for your system. Other extensions are discussed in Chapter 14.

Files are one of the areas in Pascal that have great variations from system to system, since most implementations have added commands for extra functions not provided by standard Pascal. This chapter, then, has been limited to information, instructions, and formats that belong to standard Pascal and which should be recognized by most systems. Check the reference manuals for the implementation you are using to find out whether additional external device access commands are provided.

CHAPTER 11

Recursion

CHAPTER 11
Recursion

Pascal allows functions and procedures to call themselves. This technique is known as *recursion.* Every time a routine calls itself, a new set of local variables is created. This is what makes recursion possible in Pascal, since each level of local variables is stored separately. Recursive routines are useful in solving several programming problems.

A Function That Calls Itself

Let's write a program function that calculates the factorial of a positive integer number. This value is found by multiplying the number by all integers between 1 and the number. If the number entered is 5, the factorial is

5 * 4 * 3 * 2 * 1 = 120

In particular, the factorial of 0 is 1. The only parameter that the function will need is the integer value for which you want to find the factorial. Here is the function coded by using recursion:

```
FUNCTION FACT(NUM:INTEGER):INTEGER;
BEGIN
  IF NUM=0 THEN FACT:=1
  ELSE FACT:=NUM*FACT(NUM-1)
END;
```

Look at how the function is structured. If NUM is equal to 0, the factorial of NUM is equal to 1. Otherwise, the number entered (NUM) is multiplied by the factorial of NUM−1, the next lower integer number, and assigned to FACT. In other words, the function FACT is calling itself with a parameter which is a unit smaller than before. It will continue calling itself until that parameter is 0, simplifying the routine at every call. To understand how recursion works, let's follow the routine's operation, supposing that NUM is equal to 3.

Since NUM is not 0, the routine calls itself again, with a value of 2, leaving the operation it was performing until it has calculated the factorial of 2—FACT(2). Thus, the computer leaves the following operation on "hold":

```
FACT:=3*FACT(2)
```

Then it proceeds to execute the function call FACT(2). The program reenters FACT with NUM now equal to 2. Since NUM is still not 0, the computer executes the following operation:

```
FACT:=2*FACT(1)
```

Again, this operation is left alone for the moment, and the computer continues to work on FACT(1). Reentering the function, the same thing happens:

```
FACT:=1*FACT(0)
```

The computer reenters FACT with a parameter of 0, and here the situation changes. NUM is now 0, so FACT is assigned 1 and the function ends. When the END of the function is encountered, the computer returns to the previous function call, finding the value of FACT which was on "hold":

```
FACT:=1*FACT(0):=1*1:=1
```

When this call is completed, the END of this level of the function is found, and the computer returns to the previous call and calculates:

```
FACT:=2*FACT(1):=2*1:=2
```

Again END is found, and the computer returns to the previous call:

```
FACT:=3*FACT(2):=3*2:=6
```

And the function returns with the correct value stored in FACT. By calling itself recursively, the function uses itself to calculate each step of the operation to find the factorial of NUM. This recursive routine is possible only because the computer remembers the values of the variables it is dealing with at each function call.

Drawbacks of Recursion

In many cases recursion is a natural option; however, in most instances it can be replaced by a program loop that will achieve the same results. A loop is actually preferable, because recursive routines have one major drawback—the amount of memory they consume.

Every time a routine calls itself, a new group of variables is created and a new entry will exist in Pascal's return stack. Calling a routine recursively too many times will quickly cause an OUT OF MEMORY message (due to excessive defined variables) or a STACK OVERFLOW message (due to excessive subprogram calls in the return stack). Recursive routines also execute more slowly than do equivalent nonrecursive routines because of the overhead involved in function calls.

CHAPTER 12

Dynamic Data Structures

CHAPTER 12
Dynamic Data Structures

As you saw in Chapter 7, arrays have one severe limitation: You must declare the size of the array (the maximum number of elements) at the beginning of the program, thus making it a fixed value throughout execution. Attempting to use more than the maximum number of elements results in an execution error. Using fewer elements means memory wasted. In other words, when you don't know how many elements will be stored in an array, you'll most likely end up wasting memory dimensioning a large array.

Arrays also pose a problem when you want to insert (or delete) an element into (from) the middle of the array. All the elements following the one added or deleted have to be shifted up or down, respectively, in the array.

Pascal offers the programmer a solution to all these problems by allowing the use of *dynamic data structures*—that is, using *pointers* to access any particular element of a structure indirectly. The variables we have discussed so far have been static variables. Static variables are assigned a fixed size at compile-time; that is why the declaration section is at the beginning of every Pascal program. Dynamic data structures differ from static variables in that only the pointer which will point to the structure has its size fixed at compile-time. In this chapter, you'll learn how to use linked lists, and more complicated structures like *trees* will be introduced.

For programmers who are new to the idea of dynamic data structures and pointers, all these explanations may seem a bit vague at first. If you'll read through the first sections completely once and then carefully reread the parts that are unclear, you should get a general understanding of the concept behind linked lists and pointers. Once the concept is clear to you, the chapter should be more easily understood.

Pointers

A *pointer* is an element that can "point" to any value stored in memory. It must be declared just like any other variable type, like INTEGER or REAL, but

its declaration is a little different from the types discussed in previous chapters. The following is a pointer declaration:

```
TYPE
  POINTER=↑INTEGER
```

This means that any variable declared as POINTER will be considered a pointer to an integer value in memory. POINTER is defined as a pointer type because of the up-arrow preceding the word *INTEGER* in the declaration. If you declare a variable belonging to the type called POINTER, this pointer variable will be used to point to an integer value in memory. A pointer variable might be declared as follows:

```
VAR
  ARROW:POINTER
```

Now ARROW is a pointer to an integer value.

If ARROW is a pointer variable, the object pointed to by ARROW can be referenced by ARROW↑ (the pointer variable followed by an up-arrow). This should not be confused with the up-arrow used to identify the file buffer, explained in Chapter 10. If a pointer variable is undefined—in other words, if it doesn't point to any element—you should make it equal to the keyword **NIL**. The statement

```
ARROW:=NIL
```

makes ARROW point to nothing, to no element in memory (this is similar to making a variable equal to zero).

Thus, ARROW is the pointer and ARROW↑ is what is being pointed to by ARROW. Graphically:

(pointer)

OBJECT=ARROW↑

If ARROW1 and ARROW2 are pointers, the assignment

```
ARROW1:=ARROW2
```

makes ARROW1 point to the same element as ARROW2 (Figure 12-2). If you write

```
ARROW1↑:=ARROW2↑
```

you're copying the element pointed to by ARROW2 into the position pointed to by ARROW1. The two pointers will still be pointing to the same positions; you've only made a copy of the elements being pointed to (Figure 12-3).

Figure 12-1. *Each pointer points to a corresponding element in memory.*

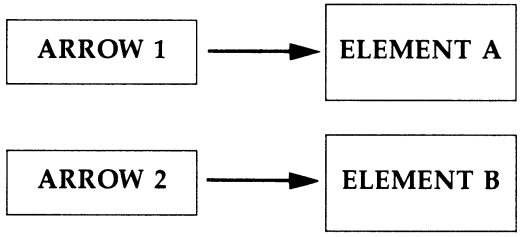
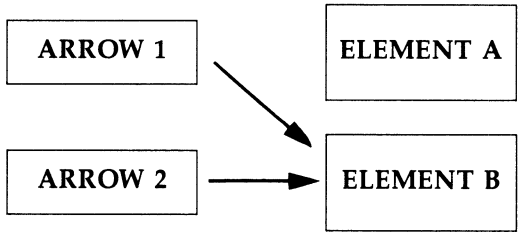
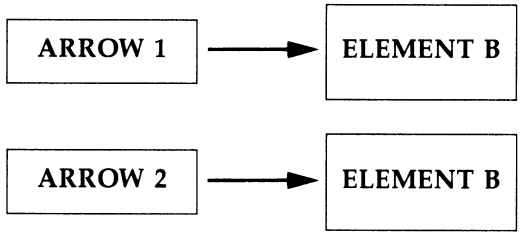


Figure 12-2. *ARROW1:=ARROW2 makes the pointers equal; thus, ARROW1 points to the same element as ARROW2 (ELEMENT B).*



Unless you've set another pointer to point to ELEMENT A, this value will be lost.

Figure 12-3. *ARROW1↑:=ARROW2↑ makes a copy of the element pointed to by ARROW2 into the position pointed to by ARROW1. The pointers still point to their respective memory locations.*



It's important to understand the difference between a pointer and the thing pointed to by the pointer. In the previous assignments, note how similar the statements look and how different the results are.

Pointer variables may be compared if they belong to the same type, and any pointer may be compared to the keyword NIL.

NEW and DISPOSE

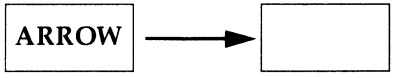
Once you have the pointers, you'll need to create a place in memory for the values you want to store. You can do this with the standard procedure **NEW**. This procedure uses a pointer variable as a parameter. Its effect is to find a place in memory for the element referenced by the pointer. The address found for the value is placed into the pointer variable. Continuing with our previous example, where POINTER was defined as a pointer to INTEGER values and ARROW was our pointer variable:

```
TYPE
  POINTER=↑INTEGER;
VAR
  ARROW:POINTER
```

the call

```
NEW(ARROW)
```

will find a place in memory large enough to store an integer value and will make ARROW point at that position in memory. NEW also marks that memory location as being in use so that the location will not be overwritten by some other value. Graphically:



(place in memory for an integer value)

To place that integer value in memory, you simply assign it to ARROW↑. The value will be stored in the corresponding position in memory. Thus, the instruction

```
ARROW↑:=23
```

will place the value 23 into the memory location pointed to by ARROW. Graphically:

If you forget to include the up-arrow after the variable name, ARROW, the value 23 will not be stored in the location pointed to by ARROW. In fact, ARROW itself will be changed to point to memory address 23—not at all what you had in mind.

Once a position in memory is no longer needed, that place can be "freed" so that other data can use it. The procedure **DISPOSE** lets you do this. Its effect is opposite that of NEW. The instruction

```
DISPOSE(ARROW)
```

frees the space in memory pointed to by ARROW (losing any data stored there) so that it can be used by other data.

Obviously, an error condition will occur when you try to access a memory position (like ARROW↑) without first having created a place in memory with the NEW procedure, or when you try to access this position after a DISPOSE of that memory position has been executed. Setting the pointer values of any pointers you no longer plan to use to NIL will prevent your program from trying to access memory locations that are no longer meaningful.

A Simple Example Using Pointers

Before discussing linked lists and other dynamic structures, let's look at a simple program that uses pointers:

```
PROGRAM POINTERS(INPUT,OUTPUT);
TYPE
  POINTER=↑INTEGER;
VAR
  ARROW1,ARROW2:POINTER;
  ANSWER:INTEGER;
BEGIN
  NEW(ARROW1);
  ARROW1↑:=12;
  NEW(ARROW2);
  ARROW2↑:=2;
  WRITE(ARROW1↑,'*',ARROW2↑,'=');
  ANSWER:=ARROW1↑*ARROW2↑;
  WRITELN(ANSWER);
  DISPOSE(ARROW1);
  DISPOSE(ARROW2)
END.
```

This program simply multiplies two integers, using pointers to reference the numbers instead of variables. First, the program declares a pointer type, POINTER, to point to INTEGER data. Two pointer variables, ARROW1 and ARROW2, and a normal INTEGER variable are defined. The program section prepares places in memory for the value. ARROW1 points to 12, and ARROW2 points to 2.

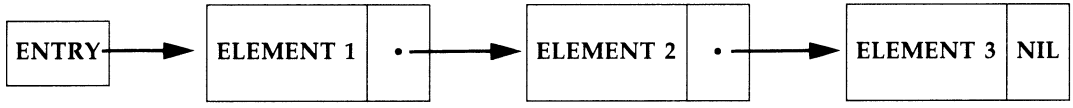
Next, the program prints the problem, calculates and prints the answer, and finally frees the memory occupied by the integer values.

Two interesting points should be mentioned about this program. First of all, at no point in the program have variables been used to store the two integer values 12 and 2. These values have been referenced indirectly throughout by using the pointers ARROW1 and ARROW2. Second, when the memory was freed, it immediately became usable by the system. Thus, the only memory which really has been used corresponds to the storage of ANSWER, the integer variable used to store the answer of the operation and that which held the pointer itself.

Defining the Corresponding Data Structure

Pointers are most useful when they're used to store variable-length lists of information in memory. Records are the most convenient structure to use with pointers. In each record, you can store all the information of an entry and a pointer value to the next element in the list of elements. This linked structure would look something like Figure 12-4.

Figure 12-4. Linked Structure



That is, each element will point to the following element in the list, and the last element will point to nothing (NIL). A pointer will show the beginning of the data list (ENTRY).

To store the name and age of a group of people in a list, you might use the following record structure for each person:

```
TYPE
  PERSON=
    RECORD
      NAME:PACKED ARRAY[1..10] OF CHAR;
      AGE:0..130;
      LINK:POINTER
    END;
```

In this record, a packed array stores the name, and an integer-subrange variable stores the age. LINK is a pointer which will point to the next entry of the data list. LINK is declared a pointer type, as mentioned before, and this

data type (POINTER) must be declared before the record structure where the pointer will be used. Completing the TYPE declarations:

```
TYPE
  POINTER=↑PERSON;
  PERSON=
    RECORD
     NAME:PACKED ARRAY[1..10] OF CHAR;
     AGE:0..130;
     LINK:POINTER
    END;
```

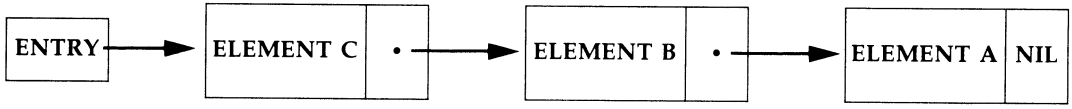
Now it can be said that LINK has been declared as a pointer to a PERSON record structure.

In other words, Pascal allows you to reference the element pointed to before the element has actually been defined (POINTER will point to PERSON even if at that point PERSON is still undefined).

Creating a Linked List

Using the declarations of the record and pointer in the previous section, we'll now proceed to build a linked list. We want to build a list similar to the one in Figure 12-5.

Figure 12-5.



First, let's set up the type declarations and variable declarations for the program. The program uses the PERSON record structure previously declared and two pointer variables, ARROW and ENTRY. These are the declarations:

```
PROGRAM LIST(INPUT,OUTPUT);
TYPE
  POINTER=↑PERSON;
  PERSON=
    RECORD
     NAME:PACKED ARRAY[1..10] OF CHAR;
     AGE:0..130;
     LINK:POINTER
    END;
VAR
  ARROW,ENTRY:POINTER;
```

ENTRY will be used when building the list, and it will always point to the position in the list where the next element will be added. Initially, there are no elements in the list, so ENTRY points to nothing (NIL):

```
BEGIN
  ENTRY:=NIL;
```

Now we'll add an element (the name and age of a person) to the list. The standard procedure NEW is used to create a position in memory for the data entry (Figure 12-6):

```
NEW(ARROW);
```

The next step is to place the data in the corresponding memory addresses. This stage requires some thought. ARROW↑ is actually the record name (PERSON) because ARROW is a pointer to PERSON. This means that ARROW↑.NAME will be the person's name and ARROW↑.AGE will be the person's age. These data elements can be stored in memory with the following lines:

```
ARROW↑.NAME:='ELIZABETH   ';
ARROW↑.AGE:=28;
```

The situation is now as shown in Figure 12-7. The entry point in the list is no longer NIL, so ENTRY must be updated and the pointer will move to the next element in the list. This is done in the next two lines:

```
ARROW↑.LINK:=ENTRY;
ENTRY:=ARROW;
```

Figure 12-6

Figure 12-7

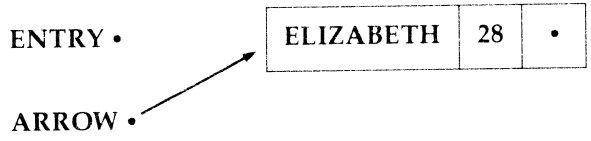


Figure 12-8

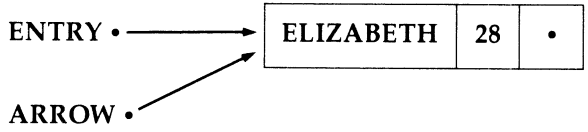


Figure 12-9

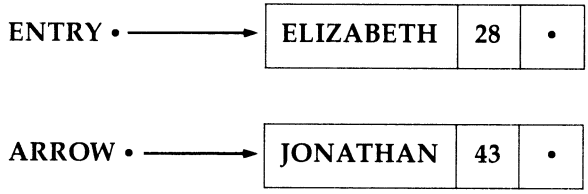


Figure 12-10

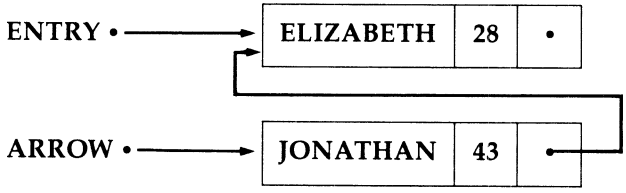
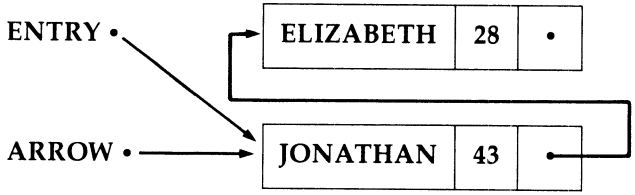


Figure 12-11



Now the situation is as shown in Figure 12-8. The entry point (ENTRY) now points to the first element in the list, and the pointer to the next element in the list points to NIL. Adding a second element to the list uses the same instructions:

```
NEW(ARROW);
ARROW↑.NAME:='JONATHAN  ';
ARROW↑.AGE:=43;
ARROW↑.LINK:=ENTRY;
ENTRY:=ARROW;
```

What has happened? A new element is created with NEW(ARROW). We add the name and age to memory (represented in Figure 12-9). Then the two key lines are found again. The pointer to the next element is pointed to the previous entry in the list (Figure 12-10), and the entry point is now set to this second component of the list (Figure 12-11). Thus, the two lines

```
ARROW↑.LINK:=ENTRY
ENTRY:=ARROW
```

are the key lines to add an element to a linked list. Try adding a third element to the linked list, using the same five lines as above, changing the data entered. Do you see how the linked list builds up? In this way, you can store as many elements in a table as memory will allow, without having declared the size of the structure in the program declarations.

Searching a Linked List

There are a number of different operations that you may want to perform with the data stored in a linked list. First, we'll explore how to search through a list for a certain element. The first element in the list is easy to locate since it's pointed to by ENTRY. Thus, the data required from our previous example with only two entries in the list is

```
ENTRY↑.NAME:='JONATHAN  '
ENTRY↑.AGE:=43
```

This is the data of the last item entered to the list when the list was created. Look at the figures of the list, and you'll see that ENTRY always points to the last element entered in the list (the first element entered points to NIL). The second element in the list can be accessed by the element LINK of the first element.

```
ENTRY↑.LINK↑.NAME:='ELIZABETH  '
ENTRY↑.LINK↑.AGE:=28
```

You can continue accessing each element of the list in this way, but it's clumsy. All you have to do to advance in a list is to make the ARROW pointer point to the next element in the list with the assignment

```
ARROW:=ARROW↑.LINK
```

This makes ARROW point to the next element in a list (because ARROW↑.LINK always points to the next element in a list). You can do this until ARROW returns the value of NIL, which means you have reached the end of the list. The following routine prints all the data of a linked list like the one created in the previous section:

```
ARROW:=ENTRY;
WHILE ARROW<>NIL DO
BEGIN
  WRITELN('NAME=',ARROW↑.NAME,'   AGE=',ARROW↑.AGE);
  ARROW:=ARROW↑.LINK
END;
```

The algorithm to search a list for an element is

```
ARROW:=BASE;
WHILE (ARROW<>NIL) AND (ARROW↑.NAME<>'SMITH       ') DO
ARROW:=ARROW↑.LINK
```

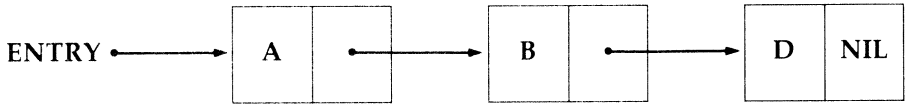
Here, the pointer (ARROW) moves along a list until either the end of the list is reached (ARROW:=NIL) or ARROW↑.NAME is equal to the name being searched for (SMITH, in the example). The routine advances ARROW along the list by making ARROW equal to ARROW↑.LINK (which points to the following element in the list). When execution leaves the WHILE loop, ARROW will either be NIL (the element wasn't found in the list), or it will point to the entry being searched for.

LIFO and FIFO Structures

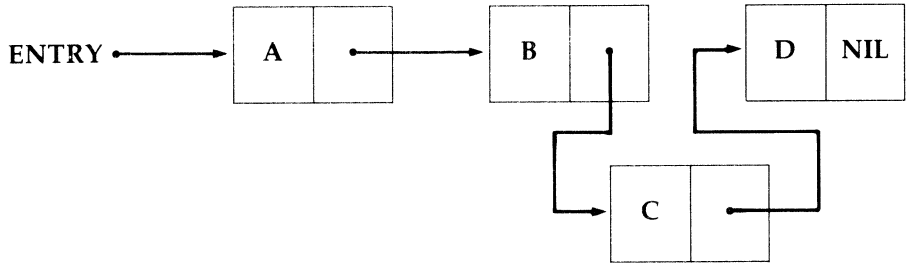
In the list created in the previous sections, the last element entered in the list is the most directly accessible element: simply use ENTRY↑.NAME and ENTRY↑.AGE. This list is called a **LIFO** structure (Last In, First Out), because the last element entered is the most accessible. A stack is a common example of a **LIFO** structure. A list that gives you immediate access to the first element entered in a list is said to have a **FIFO** structure (First In, First Out). A queue is common example of a **FIFO** structure; the person who has been longest in the queue is the first one to leave.

Inserting an Element into a List

Inserting an element into a linked list involves changing the pointers of the element before the one you're inserting and of the one being inserted. If your linked list is like the following:



and you want to insert an element (C) between B and D, the procedure is to make B point to C and to make C point to D:



This means making C.LINK=B.LINK so that C's link now points to D (D now comes after C in the list) and B.LINK=C so that B's link points to C (C now follows B).

Three situations can exist when you're adding elements to a list:

1. No elements exist in the list yet; you must add the element at the beginning.
2. Insert an element between two other elements in the list.
3. Add an element at the end of the list.

You'll need two list pointers to add an element to the list—aside from the pointer to the beginning of the list (ENTRY). Let's declare the data needed to manipulate a dynamic list of integers:

```
TYPE
  POINTER=↑ELEMENT;
  ELEMENT=
     RECORD
      NUM:INTEGER;
      NEXT:POINTER
     END;
VAR
  ENTRY,P,Q:POINTER
```

This declaration prepares a record structure which will be used to store integers (NUM) and a pointer to the next element in the linked list (NEXT). Three pointer variables are declared: ENTRY (to point to the beginning of the list), and P and Q (to move along the list). Following is an ADDELEMENT routine, printed and explained in sections. (It assumes that an integer variable, OURNUM, contains the value to be added to the linked list structure.)

```
BEGIN
  IF ENTRY=NIL THEN
  BEGIN
    NEW(P);
    ENTRY:=P;
    P↑.NUM:=OURNUM;
    P↑.NEXT:=NIL
  END;
```

This first part of the routine handles matters when the list is empty (ENTRY=NIL). A new element position is created with NEW(P), and ENTRY is set to point to the new element. The value to be stored (OURNUM) is placed in the linked list (P↑.NUM:=OURNUM), and the next element in the list is made equal to NIL (P↑.NEXT:=NIL). If the list is not empty, move along it to insert the element in the right position numerically:

```
ELSE
BEGIN
  P:=ENTRY;
  WHILE (P<>NIL) AND (P↑.NUM<=OURNUM) DO
    P:=P↑.NEXT;
```

(P↑.NUM<=OURNUM builds the number list in ascending order; P↑.NUM>OURNUM builds it in descending order.)

When the program leaves this loop, either P=NIL (add the element at the end of the list) or not (add the element in the middle of the list):

```
IF P=NIL
  THEN
    BEGIN
      NEW(Q);
      P:=Q;
      Q↑.NUM:=OURNUM;
      Q↑.NEXT:=NIL
    END
```

To add an element at the end of the list, create a new element using the Q pointer, NEW(Q), and make the last element of the list (P) point to Q instead

of NIL (P:=Q). Then, store OURNUM in Q↑.NUM and make Q↑.NEXT point to NIL, because it's the last element in the list.

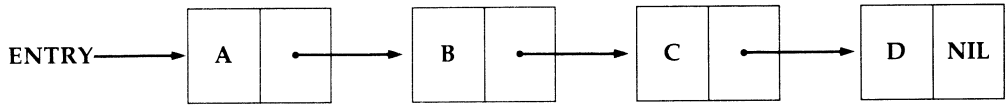
Now we're missing only the part of the routine that adds an element in the middle of the list, between two existing elements:

```
ELSE
BEGIN
  NEW(Q);
  Q↑.NUM:=OURNUM;
  Q↑.NEXT:=P↑.NEXT;
  P↑.NEXT:=Q
END
```

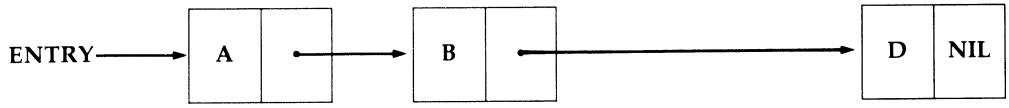
Create a new element using Q—NEW(Q). Then, store the element in Q↑.NUM. We want Q↑.NEXT to point to the same element which comes after P, so make Q↑.NEXT equal to P↑.NEXT. Finally, we want P↑.NEXT to point to the new element (P↑.NEXT:=Q). If you follow these routines on paper, you'll see how the list will be created in ascending numeric order.

Deleting an Element from a List

Deleting an element also involves a change of pointers, but this procedure is easier than inserting an element because there are no special cases to consider. One deleting routine can be used for all elements. If your list looks like this one:



and you want to delete element C:



you must remove any reference to C from the list. This means making B's link point to the element that C's link was pointing to. In other words, B's link must point to D so that C is no longer referenced by the list. Setting B's pointer to point to D has the effect of deleting C from the list. Since C isn't being used, DISPOSE can be used to free the memory so that it can be reused.

Let's see how the DELETE routine is written, using the structure from the previous section. The number to be deleted is stored in variable DELNUM, and FOUND is a Boolean variable flag which will be TRUE when an element has been deleted and FALSE when it has not.

```
BEGIN
  P:=ENTRY;
  Q:=ENTRY;
  FOUND:=FALSE;
  WHILE (P<>NIL) AND (FOUND=FALSE) DO
  BEGIN
    IF P↑.NUM=DELNUM THEN
    BEGIN
     Q↑.NEXT:=P↑.NEXT;
     DISPOSE(P);
     FOUND:=TRUE
    END
    ELSE
    BEGIN
     Q:=P;
     P:=P↑.NEXT
    END
  END
END
```

The trick to this routine is that Q is always pointing to the element before P (except with the first element). If the program exits the WHILE loop with P=NIL, the element has not been found in the list. If the element is found (P↑.NUM=DELNUM—P↑ is the element we were looking for), then make Q↑ (the element in the list before P↑) point to the element after P↑ (Q↑.NEXT:=P↑.NEXT), and get rid of P↑—DISPOSE(P). While moving along the list, the value of P is always copied into Q before P is advanced to the next element in the list (P:=P↑.NEXT). Thus, Q will always be pointing to the element in the list before P, making this routine possible.

An Example Program

The following program creates a list and then prints it. It uses two procedures to create and print a linked list of characters. The first procedure, ADD, will add characters to the linked list until the EOLN character is found. The second procedure, WRITELIST, will print the list on the screen. The program should print your exact input below your entry.

All the characters are stored in a linked list which uses a record structure called ELEMENT to store the characters. Each entry in the list consists of one character (DATA) and a pointer to the next entry in the list (LINK).

First, we declare the data structures and variables of the main program:

```
PROGRAM COPY(INPUT,OUTPUT);
TYPE
  POINTER=↑ELEMENT;
```

```
  ELEMENT=
    RECORD
     DATA:CHAR;
     LINK:POINTER
    END;
VAR
  ENTRY:POINTER;
```

Now, we declare a procedure which will be used to add each character entered to the linked list structure. This program needs the entry point of the list (ENTRY) as a parameter. ENTRY must also be updated, so a pass-by-reference parameter is used (PTR).

The routine uses two auxiliary pointers to construct the list: P and Q. They are declared as local parameters. The procedure enters a loop which will continue until the last character has been processed (EOLN=TRUE). The routine first checks whether the list is empty. If it is, a new position in memory is created—NEW(P). Then the link pointer is set to NIL, and the data is stored in the new element.

On the other hand, if the list is not empty, pointer P is moved along the list up to the last element, which has P↑.LINK=NIL. When that position is reached, a new position which will follow P is created by using pointer Q, the link of that element pointed to NIL (because it is our new last element) and the data of the element read. Then, P is linked to Q. This adds Q to the end of the list. When execution leaves the WHILE loop, the procedure ends.

```
PROCEDURE ADD(VAR PTR:POINTER);
VAR
  P,Q:POINTER;
BEGIN
  WHILE NOT(EOLN) DO
  BEGIN
    P:=PTR;
    IF P=NIL THEN
    BEGIN
     NEW(P);
     P↑.LINK:=NIL;
     READ(P↑.DATA);
     PTR:=P
    END
    ELSE
    BEGIN
     WHILE P↑.LINK<>NIL DO P:=P↑.LINK;
     NEW(Q);
```

```
      Q↑.LINK:=NIL;
      READ(Q↑.DATA);
      P↑.LINK:=Q
    END
  END
END;
```

The next procedure is used to print the list. Again, it uses PTR as a pass-by-value parameter initially equal to the entry point of the list. This procedure simply moves PTR along the list, printing all the elements (PTR↑.DATA) until PTR=NIL. Then the procedure ends.

```
PROCEDURE WRITELIST(PTR:POINTER);
BEGIN
  WHILE PTR<>NIL DO
  BEGIN
    WRITE(PTR↑.DATA);
    PTR:=PTR↑.LINK
  END
END;
```

The next section is the main program, which simply makes ENTRY:=NIL initially. It then calls the two procedures to build and print the list.

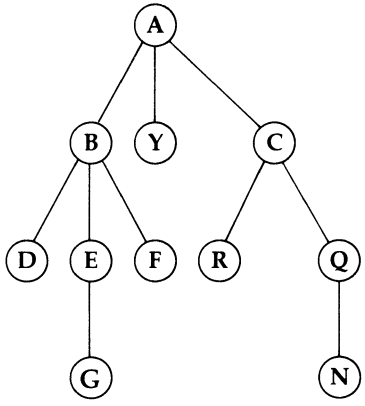
```
BEGIN
  ENTRY:=NIL;
  ADD(ENTRY);
  WRITELIST(ENTRY)
END.
```

Tree Structures

A list is a very simple type of data structure; each element has one pointer to the next element in the list. Several other types of structures exist; for example, there are more complicated lists where an element may point to the element before and the element after itself.

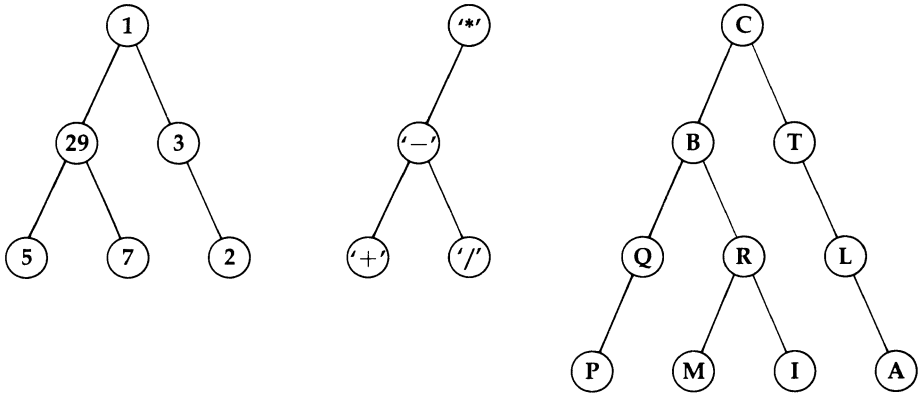
A *tree* structure is one in which each element in the data structure may have pointers to several other data items. These structures are called trees because of the way they are represented graphically. Starting from one element, the graph opens out like a tree, as more and more branches are found. Figure 12-12 is a tree which stores letters.

Figure 12-12. A Tree Structure



We'll discuss just one type of tree structure, the *binary tree*. In a binary tree structure, each element in the tree can have a maximum of two branches (pointers). Figure 12-13 illustrates three possible binary tree structures.

Figure 12-13. Binary Tree Structures



If an element in the binary tree has both branches, the branches are described as the left branch and the right branch.

Binary trees use a record structure as a base type; the fields for the data are specified and two pointer fields are declared for the two possible branches of the tree entry. A typical tree record structure might look like this:

```
TYPE
  POINTER=↑TREE;
  TREE=
```

```
RECORD
 DATA:INTEGER;
 LEFTBRANCH:POINTER;
 RIGHTBRANCH:POINTER
END
```

Recursive routines are ideal to move through all the elements of a binary tree structure. By using recursion, you can write routines that step all the way down one branch of a tree. Then they return to the point where there was a branch-off that was not checked and investigate those branches—thus checking the whole binary tree structure. The following steps would do this:

1. Read the element data
2. Check the left branch
3. Check the right branch

By exchanging the execution order of these three steps, you can execute them in six different ways. It's standard procedure to check the left branch of a tree before the right branch, so a tree can be read in three different ways. The recursive procedure to examine a binary tree completely would be as follows:

```
PROCEDURE MOVEALONG(ARROW:POINTER);
BEGIN
  IF ARROW<>NIL THEN
  BEGIN
    READDATA(ARROW);
    MOVEALONG(ARROW↑.LEFT);
    MOVEALONG(ARROW↑.RIGHT)
  END
```

READDATA is a procedure that will process the current information pointed to by ARROW.

We'll not go into greater detail about trees or other dynamic memory structures in this chapter because we would be escaping from the beginner's scope of this book. Several advanced Pascal publications examine the subject deeper, if it's of interest to you. For most applications, though, linked lists should be sufficient.

CHAPTER 13

The GOTO Statement

CHAPTER 13
The GOTO Statement

Earlier, it was stated that the GOTO instruction exists in Pascal, but that it's seldom used in programs. Given the richness of programming structures offered by Pascal, the GOTO statement should be completely unnecessary (some versions of Pascal do not even recognize it). This chapter will discuss the GOTO statement, since it's available in several Pascal implementations, but its use should be limited.

Labeling Lines

To be able to transfer program control to any line you wish with the GOTO instruction, you must be able to identify that program statement. Since line numbers are not used in Pascal, labels are necessary. Labels are unsigned integer numbers up to 9999 (though some Pascal implementations recognize more digits). To label a certain line, simply write the label, followed by a colon and the instruction:

```
175: WRITELN('HELLO')
```

A label may not be repeated on more than one line, because the computer would not know what line to jump to. Jumping to a nonexistent label causes an execution error.

When you want to transfer control to a labeled line, use GOTO, followed by the label of the line you're jumping to:

```
GOTO 175
```

Declaring GOTO Labels

So that it will be easier to understand program listings which use GOTO jumps, you must declare all labels used in GOTO jumps at the beginning of a Pascal program, before all the other declarative parts (before any TYPE declarations). To declare the labels, use the **LABEL** keyword, followed by the list of

labels used in the program, separated by commas. This example inputs an integer value and, according to the number, executes one of three statements:

```
PROGRAM JUMP(INPUT,OUTPUT);
LABEL 3,28,99;
VAR
  NUM:INTEGER;
BEGIN
  WRITELN('INPUT 1 OR 2  ');
  READLN(NUM);
  IF NUM=1 THEN GOTO 28;
  IF NUM<>2 THEN GOTO 3;
  WRITELN('TWO');
  GOTO 99;
  28:WRITELN('ONE');
  GOTO 99;
  3:WRITELN('WRONG INPUT');
  99:END.
```

In the example, the labels declared are 3, 28, and 99. One variable (NUM) is declared to store integer numbers. Then program execution begins, printing an input prompt. The value is then read in NUM. Next, the comparisons are made. If the value is a 1, execution jumps to the line labeled 28, where the word *ONE* is printed and control is passed to line 99 to end the program. If the value read in NUM is not 1 or 2, the program jumps to the line labeled 3, where an error message is printed and the program ends. If the value is a 2, the word *TWO* is printed, and execution is sent to the end of the program.

A label may also exist without any instruction following it, as in this case:

```
BEGIN
  WRITELN('HELLO');
  GOTO 444;
  WRITELN('THE END');
444:
END
```

In this program segment, control is passed to the line labeled 444, which acts like a blank line and passes control to the END in the next line. The label without any instruction is also considered a Pascal statement, so the statement prior to the labeled line—WRITELN('THE END');—has a semicolon following it as a separator. The label line is not ended with a semicolon because an END follows it.

Incorrect GOTO Jumps

These GOTO jumps are incorrect:

- Jumping to a nonexistent label.
- Jumping into a subprogram or any kind of multiple-statement program loop.

Note that, if you transfer control with a GOTO instruction to a statement belonging to a function or procedure, when the computer finishes executing that routine it will not know where to return since the routine was not actually called by the program. If your program does jump into a function or procedure, the program will execute unpredictably because all subroutine returns will be sequenced incorrectly. The same thing happens when there's a jump to a statement that belongs to a multiple-loop structure (with more than one statement belonging to the loop). The execution of the program from that point on will be unpredictable due to the incorrect values of variables and return jumps.

For example, the following structure is incorrect:

```
FOR X:=1 TO 5 DO
BEGIN
  WRITELN('HELLO');
  9:WRITELN
END;
GOTO 9
```

You may use a GOTO jump to exit a function, procedure, or loop structure—taking care of where control is transferred in the program.

Here are two sections of a Pascal program; both segments produce the same results:

```
IF A=B THEN GOTO 50;
WRITELN('NOT EQUAL');
GOTO 51;
50:WRITELN('EQUAL');
51:...(program continues)
```

```
IF A=B THEN WRITELN('EQUAL')
ELSE WRITELN('NOT EQUAL');
...(program continues)
```

The second segment is far clearer than the first.

When to Use GOTO

The most frequent use of the GOTO statement is to exit a loop. If your program uses a FOR loop, you might want to exit the loop as soon as a certain condition

is met, without having to finish the FOR-DO loop first. You can then use GOTO to exit (in most cases, though, it's easier to use a WHILE or REPEAT structure).

GOTO can also be used to exit a Pascal program whenever an error condition is detected upon execution. When this happens, you can simply jump to the last line of the program.

Suppose you're going to divide the variable A by the variable B. You can first check the value of B to make sure that it's not equal to zero; if it is, transfer control to the end of the program, as in this case:

```
  IF B=0 THEN GOTO 50
  ELSE C:=A/B;
  ...
  (program continues)
  ...
  50:WRITELN('END OF PROGRAM EXECUTION')
END.
```

GOTO should be used when a loop or subprogram will become too complicated because you cannot exit it immediately with a conditional jump. Always try to write your programs without using GOTO. Use GOTO only when a section of your program is very difficult to understand because of a complicated loop or subprogram.

CHAPTER 14

Extensions and Programming Techniques

CHAPTER 14
Extensions and Programming Techniques

Some implementations of Pascal add additional (and/or variations of) instructions to standard Pascal. You should check whether your version of Pascal supports these extensions to standard Pascal.

In this chapter, you'll be introduced to some common extensions to standard Pascal, and we'll look at some of the differences in the various implementations.

OTHERWISE or OTHERS

Several implementations support the **OTHERWISE**, or **OTHERS**, keyword. It's used with a CASE selector when you want control to be transferred to a certain command, or series of commands, when the label in the CASE statement is not found. The idea is simple: Transfer control to a certain label; OTHERWISE (label wasn't found) execute the program statement(s) following OTHERWISE. (OTHERS and OTHERWISE are equivalents.)

Here's an example:

```
WRITELN('INPUT A NUMBER BETWEEN 1 AND 3');
READLN(NUM);
CASE NUM OF
  1:WRITELN('ONE');
  2:WRITELN('TWO');
  3:WRITELN('THREE');
  OTHERWISE:WRITELN('WRONG INPUT')
END
```

System Specifications

The syntax of Pascal programs will vary according to the system you're using. Some systems allow you to put redundant semicolons before certain END keywords; others don't. The programs in this book have been listed using the most standard specifications possible, but if a program you have written presents problems, the cause may be a wrongly placed separator.

Also varying according to the system are the standard function results. Aside from the default field display values for numbers in each implementation, you may find differences in certain routines. Some systems don't allow—or they calculate incorrectly—integer division when they're dealing with negative integers. You may also find that you can use only real parameters (not integer) with mathematical functions that produce real results, like SQRT, SIN, and COS.

Several implementations support alternatives to the symbols prescribed in this book, usually because character sets vary from computer to computer. Although most home computers will work with the symbols described here, check your system's manuals for help if you experience problems.

More on FOR-DO

In Chapter 4, we discussed the use of a FOR loop to vary between INTEGER values. Pascal also allows you to vary the index type of the loop through the values of any scalar type. This means that you might use the following loops:

```
FOR LETTER:='A' TO 'Z' DO...
```

or

```
FOR DAY:=MONDAY TO SUNDAY DO...
```

The second example assumes that a data type with the corresponding list of days has been declared.

FOR loops of any type are useful for cycling through all the possible indexes of an array, which also may have any scalar range as an index.

Extra Functions and Procedures

Several Pascal implementations provide extended subprograms to access files, dynamic memory, and other computer functions. Because file operations are limited to sequential files in standard Pascal, many systems offer subprograms that improve file handling, like the OPEN and SEEK subprograms used to manipulate direct access files. You'll also find subprograms that check whether a file already exists, rename files, and perform other file operations.

Also, according to the implementation used there are variations in the maximum number of files that can be opened simultaneously and the maximum number of characters permitted in a filename.

Some extra (nonstandard) subprograms are offered for dynamic memory manipulation. **FREE** and **MEMAVAIL** are used to check for free memory; **SIZEOF** is used to find the memory used by a certain data type specified as a parameter.

The **DISPOSE** procedure varies from computer to computer. Sometimes it's not even recognized by the interpreter/compiler. Other times it is accepted, but has no actual effect on the amount of memory available. Some systems do not recognize DISPOSE, but let you use the nonstandard procedures **MARK** or **RELEASE** to achieve the same result.

Many Pascal implementations allow the use of the keyword **PACKED** prior to data structures other than ARRAYs—for example, records, files, and sets. The actual effect on memory usage will vary according to the system (PACK and UNPACK are used only with arrays).

If your computer has sound and graphics capabilities, commercial Pascal packages will most likely offer you extended commands to access these features—like high-resolution graphics commands, direct memory manipulation, sound commands, linking to machine language routines, and others.

Writing Pascal Programs

Pascal programs must be written with great care and much thought. Because of the modular design necessary to write good Pascal progams, many BASIC programmers have some difficulty when they first start to program in Pascal.

It's a great help to think of the program as a whole from the beginning and try to design the entire structure of the program before you start writing the code. For example, if the program will have a main menu of choices, remember that most of the program must be included in a large WHILE loop, and that when the END choice is found, execution will leave the main loop and end.

Also try to break up the program into smaller series of routines, which you can then implement as functions and procedures. The subprograms can usually be separated into even smaller program blocks.

Here are some tips that will be useful:

- Choose meaningful identifier names. Remember that you can generally use eight or more character names, and that it's easier to understand identifiers such as RESULT or CHOICE than X or Y.
- Add comments to your programs. Comments will make a program listing easy

to understand if you need to make corrections or modifications. They will be ignored by the compiler when it compiles the program.

- Always declare constants as global parameters. It's always easier to find constant declarations if they're all grouped in the first part of the program than if they're sprinkled throughout all the functions and procedures.
- Type declarations should also be global, as much as possible, except when a variable type is specific to the subprogram only.
- Declare variables according to where they are used. Procedures should operate with their own local parameters; they should avoid modifying global parameters directly if possible. If a procedure must modify a global variable, make sure to mention that fact in your comments. Global parameters should not be modified by functions.
- Try to write your program legibly (indenting loops, subprograms, and so forth) for easier understanding and tracking of program structures.
- Pascal varies from one system to another, so complete compatibility with other packages cannot be insured. It's impossible to determine whether your program will run without any modifications on other systems, but you can write your programs carefully to make any conversions as simple as possible. This means using as few machine-specific routines offered by the compiler as you can. If you do use any of the standard Pascal extensions, document their use carefully in the program listing so that you can easily locate and modify them as necessary.

The Next Step

This book has tried to open the world of Pascal to you, offering a beginner's view of the language. It began with simple programs and comments and led you, step by step, to more advanced topics, covering most of the material necessary to understand and implement any standard Pascal program.

Some topics have not been developed fully, either because the subject was too advanced to be dealt with in this book, or because it was too machine-specific to be generally explained. Such was the case with dynamic data structures and files, which vary greatly according to the system used.

Several publications offer additional information on Pascal. You'll find information that deals with such topics as widely accepted extensions to standard Pascal, different Pascal implementations, and structured programming.

In the appendices is a listing for a Pascal interpreter program. It will allow you to enter most programs in this book and try out your own programs. All commands recognized by the interpreter and specific syntax requirements are listed.

APPENDIX A

Keywords

APPENDIX A
Keywords

Pascal keywords are defined by the language and cannot be changed. They must be written as they appear, without any embedded spaces.

AND	ARRAY	
BEGIN		
CASE	CONST	
DIV	DO	DOWNTO
ELSE	END	
FILE	FOR	FUNCTION
GOTO		
IF	IN	
LABEL		
MOD		
NIL	NOT	
OF	OR	
PACKED	PROCEDURE	PROGRAM
RECORD	REPEAT	
SET		
THEN	TO	TYPE
UNTIL		
VAR		
WHILE	WITH	

Special Symbols

The following is a list of special symbols recognized by Pascal. Special symbols vary according to the computer, so the symbols in this list may not be complete or may vary slightly according to the computer and Pascal implementation being used.

+	−	*	/	↑
<	<>	<=	>=	=
(	)	[	]	:=
(*	*)	;	..	:
;	,	'		

APPENDIX B

Identifiers

APPENDIX B
Identifiers

Identifiers are names chosen by the programmer to identify constants, variables, data types, functions, and procedures. They are formed by a letter followed by a series of letters and digits. Pascal differentiates identifiers by the first eight characters only. The following are the identifiers provided by Pascal (which may be redefined by the programmer):

Standard Functions

ABS	ARCTAN		
CHR	COS		
EOF	EOLN	EXP	
LN			
ODD	ORD		
PRED			
ROUND			
SIN	SQR	SQRT	SUCC
TRUNC			

Standard Procedures

DISPOSE			
GET			
NEW			
PACK	PAGE	PUT	
READ	READLN	RESET	REWRITE
UNPACK			
WRITE	WRITELN		

Standard Constants

FALSE
MAXINT
TRUE

Standard Types

BOOLEAN
CHAR
INTEGER
REAL
TEXT

Standard Files

INPUT
OUTPUT

APPENDIX C

Data Types

APPENDIX C
Data Types

Simple Data Types

REAL
SCALAR
INTEGER
CHAR
BOOLEAN
SUBRANGE
ENUMERATED

Structured

Determined length
Simple base type (arrays, sets, records)
Mixed base type (records, variable records)

Undetermined length
Data files
TEXT files
Dynamic data structures

Pointers

(using memory addresses)

APPENDIX D

Functions and Procedures

APPENDIX D
Functions and Procedures

Standard Functions

SIN(X)
X Integer or real value. The parameter is an angle expressed in radians.
Result Returns the sine of X (a real value).

COS(X)
X Integer or real value. The parameter is an angle expressed in radians.
Result Returns the cosine of X (a real value).

ARCTAN(X)
X Integer or real value
Result The arctangent of X, expressed in radians.

SQRT(X)
X Integer or real
Result The square root of X (a real value).

EXP(X)
X Integer or real
Result *e* to the power of X. The result is a real value.

LN(X)
X Integer or real
Result Returns the logarithm of X in base *e* (a real value).

SQR(X)
X Integer or real
Result The square of X, with the same type as X (integer or real).

ABS(X)
X Integer or real
Result The absolute (positive) value of X, with the same type as X (integer or real).

ODD(X)
X Integer
Result Boolean. TRUE if X is ODD; FALSE if it is even.

TRUNC(X)
X Real
Result Integer. The integer part of X; the decimal part has been truncated.

ROUND(X)
X Real
Result Integer. X rounded to the closest integer number.

PRED(X)
X Any ordinal (all scalars except REAL) type
Result The same ordinal type. Returns the value prior to X, if it exists.

SUCC(X)
X Any ordinal type
Result The same ordinal type. Returns the value following X, if it exists.

ORD(X)
X Any ordinal type
Result Integer. Returns the position of X within its data type.

CHR(X)
X Integer
Result A CHARacter whose ordinal value is X, if it exists.

EOF(X)
X File
Result Boolean. Returns TRUE if the end of the file has been reached and FALSE if it has not.

EOLN(X)
X Text file
Result Boolean. Returns TRUE if the end of the current line has been reached and FALSE if not.

Standard Procedures

READ(X,Y)
X File
Y Parameter list
Result Reads a value from file X and stores it in Y. Several variables may be read in one READ statement by putting several variables in the parameter list. The variable type must be compatible with the data type stored in the file. If X is omitted, READ defaults to the file INPUT.

READLN(X,Y)
X Text file
Y Parameter list
Result Reads a value from text file X and stores it in Y, advancing the file pointer to the start of the next line in the file. Several variables may be read in one READLN statement. If any data remains on the line after a READLN, that data will be skipped. The variable type must be compatible with the data type stored in the file. READLN defaults to the file INPUT.

WRITE(X,Y)
X File
Y Parameter list
Result Writes the value(s) of Y to file X. Several values may be written in one WRITE statement by simply listing the identifiers or constants in the parameter list. If X is omitted, WRITE defaults to the file OUTPUT.

WRITELN(X,Y)
X Text file
Y Parameter list
Result Writes the value(s) of Y in text file X and appends an end-of-line marker to the end of the data being written. Several values may be printed in one WRITELN statement. WRITELN defaults to the file OUTPUT.

RESET(X)
X File
Result Prepares file X for reading, placing the buffer pointer to the first element in the file.

REWRITE(X)
X File
Result Prepares file X for writing, placing the buffer pointer to the first position in the file.

PAGE(X)
X Text file
Result Moves the file pointer to a new "page" of the file. If used without parameters or with the OUTPUT file, clears the screen.

PUT(X)
X File
Result Adds the contents of the file buffer (X↑) to file X and leaves X↑ undefined.

GET(X)
X File—EOF(X) must be FALSE
Result Moves the file pointer to the next element in the file. If it does not exist, EOF(X) is made TRUE. If it does exist, the element is copied in the file buffer (X↑).

NEW(X)
X Pointer
Result Finds a position in memory large enough to hold the data structure that X points to and makes X point to it. An out-of-memory error will occur if there is not enough free memory to make room for the data structure.

DISPOSE(X)
X Pointer
Result Frees the memory used by X↑, leaving pointer X undefined.

PACK(X,Y,Z)
X Array
Y Index type of X
Z Packed array, with same structure type as X
Result Copies X[Y], X[SUCC(Y)], and so forth, to the elements of packed array Z.

UNPACK(X,Y,Z)
X Packed array
Y Array with the same structure type as X
Z Index type of Y
Result Copies all the elements of X to the elements Y[Z], Y[SUCC(Z)], and so forth.

APPENDIX E

Using the Interpreter

APPENDIX E
Using the Interpreter

The Pascal interpreter included here is for the Commodore 64, 64C, and the 128 running in 64 mode. The interpreter is quite long and written entirely in machine language. It is *not* a full-featured interpreter; its purpose is to help a beginner to start programming in Pascal before having to make an investment in a Pascal compiler. Advanced features of Pascal are *not* implemented in the interpreter. Advanced Pascal programmers may want to purchase a full-featured compiler. The interpreter will only save programs to disk (it will not save programs to tape). This appendix explains each of the keywords that the interpreter understands.

The interpreter here will allow you to try many of the examples in the book, especially in the first half, by simply typing in the programs and running them.

The interpreter follows certain syntax rules to operate, and they're explained in the next sections of this appendix, together with a description and explanation of the commands recognized.

Using the Interpreter

An interpreter was chosen over a compiler in order to give you the simplest possible environment while learning Pascal. When operating the interpreter, all you need to do to try a Pascal program is type RUN and press RETURN. Editing, compiling, and linking is not necessary.

Once you have typed in and saved the interpreter using "MLX," you can load the interpreter just as you would any BASIC program:

LOAD ***"filename",8*** <RETURN>

Once the program has loaded, just type **RUN** and press RETURN. The screen will clear and turn black. You are now in Pascal command mode.

If you purchased the interpreter on disk you may want to look at the directory, **LOAD "$",8**, before loading and running the interpreter. The interpreter on the purchased disk has the filename **PASCAL**.

Immediate Commands

The interpreter recognizes the following seven commands: LIST, RUN, SAVE, LOAD, LPRINT, NEW, and X (exit to BASIC). Here's a description of each one:

LIST
Lists a Pascal program currently in memory on the screen. You may slow down the listing with the CONTROL key, and the listing may be interrupted by pressing the RUN/STOP key.

LPRINT
Same as LIST, but lists the program to your printer.

RUN
Executes the Pascal program currently stored in memory.

SAVE
Saves your Pascal program to disk. SAVE must be followed by a space and the filename (do not use quotation marks). The interpreter saves to disk only. You may also follow the filename with a ,8. Here are some examples:

```
SAVE PROGRAM      (saves the program PROGRAM to disk).
SAVE MYPROGRAM,8  (saves the program MYPROGRAM to disk).
```

If an error occurs during the SAVE to disk, the error is printed on the screen. If the SAVE was executed correctly, the appropriate message is displayed.

LOAD
Loads a Pascal program from disk. It is used in the same way as SAVE, following LOAD by the program name (without the quotation marks). You may also follow the program name by a comma and an 8, which means the same:

```
LOAD PROGRAM
LOAD PROGRAM,8
```

After the LOAD, the status condition and any possible errors are reported.

NEW
Resets the interpreter, losing any current Pascal program in memory.

X

Exits to BASIC by warm-starting the computer. You can reenter the interpreter—without losing the Pascal program currently in memory (if you haven't typed any BASIC lines or asked for a directory)—by typing SYS 2080. SYS 2075 will reset the interpreter, NEWing any program in memory.

Any other line entered will be considered a Pascal statement and analyzed as such.

Entering Pascal Programs

A Pascal program line consists of a line number and the Pascal statement. The line number is used to help you enter the program and find errors. Lines which do not start with a numeric digit are rejected immediately. You must separate the line number from the program statement by at least one space. All indentation (spacing between the line number and the Pascal statement) is respected. You may use full-screen editing to enter the Pascal program, just as when entering BASIC programs.

Commands and Structures

The interpreter is not affected by the terminator symbols in the listing. This means that you may or may not use a semicolon to end specific lines. The interpreter will accept lines with or without the semicolon. If you have not balanced the BEGINs and ENDs correctly, the interpreter will report an error when it comes across the last statement of the program. This error also occurs with the incorrect use of certain structures.

Keywords Recognized by the Interpreter

PROGRAM

The PROGRAM keyword is used to mark the beginning of your program. The interpreter does not require you to specify what files you will be using in the program header, though you may do so. All programs default to the screen and the keyboard as devices. Examples:

```
PROGRAM HELLO; {no files}
PROGRAM TEST(INPUT,OUTPUT);
```

CONST

Following the program header, constants may be declared using the CONST keyword to start the section. The first constant declaration may follow the keyword CONST on the same line. Thus, the following declarations are the same:

```
CONST A=12;
  TXT='HELLO'
```

or,

```
CONST
  A=12;
  TXT='HELLO'
```

Constants are stored in a table in memory, with the name, value, and type specified.

Constants may only be of INTEGER, REAL, BOOLEAN, or CHAR types. An operation may follow the constant assignment, using constants already defined as well as standard functions:

```
A=12
B=5*2.3-A
C=SQR(A)-LN(B)
```

MAXINT is a predefined integer constant with the value of 32767.

TYPE

You may declare data types using TYPE to start the section. Again, the first definition may follow the keyword TYPE, as with constants. Data types include subranges (integer or character) and references to other types.

Subranges separate the two ranges by two consecutive dots. Subranges may be of integers or characters:

```
INTRANGE=3..5;
BIGRANGE=1000..MAXINT;
CHARACTERS='C'..'F'
```

You can refer to another type in the assignment:

```
OURINT=INTEGER;
YOURINT=OURINT
```

VAR

The variable declaration section comes after the TYPE declarations and starts with the VAR keyword. The first variable declaration may follow the VAR keyword. Following the variable identifier, you may assign any basic data type, defined type, subrange, and so on, just as with type. Remember that a colon is used to separate variable declarations. If several variable names share one type, separate the variables by commas:

```
VAR A:INTEGER;
  B,C,D:REAL;
  NUMBERS: TYPEDEF;   (previously defined in a TYPE statement)
  DAYS: 3..7;
  ONEMORE:'N'..'V';
```

In other words, you can leave out TYPE declarations completely, making all the declarations directly in the VARiable section. That's up to you.

VAR has the effect of creating the variables you declare for later use in the program. The variable name and type are stored, and the value is left undefined. The memory is only set aside, not initialized (in most cases), so don't assume numeric variables will be zero, as in BASIC, and so forth. You should always reset the values of the variables used yourself.

BEGIN

This keyword marks the beginning of any specific structure, and in particular, the beginning of the Pascal program. BEGIN may be followed by an instruction on the same line:

```
BEGIN
  A:=12
```

is the same as

```
BEGIN A:=12
```

Avoid following BEGIN by a loop structure such as WHILE, REPEAT, or FOR.

END

END marks the end of a given structure. The interpreter either uses the jump stack to return to another point in the program or continues execution, according to the structure in which END is used. If the END keyword is followed by a full stop, program execution is terminated. If END is marking the end of a procedure, control returns to the instruction following the subprogram call.

Always use END on a separate program line.

REPEAT

This keyword is used to build a loop structure. When it is found, the position of the statement is stored in the return stack of the program, for later return according to the UNTIL condition. A statement may follow REPEAT, as with BEGIN.

UNTIL

The expression following the keyword UNTIL is evaluated. If it is FALSE, control returns to the corresponding REPEAT address on the stack. If it is TRUE, the return address is removed from the stack and execution continues with the next statement.

WHILE..DO

When the interpreter finds a WHILE keyword, it evaluates the expression following. If the result is TRUE, the following statement (or group of statements

enclosed by BEGIN and END) is executed, and control is transferred back to the WHILE statement for a new condition evaluation. If the result is FALSE, execution skips the loop instructions and continues.

IF..THEN..ELSE
The interpreter evaluates the condition following the IF statement. If it is TRUE, it executes the statement (or group of statements enclosed by BEGIN and END) following THEN. If it is FALSE, the interpreter checks for the existence of an ELSE condition. If it exists, it executes the ELSE statement(s).

FOR..TO/DOWNTO..DO
The FOR loop is executed in the same way as the FOR/NEXT in BASIC, but you're not restricted to integer variables as control variables. The control variable may be any structured data type such as characters, enumerated types, and so on. If TO is used, the interpreter finds the following value of the control variable until it matches the second parameter using the SUCC function. If it is DOWNTO, it finds the previous value until it matches the second condition using the PRED function. You may enclose one statement in the FOR loop, or a group of statements enclosed by BEGIN and END. Here are some examples:

```
FOR A:='A' TO 'Z' DO...
FOR I:=5 DOWNTO 1 DO...
```

AND, OR, and **NOT**
These three keywords are used with logical evaluations, discussed below.

Standard Procedures

The following procedures are recognized by the interpreter, but may be redefined by the programmer.

PAGE
PAGE will clear the screen if used without parameter or with the OUTPUT file as parameter.

WRITE
This procedure is used to print information on the screen. WRITE prints the corresponding value(s) and leaves the cursor to the right of whatever was just printed. WRITE may be used to print string constants, INTEGER, REAL, BOOLEAN, and CHAR, whether expressions, constants, or variables. If more than one element is to be printed in one WRITE, separate the elements by commas.

With all data types, you may specify the field size in which the value should be printed. This may be any expression which, when evaluated, returns

an INTEGER result. When printing REAL numbers, you may also include a second parameter which determines the number of digits to be printed to the right of the decimal point, either truncating the number of digits or completing them with zeros.

All values are printed following each other with no default field-size, and REAL numbers default to scientific notation (with eight decimal digits) if printed without two field specifications.

WRITELN
This command works in the same way as WRITE, but follows the output with a line feed.

READ
READ is used to read values from the keyboard. The variables read must be INTEGER, REAL, or CHAR. When a value or expression has been read from the keyboard, it is evaluated and assigned to the corresponding variable. Input for several numerical variables (real or integer) may be written on the same line, separating each value by at least one space. When you are entering values to be assigned to CHARacter variables, leading spaces are not ignored (they are with numeric data). Spaces at the end of an input line are ignored, as in BASIC.

READLN
Essentially the same as READ, but a line feed follows the READ and the "read-buffer" is cleared. (Due to the routine used to enter data in READ, the cursor always goes down one line at a READ or READLN. The difference is that with READ, the "read-buffer" is not cleared).

Standard Functions

These functions are already defined by the interpreter, but may be redefined by the Pascal programmer.

SIN, COS, ARCTAN
These functions operate with angles expressed in radians. Their parameters may either be INTEGER or REAL. The result is always REAL.

SQRT, LN, EXP
Parameters for these functions may either be INTEGER or REAL. The result of the operation is always a REAL value.

ABS, TRUNC, SQR, ROUND
These functions operate with INTEGER or REAL values. ABS and SQR return either INTEGER or REAL values, according to the parameter type, and TRUNC and ROUND return INTEGER values.

ORD, SUCC, PRED

Used with any scalar type parameter (INTEGER, BOOLEAN, CHAR or a defined TYPE), these functions return the previous element in the list (PRED), and the following element in the list (SUCC). ORD returns an ASCII value if used with characters, the same integer number if used with INTEGER, and the position in the data list if used with a defined type.

CHR

Returns the character corresponding to the ASCII code passed as the parameter (as CHR$ in BASIC).

ODD

This function returns a Boolean value of TRUE or FALSE. It uses any expression which, when evaluated, returns an integer value.

Mathematical Evaluations

Mathematical evaluations support all Pascal operators and parentheses to change evaluation order.

Expressions in parentheses are evaluated first, with the + and − symbols as main operators. Within these "evaluation blocks," the *, /, MOD, and DIV operators all have the same precedence and are evaluated on a left-to-right basis.

If you are not sure of what operations will be evaluated first in an expression, use parentheses.

Numbers in scientific notation may be used, as well as constants and variables. Negative values may be used directly without parentheses. That is, you may write

```
2*-3
```

which is the same as 2*(−3). To negate a variable directly, enclose it in parentheses. For example, A:=−MAXINT should be written as A:=−(MAXINT).

Logical Evaluations

All logical operators have the same precedence (AND, OR, <, >, <>, =, <=, >=). Use parentheses to change evaluation order. NOT is recognized by the interpreter, but it is implemented as a function, so the expression which you want to negate must follow the word *NOT* in parentheses. Characters may be compared with the relational operators (but not with AND, OR).

Interpreter Errors

If an error occurs when you are entering an immediate command, the interpreter reports a syntax error. If the error occurs during the execution of a Pascal program, an error message will be printed, together with the approximate (and rather accurate) line number where the error was first found. The errors are:

DATA TYPE ERROR
Incompatible types are being used.

DIVISION BY ZERO ERROR

IDENTIFIER ERROR
Incorrectly spelled keyword, standard function or procedure, wrong user-identifier, and so on.

ILLEGAL QUANTITY ERROR
An illegal value resulted in an operation.

INTEGER ERROR
Caused by a value used outside an integer context.

OVERFLOW ERROR
A number too large resulted in an operation.

STACK ERROR
Unbalanced structures in the program.

SUBRANGE ERROR
Invalid range has been declared, or a value is not in a specified range.

SYNTAX ERROR
Caused by an incorrect Pascal statement. Usually this means an uneven number of parentheses, misspelled words, and such.

Other errors are just marked with an ERROR IN message and the line number. These are errors undetermined by the computer.

APPENDIX F

Typing In the Interpreter

APPENDIX F
Typing In the Interpreter

In order to make the typing of the Pascal Interpreter as easy as possible on your Commodore 64 and 128, we've included two program entry aids written in BASIC: "The Automatic Proofreader" and "MLX." To assist you in understanding how to enter these programs, COMPUTE! has established the following listing conventions.

Note: The Pascal interpreter included here is for the Commodore 64 and 128 running in 64 mode. It is *not* a full-featured Pascal interpreter—rather it's intended for those Commodore owners who do not own a compiler or interpreter, and who wish to learn Pascal before investing in a full-featured Pascal. (The interpreter included here will only save Pascal programs to disk; it will *not* save programs to tape.) Read Appendix E before typing in the interpreter to see if it meets your needs.

Since all the programs in the appendix are for the Commodore 64 and the Commodore 128 running in 64 mode, 128 owners should type **GO 64** and press RETURN before beginning to enter or load any programs from this book.

Generally, BASIC program listings like the one for MLX will contain words within braces which spell out any special characters: {DOWN} means to press the cursor-down key; {5 SPACES} means to press the space bar five times.

To indicate that a key should be *shifted* (pressing the key while holding down the SHIFT key), the key will be underlined in our listings. For example, S means to type the S key while holding the SHIFT key. This would appear on your screen as a heart symbol. If you find an underlined key enclosed in braces (for example, {10 N}), you should type the key as many times as indicated. In that case, you would enter ten shifted N's.

If a key is enclosed in special brackets, [< >], you should hold down the *Commodore key* while pressing the key inside the special brackets. (The Commodore key is the key in the lower left corner of the keyboard.) Again, if the key is preceded by a number, you should press the key as many times as indicated; [<9@>] means type Commodore-@ nine times.

Refer to the following table when entering cursor and color control keys:

Keyboard Conventions

When You Read:	Press:	See:
{CLR}	SHIFT + CLR/HOME	
{HOME}	CLR/HOME	
{UP}	SHIFT + ↑ CRSR ↓	
{DOWN}	↑ CRSR ↓	
{LEFT}	SHIFT + ← CRSR →	
{RIGHT}	← CRSR →	
{RVS}	CTRL + 9	
{OFF}	CTRL + 0	
{BLK}	CTRL + 1	
{WHT}	CTRL + 2	
{RED}	CTRL + 3	
{CYN}	CTRL + 4	
{PUR}	CTRL + 5	
{GRN}	CTRL + 6	
{BLU}	CTRL + 7	
{YEL}	CTRL + 8	

When You Read:	Press:	See:
{1}	COMMODORE + 1	
{2}	COMMODORE + 2	
{3}	COMMODORE + 3	
{4}	COMMODORE + 4	
{5}	COMMODORE + 5	
{6}	COMMODORE + 6	
{7}	COMMODORE + 7	
{8}	COMMODORE + 8	
{F1}	f1	
{F2}	SHIFT + f1	
{F3}	f3	
{F4}	SHIFT + f3	
{F5}	f5	
{F6}	SHIFT + f5	
{F7}	f7	
{F8}	SHIFT + f7	
←	←	
↑	SHIFT + ↑	

The Automatic Proofreader

Philip I. Nelson

"The Automatic Proofreader" helps you type in BASIC programs without typing mistakes. It's a short error-checking program that conceals itself in memory and adheres to your Commodore's operating system. Each time you press RETURN to enter a program line, the Proofreader displays a two-letter checksum in reverse video at the top of your screen. If the checksum on your screen doesn't match the one in the printed listing, you've typed the line incorrectly—it's that simple. You don't have to use the Proofreader to enter printed listings, but doing so greatly reduces your chances of making a typographical error. We suggest you use the Proofreader to enter the "MLX" program.

Getting Started

First, type in the Automatic Proofreader (Program F-1) *exactly* as it appears in the listing. Since the Proofreader can't check itself, type carefully to avoid mistakes. Don't omit any lines, even if they contain unfamiliar commands or you think they don't apply to your computer. As soon as you've finished typing the Proofreader, save at least two copies on disk before running it the first time. This is very important because the Proofreader erases the BASIC portion of itself when you run it, leaving only the machine language portion in memory.

When that's done, type RUN and press RETURN. After announcing which computer it's running on, the Proofreader installs the ML routine in memory, displays the message PROOFREADER ACTIVE, erases the BASIC portion of itself, and ends. If you type LIST and press RETURN, you'll see that no BASIC program remains in memory. The computer is ready for you to type in MLX (Program F-2).

Entering MLX

Once the Proofreader is active, you can begin typing in MLX (Program F-2). Every time you finish typing a line and press RETURN, the Proofreader displays a two-letter checksum (reverse-video letters) in the upper left corner of the screen. Compare this checksum with the two-letter checksum printed to the left of the corresponding line in the program listing. If the letters match, it's almost certain the line was typed correctly. If the letters don't match, check for your mistake and correct the line.

The Proofreader ignores spaces that aren't enclosed in quotation marks, so you can omit spaces (or add extra ones) between keywords and still see a matching checksum. For example, these two lines generate the same checksum:

```
10 PRINT"THIS IS BASIC"
10 PRINT          "THIS IS BASIC"
```

However, since spaces inside quotation marks are almost always significant, the Proofreader pays attention to them. For instance, these two lines generate different checksums:

```
10 PRINT"THIS IS BASIC"
10 PRINT"THIS ISBA     SIC"
```

A common typing mistake is transposition—typing two successive characters in the wrong order, as in *PIRNT* instead of *PRINT*, or *64378* instead of *64738*. A checksum program that adds up the values of all the characters in a line can't possibly detect transposition errors (it can only tell whether the right characters are present, regardless of what order they're in). Because the Proofreader computes the checksum with a more sophisticated formula, it is also sensitive to the *position* of each character within the line and thus catches transposition errors.

The Proofreader does *not* accept keyword abbreviations (for example, ? instead of PRINT). If you prefer to use abbreviations, you can still check the line with the Proofreader: Simply LIST the line after typing it, move the cursor back onto the line, and press RETURN. LISTing the line substitutes the full keyword for the abbreviation and allows the Proofreader to work properly. The same technique works for rechecking a program you've already typed in: Reload the program, LIST several lines on the screen, and press RETURN over them.

Though the Proofreader doesn't interfere with other BASIC operations, it's always a good idea to disable it before running any other program. Some programs may need the space occupied by the Proofreader's ML routine or may create other memory conflicts. However, the Proofreader is purposely made difficult to dislodge: It's not affected by disk operations, or by pressing RUN/STOP–RESTORE. The simplest way to disable it is to turn the computer off, then on again.

Program F-1. The Automatic Proofreader

```
10 VEC=PEEK(772)+256*PEEK(773):LO=43:HI=44
20 PRINT "AUTOMATIC PROOFREADER FOR ";:IF VEC=4236
   4 THEN PRINT "C-64"
30 IF VEC=50556 THEN PRINT "VIC-20"
40 IF VEC=35158 THEN GRAPHIC CLR:PRINT "PLUS/4 & 1
   6"
50 IF VEC=17165 THEN LO=45:HI=46:GRAPHIC CLR:PRINT
   "128"
60 SA=(PEEK(LO)+256*PEEK(HI))+6:ADR=SA
70 FOR J=0 TO 166:READ BYT:POKE ADR,BYT:ADR=ADR+1:
   CHK=CHK+BYT:NEXT
80 IF CHK<>20570 THEN PRINT "*ERROR* CHECK TYPING
   {SPACE}IN DATA STATEMENTS":END
90 FOR J=1 TO 5:READ RF,LF,HF:RS=SA+RF:HB=INT(RS/2
   56):LB=RS-(256*HB)
100 CHK=CHK+RF+LF+HF:POKE SA+LF,LB:POKE SA+HF,HB:N
   EXT
110 IF CHK<>22054 THEN PRINT "*ERROR* RELOAD PROGR
   AM AND CHECK FINAL LINE":END
120 POKE SA+149,PEEK(772):POKE SA+150,PEEK(773)
130 IF VEC=17165 THEN POKE SA+14,22:POKE SA+18,23:
   POKESA+29,224:POKESA+139,224
140 PRINT CHR$(147);CHR$(17);"PROOFREADER ACTIVE":
   SYS SA
150 POKE HI,PEEK(HI)+1:POKE (PEEK(LO)+256*PEEK(HI)
   )-1,0:NEW
160 DATA 120,169,73,141,4,3,169,3,141,5,3
170 DATA 88,96,165,20,133,167,165,21,133,168,169
180 DATA 0,141,0,255,162,31,181,199,157,227,3
190 DATA 202,16,248,169,19,32,210,255,169,18,32
200 DATA 210,255,160,0,132,180,132,176,136,230,180
210 DATA 200,185,0,2,240,46,201,34,208,8,72
220 DATA 165,176,73,255,133,176,104,72,201,32,208
230 DATA 7,165,176,208,3,104,208,226,104,166,180
240 DATA 24,165,167,121,0,2,133,167,165,168,105
250 DATA 0,133,168,202,208,239,240,202,165,167,69
260 DATA 168,72,41,15,168,185,211,3,32,210,255
270 DATA 104,74,74,74,74,168,185,211,3,32,210
280 DATA 255,162,31,189,227,3,149,199,202,16,248
290 DATA 169,146,32,210,255,76,86,137,65,66,67
300 DATA 68,69,70,71,72,74,75,77,80,81,82,83,88
310 DATA 13,2,7,167,31,32,151,116,117,151,128,129,
   167,136,137
```

MLX
Machine Language Entry Program

Ottis R. Cowper

Type in and save at least two copies of "MLX" (you'll want to use it to enter future Commodore 64 machine language programs from *COMPUTE!* magazine, *COMPUTE!'s Gazette*, and COMPUTE! books). If you're using a Commodore 128, you can still use MLX—you must enter it and any ML programs in 64 mode, however.

The Pascal interpreter included here is for the Commodore 64 and 128 running in 64 mode. It is *not* a full-featured Pascal interpreter—rather it's intended for those Commodore owners who do not own a compiler or interpreter, and who wish to learn Pascal before investing in a full-featured Pascal. Read Appendix E before typing in the interpreter to see if it meets your needs.

When you're ready to enter the Pascal interpreter (Program F-3 below) for the Commodore 64 and 128, load and run MLX (Commodore 128 owners must be in 64 mode). It asks you for a starting address and an ending address. These addresses are:

Starting address: $0801
Ending address: $3C70

If you're unfamiliar with machine language, the addresses (and all other values you enter in MLX) may appear strange. Instead of the usual decimal numbers you're accustomed to, these numbers are in *hexadecimal*—a base-16 numbering system commonly used by ML programmers. Hexadecimal—*hex* for short—includes the numerals 0–9 and the letters A–F. But don't worry—even if you know nothing about ML or hex, you should have no trouble using MLX.

After you have entered the starting and ending addresses, you'll be offered the option of clearing the workspace. Choose this option if you're starting to enter a new listing. If you're continuing a listing that's partially typed from a previous session, don't choose this option.

A functions menu appears. The first option in the menu is ENTER DATA. If you're just starting to type in a program, pick this. Press the E key, and type the first number in the first line of the program listing. If you've already typed in part of a program, type the line number where you left off typing at the end of the previous session (be sure to load the partially completed program before you resume entry). In any case, make sure the address you enter corresponds to the address of a line in the listing you are entering. Other-

wise, you'll be unable to enter the data correctly. If you press E by mistake, you can return to the command menu by pressing RETURN alone when asked for the address. (You can get back to the menu from most options by pressing RETURN with no other input.)

Entering a Listing

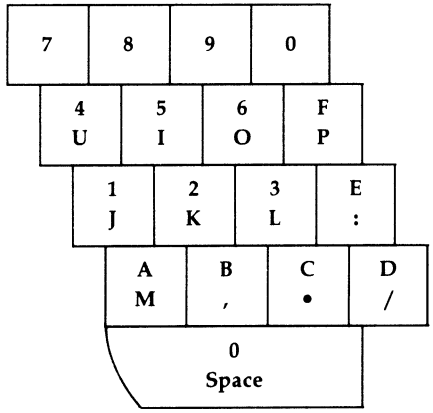
Once you're in Enter mode, MLX prints the address for each program line for you. You then type in all nine numbers on that line, beginning with the first two-digit number after the colon (:). Each line represents eight data bytes and a checksum. Although an MLX-format listing appears similar to the "hex dump" listings from a machine language monitor program, the extra checksum number on the end allows MLX to check your typing.

When you enter a line, MLX recalculates the checksum from the eight bytes and the address, and compares this value with the number from the ninth column. If the values match, you'll hear a bell tone, the data will be added to the workspace area, and the prompt for the next line of data will appear. But if MLX detects a typing error, you'll hear a low buzz and see an error message. The line will then be redisplayed for editing.

Invalid Characters Banned

Only a few keys are active while you're entering data, so you may have to unlearn some habits. You *do not* type spaces between the columns; MLX automatically inserts these for you. You *do not* press RETURN after having typed the last number in a line; MLX automatically enters and checks the line after you type the last digit.

64 MLX Keypad



Only the numerals 0–9 and the letters A–F can be typed in. If you press any other key (with some exceptions noted below), you'll hear a warning buzz. To simplify typing, a keypad is available. The keypad—part of the keyboard—is active only while you're entering data. Addresses must be entered with the normal letter and number keys. The figure shows the keypad configuration.

MLX checks for transposed characters. If you're supposed to type in A0 and enter 0A instead, MLX will catch your mistake. There is one error that can slip past MLX: Because of the checksum formula used, MLX won't notice if you accidentally type FF in place of 00, and vice versa. And there's a very slim chance that you could garble a line and still end up with a combination of characters that adds up to the proper checksum. However, these mistakes should not occur if you take reasonable care while entering data.

Editing Features

To correct typing mistakes before finishing a line, use the INST/DEL key to delete the character to the left of the cursor. (The cursor-left key also deletes.) If you really mess up a line, press CLR/HOME to restart the line. The RETURN key is also active, but only before any data is typed on a line. Pressing RETURN at this point returns you to the command menu. After you've typed a character of data, MLX disables RETURN until the cursor returns to the start of a line. Remember, you can press CLR/HOME to get to a line number prompt quickly.

More editing features are available when you're correcting lines in which MLX has detected an error. To make corrections in a line that MLX has redisplayed for editing, compare the line on the screen with the one printed in the listing; then move the cursor to the mistake and type the correct key. The cursor-left and -right keys provide the normal cursor controls. (The INST/DEL key now works as an alternative cursor-left key.) You cannot move left beyond the first character in the line. If you try to move beyond the rightmost character, you'll reenter the line. During editing, RETURN is active; pressing it tells MLX to recheck the line. You can press the CLR/HOME key to clear the entire line if you want to start from scratch or if you want to get to a line number prompt to use RETURN to get back to the menu.

Display Data

The second menu choice, DISPLAY DATA, examines memory and shows the contents in the same format as the program listing (including the checksum). When you press D, MLX asks you for a starting address. Be sure that the starting address you give corresponds to a line number in the listing. Otherwise, the checksum display will be meaningless. MLX displays program lines until it

reaches the end of the program, at which point the menu is redisplayed. You can pause the display by pressing the space bar. (MLX finishes printing the current line before halting.) Press the space bar again to restart the display. To break out of the display and get back to the menu before the ending address is reached, press RETURN.

Other Menu Options

Two more menu selections let you save programs and load them back into the computer. These are SAVE FILE and LOAD FILE; their operation is quite straightforward. When you press S or L, MLX asks you for the filename. You'll then be asked to press either D or T to select disk or tape.

You'll notice the disk drive starting and stopping several times during a load or save. Don't panic; this is normal behavior. MLX opens and reads from or writes to the file instead of using the usual LOAD and SAVE commands. Disk users should also note that the drive prefix 0: is automatically added to the filename (line 750), so it should *not* be included when you enter the name. This also precludes the use of @ for Save-with-Replace, so remember to give each version you save a different name.

Remember that MLX saves the entire workspace area from the starting address to the ending address, so the save or load may take longer than you might expect if you've entered only a small amount of data from a long listing. When saving a partially completed listing, make sure to note the address where you stopped typing so that you'll know where to resume entry when you reload.

MLX reports the standard disk error messages if any problems are detected during the save or load. MLX also has three special load error messages: INCORRECT STARTING ADDRESS, which means the file you're trying to load does not have the starting address you specified when you ran MLX; LOAD ENDED AT *address*, which means the file you're trying to load ends before the ending address you specified when you started MLX; and TRUNCATED AT ENDING ADDRESS, which means the file you're trying to load extends beyond the ending address you specified when you started MLX. If you see one of these messages and feel certain that you've loaded the right file, exit and rerun MLX, being careful to enter the correct starting and ending addresses.

The QUIT menu option has the obvious effect—it stops MLX and enters BASIC. The RUN/STOP key is disabled, so the Q option lets you exit the program without turning off the computer. (Of course, RUN/STOP–RESTORE also gets you out.) You'll be asked for verification; press Y to exit to BASIC or any other key to return to the menu. After quitting, you can type RUN again and reenter MLX without losing your data, as long as you don't use the clear workspace option.

The Finished Product

When you've finished typing all the data for the interpreter and saved your work, you're ready to see the results. To load and run the interpreter, simply load and run it as you would a BASIC program:

LOAD *"filename",8* <RETURN>

and type **RUN**; then press RETURN. For complete interpreter instructions, read Appendix E.

An Ounce of Prevention

By the time you finish typing in the data for a long ML program, you may have several hours invested in the project. Don't take chances—read "How to Type In Programs" (above), use "The Automatic Proofreader" to type in MLX, and then test your copy *thoroughly* before first using it to enter any significant amount of data. Make sure all the menu options work as they should. Enter fragments of the program starting at several different addresses; then use the Display option to verify that the data has been entered correctly. And be sure to test the Save and Load options several times to insure that you can recall your work from disk. Don't let a simple typing error in the new MLX cost you several nights of hard work.

Program F-2. MLX

For error-free entry, use "The Automatic Proofreader" (Program F-1) to type in this program.

```
EK 100 POKE 56,50:CLR:DIM IN$,I,J,A,B,A$,B$,A(7),N
       $
DM 110 C4=48:C6=16:C7=7:Z2=2:Z4=254:Z5=255:Z6=256:
       Z7=127
CJ 120 FA=PEEK(45)+Z6*PEEK(46):BS=PEEK(55)+Z6*PEEK
       (56):H$="0123456789ABCDEF"
SB 130 R$=CHR$(13):L$="{LEFT}":S$=" ":D$=CHR$(20):
       Z$=CHR$(0):T$="{13 RIGHT}"
CQ 140 SD=54272:FOR I=SD TO SD+23:POKE I,0:NEXT:PO
       KE SD+24,15:POKE 788,52
FC 150 PRINT"{CLR}"CHR$(142)CHR$(8):POKE 53280,15:
       POKE 53281,15
EJ 160 PRINT T$" {RED}{RVS}{2 SPACES}[8 @]
       {2 SPACES}"SPC(28)"{2 SPACES}{OFF}{BLU} MLX
        II {RED}{RVS}{2 SPACES}"SPC(28)"
       {12 SPACES}{BLU}"
FR 170 PRINT"{3 DOWN}{3 SPACES}COMPUTE!'S MACHINE
       {SPACE}LANGUAGE EDITOR{3 DOWN}"
```

```
JB 180 PRINT"{BLK}STARTING ADDRESS[43";:GOSUB300:S
       A=AD:GOSUB1040:IF F THEN180
GF 190 PRINT"{BLK}{2 SPACES}ENDING ADDRESS[43";:GO
       SUB300:EA=AD:GOSUB1030:IF F THEN190
KR 200 INPUT"{3 DOWN}{BLK}CLEAR WORKSPACE [Y/N][43
       ";A$:IF LEFT$(A$,1)<>"Y"THEN220
PG 210 PRINT"{2 DOWN}{BLU}WORKING...";:FORI=BS TO
       {SPACE}BS+EA-SA+7:POKE I,0:NEXT:PRINT"DONE"
DR 220 PRINTTAB(10)"{2 DOWN}{BLK}{RVS} MLX COMMAND
        MENU {DOWN}[43":PRINT T$"{RVS}E{OFF}NTER D
       ATA"
ED 230 PRINT T$"{RVS}D{OFF}ISPLAY DATA":PRINT T$"
       {RVS}L{OFF}OAD DATA"
JS 240 PRINT T$"{RVS}S{OFF}AVE FILE":PRINT T$"
       {RVS}Q{OFF}UIT{2 DOWN}{BLK}"
JH 250 GET A$:IF A$=N$ THEN250
HK 260 A=0:FOR I=1 TO 5:IF A$=MID$("EDLSQ",I,1)THE
       N A=I:I=5
FD 270 NEXT:ON A GOTO420,610,690,700,280:GOSUB1060
       :GOTO250
EJ 280 PRINT"{RVS} QUIT ":INPUT"{DOWN}[43ARE YOU S
       URE [Y/N]";A$:IF LEFT$(A$,1)<>"Y"THEN220
EM 290 POKE SD+24,0:END
JX 300 IN$=N$:AD=0:INPUTIN$:IFLEN(IN$)<>4THENRETUR
       N
KF 310 B$=IN$:GOSUB320:AD=A:B$=MID$(IN$,3):GOSUB32
       0:AD=AD*256+A:RETURN
PP 320 A=0:FOR J=1 TO 2:A$=MID$(B$,J,1):B=ASC(A$)-
       C4+(A$>"@")*C7:A=A*C6+B
JA 330 IF B<0 OR B>15 THEN AD=0:A=-1:J=2
GX 340 NEXT:RETURN
CH 350 B=INT(A/C6):PRINT MID$(H$,B+1,1);:B=A-B*C6:
       PRINT MID$(H$,B+1,1);:RETURN
RR 360 A=INT(AD/Z6):GOSUB350:A=AD-A*Z6:GOSUB350:PR
       INT":";
BE 370 CK=INT(AD/Z6):CK=AD-Z4*CK+Z5*(CK>Z7):GOTO39
       0
PX 380 CK=CK*Z2+Z5*(CK>Z7)+A
JC 390 CK=CK+Z5*(CK>Z5):RETURN
QS 400 PRINT"{DOWN}STARTING AT[43";:GOSUB300:IF IN
       $<>N$ THEN GOSUB1030:IF F THEN400
EX 410 RETURN
HD 420 PRINT"{RVS} ENTER DATA ":GOSUB400:IF IN$=N$
        THEN220
```

```
JK 430 OPEN3,3:PRINT
SK 440 POKE198,0:GOSUB360:IF F THEN PRINT IN$:PRIN
       T"{UP}{5 RIGHT}";
GC 450 FOR I=0 TO 24 STEP 3:B$=S$:FOR J=1 TO 2:IF
       {SPACE}F THEN B$=MID$(IN$,I+J,1)
HA 460 PRINT"{RVS}"B$L$;:IF I<24THEN PRINT"{OFF}";
HD 470 GET A$:IF A$=N$ THEN470
FK 480 IF(A$>"/"ANDA$<":")OR(A$>"@"ANDA$<"G")THEN5
       40
MP 490 IF A$=R$ AND((I=0)AND(J=1)OR F)THEN PRINT B
       $;:J=2:NEXT:I=24:GOTO550
KC 500 IF A$="{HOME}" THEN PRINT B$:J=2:NEXT:I=24:
       NEXT:F=0:GOTO440
MX 510 IF(A$="{RIGHT}")ANDF THENPRINT B$L$;:GOTO54
       0
GK 520 IF A$<>L$ AND A$<>D$ OR((I=0)AND(J=1))THEN
       {SPACE}GOSUB1060:GOTO470
HG 530 A$=L$+S$+L$:PRINT B$L$;:J=2-J:IF J THEN PRI
       NT L$;:I=I-3
QS 540 PRINT A$;:NEXT J:PRINT S$;
PM 550 NEXT I:PRINT:PRINT"{UP}{5 RIGHT}";:INPUT#3,
       IN$:IF IN$=N$ THEN CLOSE3:GOTO220
QC 560 FOR I=1 TO 25 STEP3:B$=MID$(IN$,I):GOSUB320
       :IF I<25 THEN GOSUB380:A(I/3)=A
PK 570 NEXT:IF A<>CK THEN GOSUB1060:PRINT"{BLK}
       {RVS} ERROR: REENTER LINE [43]":F=1:GOTO440
HJ 580 GOSUB1080:B=BS+AD-SA:FOR I=0 TO 7:POKE B+I,
       A(I):NEXT
QQ 590 AD=AD+8:IF AD>EA THEN CLOSE3:PRINT"{DOWN}
       {BLU}** END OF ENTRY **{BLK}{2 DOWN}":GOTO7
       00
GQ 600 F=0:GOTO440
QA 610 PRINT"{CLR}{DOWN}{RVS} DISPLAY DATA ":GOSUB
       400:IF IN$=N$ THEN220
RJ 620 PRINT"{DOWN}{BLU}PRESS: {RVS}SPACE{OFF} TO
       {SPACE}PAUSE, {RVS}RETURN{OFF} TO BREAK[43]
       {DOWN}"
KS 630 GOSUB360:B=BS+AD-SA:FORI=BTO B+7:A=PEEK(I):
       GOSUB350:GOSUB380:PRINT S$;
CC 640 NEXT:PRINT"{RVS}";:A=CK:GOSUB350:PRINT
KH 650 F=1:AD=AD+8:IF AD>EA THENPRINT"{DOWN}{BLU}*
       * END OF DATA **":GOTO220
KC 660 GET A$:IF A$=R$ THEN GOSUB1080:GOTO220
```

```
EQ 67Ø IF A$=S$ THEN F=F+1:GOSUB1Ø8Ø
AD 68Ø ONFGOTO63Ø,66Ø,63Ø
CM 69Ø PRINT"{DOWN}{RVS} LOAD DATA ":OP=1:GOTO71Ø
PC 7ØØ PRINT"{DOWN}{RVS} SAVE FILE ":OP=Ø
RX 71Ø IN$=N$:INPUT"{DOWN}FILENAME[4]";IN$:IF IN$=
   N$ THEN22Ø
PR 72Ø F=Ø:PRINT"{DOWN}{BLK}{RVS}T{OFF}APE OR
   {RVS}D{OFF}ISK: [4]";
FP 73Ø GET A$:IF A$="T"THEN PRINT"T{DOWN}":GOTO88Ø
HQ 74Ø IF A$<>"D"THEN73Ø
HH 75Ø PRINT"D{DOWN}":OPEN15,8,15,"IØ:":B=EA-SA:IN
   $="Ø:"+IN$:IF OP THEN81Ø
SQ 76Ø OPEN 1,8,8,IN$+",P,W":GOSUB86Ø:IF A THEN22Ø
FJ 77Ø AH=INT(SA/256):AL=SA-(AH*256):PRINT#1,CHR$(
   AL);CHR$(AH);
PE 78Ø FOR I=Ø TO B:PRINT#1,CHR$(PEEK(BS+I));:IF S
   T THEN8ØØ
FC 79Ø NEXT:CLOSE1:CLOSE15:GOTO94Ø
GS 8ØØ GOSUB1Ø6Ø:PRINT"{DOWN}{BLK}ERROR DURING SAV
   E:[4]":GOSUB86Ø:GOTO22Ø
MA 81Ø OPEN 1,8,8,IN$+",P,R":GOSUB86Ø:IF A THEN22Ø
GE 82Ø GET#1,A$,B$:AD=ASC(A$+Z$)+256*ASC(B$+Z$):IF
    AD<>SA THEN F=1:GOTO85Ø
KH 83Ø FOR I=Ø TO B:GET#1,A$:POKE BS+I,ASC(A$+Z$):
   IF ST AND(I<>B)THEN F=2:AD=I:I=B
FA 84Ø NEXT:IF ST<>64 THEN F=3
FQ 85Ø CLOSE1:CLOSE15:ON ABS(F>Ø)+1 GOTO96Ø,97Ø
SA 86Ø INPUT#15,A,A$:IF A THEN CLOSE1:CLOSE15:GOSU
   B1Ø6Ø:PRINT"{RVS}ERROR: "A$
GQ 87Ø RETURN
EJ 88Ø POKE183,PEEK(FA+2):POKE187,PEEK(FA+3):POKE1
   88,PEEK(FA+4):IFOP=ØTHEN92Ø
HJ 89Ø SYS 63466:IF(PEEK(783)AND1)THEN GOSUB1Ø6Ø:P
   RINT"{DOWN}{RVS} FILE NOT FOUND ":GOTO69Ø
CS 9ØØ AD=PEEK(829)+256*PEEK(83Ø):IF AD<>SA THEN F
   =1:GOTO97Ø
SC 91Ø A=PEEK(831)+256*PEEK(832)-1:F=F-2*(A<EA)-3*
   (A>EA):AD=A-AD:GOTO93Ø
KM 92Ø A=SA:B=EA+1:GOSUB1Ø1Ø:POKE78Ø,3:SYS 63338
JF 93Ø A=BS:B=BS+(EA-SA)+1:GOSUB1Ø1Ø:ON OP GOTO95Ø
   :SYS 63591
AE 94Ø GOSUB1Ø8Ø:PRINT"{BLU}** SAVE COMPLETED **":
   GOTO22Ø
```

```
AX 950 POKE147,0:SYS 63562:IF ST<>64 THEN970
FR 960 GOSUB1080:PRINT"{BLU}** LOAD COMPLETED **":
       GOTO220
DP 970 GOSUB1060:PRINT"{BLK}{RVS}ERROR DURING LOAD
       :{DOWN}{4}":ON F GOSUB980,990,1000:GOTO220
PP 980 PRINT"INCORRECT STARTING ADDRESS (";:GOSUB3
       60:PRINT")":RETURN
GR 990 PRINT"LOAD ENDED AT ";:AD=SA+AD:GOSUB360:PR
       INT D$:RETURN
FD 1000 PRINT"TRUNCATED AT ENDING ADDRESS":RETURN
RX 1010 AH=INT(A/256):AL=A-(AH*256):POKE193,AL:POK
        E194,AH
FF 1020 AH=INT(B/256):AL=B-(AH*256):POKE174,AL:POK
        E175,AH:RETURN
FX 1030 IF AD<SA OR AD>EA THEN1050
HA 1040 IF(AD>511 AND AD<40960)OR(AD>49151 AND AD<
        53248)THEN GOSUB1080:F=0:RETURN
HC 1050 GOSUB1060:PRINT"{RVS} INVALID ADDRESS
        {DOWN}{BLK}":F=1:RETURN
AR 1060 POKE SD+5,31:POKE SD+6,208:POKE SD,240:POK
        E SD+1,4:POKE SD+4,33
DX 1070 FOR S=1 TO 100:NEXT:GOTO1090
PF 1080 POKE SD+5,8:POKE SD+6,240:POKE SD,0:POKE S
        D+1,90:POKE SD+4,17
AC 1090 FOR S=1 TO 100:NEXT:POKE SD+4,0:POKE SD,0:
        POKE SD+1,0:RETURN
```

Program F-3. Pascal Interpreter for the Commodore 64 and 128 (in 64 mode)

For mistake-proof program entry, use "MLX" to type in this program. This program is not a full-featured interpreter; see Appendix E for a list of the interpreter's features.

Starting address: $0801
Ending address: $3C70

```
0801:0B 08 0A 00 9E 32 30 37 2F
0809:35 00 00 00 C8 D0 02 E6 28
0811:FC 60 88 C4 FF D0 02 C6 23
0819:FC 60 A9 FF 8D 00 50 A9 AB
0821:0F A8 A2 08 20 BA FF 20 C3
0829:C0 FF AD 21 D0 8D 86 02 2D
0831:20 A5 0B A9 00 8D 20 D0 FD
0839:8D 21 D0 A9 0E A0 0E 20 3C
```

```
0841:1E AB A9 70 8D 00 03 A9 A3
0849:0E 8D 01 03 A9 C1 8D 18 9B
0851:03 EA EA EA EA EA 20 CE BB
0859:0A C0 00 D0 15 4C 57 08 3C
0861:A9 0D 20 D2 FF 4C 57 08 A2
0869:A9 37 A0 0E 20 1E AB 4C 2E
0871:61 08 A0 00 B9 00 02 F0 0B
0879:07 99 3C 03 C8 4C 75 08 95
0881:A9 0D 99 3C 03 A9 00 8D EC
0889:03 C5 20 7B 12 AD 03 C5 5B
0891:8D 04 C5 A9 20 20 27 12 9E
0899:90 05 A9 0D 20 27 12 AE A9
08A1:03 C5 BD 3C 03 C9 31 90 52
08A9:BF C9 3A 90 03 4C 4D 0A 4A
08B1:20 00 30 AD C5 02 C9 03 7F
08B9:B0 AE A9 F2 A0 03 20 A2 26
08C1:BB A9 00 8D 5A C7 20 F7 1D
08C9:B7 AE 5A C7 D0 9A C9 FF AD
08D1:F0 96 8C 52 C7 8D 53 C7 99
08D9:AC 04 C5 B9 3C 03 C9 0D 24
08E1:F0 08 8C 03 C5 20 7B 12 E5
08E9:90 03 4C 65 0C 20 DD 0C 8B
08F1:B1 FB C9 FF F0 2B CD 53 36
08F9:C7 90 14 D0 5C 20 0D 08 27
0901:B1 FB CD 52 C7 D0 03 4C 9D
0909:D4 09 90 03 4C 57 09 20 FB
0911:0D 08 B1 FB C9 0D F0 03 09
0919:4C 10 09 20 0D 08 4C F1 8B
0921:08 AD 53 C7 91 FB 20 0D 53
0929:08 AD 52 C7 91 FB 20 0D 3B
0931:08 A2 00 8E 03 C5 20 7B C3
0939:12 AE 03 C5 BD 3C 03 91 33
0941:FB C9 0D F0 07 20 0D 08 4F
0949:E8 4C 3D 09 20 0D 08 A9 0A
0951:FF 91 FB 4C 57 08 20 13 3A
0959:08 20 E7 0C A2 00 8E 03 6A
0961:C5 20 7B 12 AE 03 C5 BD B9
0969:3C 03 C9 0D F0 04 E8 4C 1A
0971:68 09 8A 18 69 03 38 ED 82
0979:04 0E AA 8E 54 C7 18 A5 E6
0981:FB 6D 55 C7 8D 57 C7 90 FD
0989:0B A5 FC 8D 58 C7 EE 58 1B
0991:C7 4C 9A 09 A5 FC 8D 58 13
```

```
0999:C7 AD 57 C7 18 6D 54 C7 49
09A1:8D 59 C7 90 0C AD 58 C7 62
09A9:8D 5A C7 EE 5A C7 4C B8 44
09B1:09 AD 58 C7 8D 5A C7 20 C0
09B9:18 0D 20 F5 0C AD 53 C7 04
09C1:91 FB 20 0D 08 AD 52 C7 D3
09C9:91 FB 20 E7 0C 20 4F 0D B2
09D1:4C 57 08 20 E7 0C A2 00 97
09D9:20 0D 08 B1 FB C9 0D F0 6D
09E1:04 E8 4C D9 09 8E 54 C7 4A
09E9:20 0D 08 A5 FB 8D 57 C7 37
09F1:A5 FC 8D 58 C7 8C 08 0E DB
09F9:AD 57 C7 18 6D 08 0E 90 6B
0A01:03 EE 58 C7 8D 57 C7 A2 D5
0A09:00 8E 03 C5 20 7B 12 AE 3F
0A11:03 C5 BD 3C 03 C9 0D F0 DD
0A19:04 E8 4C 13 0A EC 54 C7 98
0A21:F0 4C 90 50 8A 38 ED 54 3D
0A29:C7 18 6D 57 C7 8D 59 C7 39
0A31:90 0B AD 58 C7 69 01 8D FE
0A39:5A C7 4C 44 0A AD 58 C7 B9
0A41:8D 5A C7 20 18 0D 20 4F 32
0A49:0D 4C 57 08 20 DD 0C A9 9C
0A51:48 A2 0E A0 00 20 50 11 30
0A59:90 03 4C 69 08 0A A8 B9 0A
0A61:A9 0A 8D C9 02 B9 AA 0A 71
0A69:8D CA 02 6C C9 02 20 4F E3
0A71:0D 4C 57 08 AD 54 C7 8E 67
0A79:54 C7 38 ED 54 C7 8D 56 C2
0A81:C7 AD 57 C7 38 ED 56 C7 3A
0A89:8D 59 C7 B0 0C AD 58 C7 4E
0A91:8D 5A C7 CE 5A C7 4C A0 14
0A99:0A AD 58 C7 8D 5A C7 20 2B
0AA1:6E 0D 20 4F 0D 4C 57 08 79
0AA9:DC 0A B7 0A 1B 08 1C 0B 82
0AB1:4E 0C C5 0B 4B 0C AD 00 3F
0AB9:50 C9 FF D0 03 4C 57 08 75
0AC1:A9 24 8D 00 03 A9 10 8D D1
0AC9:01 03 4C 40 10 20 60 A5 14
0AD1:AC 00 02 C0 20 D0 03 AC 7F
0AD9:01 02 60 AD 00 50 C9 FF AA
0AE1:D0 03 4C 11 0B 20 DD 0C 5A
0AE9:A9 0D 20 D2 FF 20 0D 08 E9
```

```
ØAF1:2Ø ØD Ø8 B1 FB 2Ø D2 FF 7B
ØAF9:C9 ØD FØ Ø3 4C F1 ØA 2Ø E2
ØBØ1:ØD Ø8 B1 FB C9 FF FØ Ø8 CD
ØBØ9:2Ø ØD Ø8 2Ø E1 FF DØ EØ Ø7
ØB11:A9 Ø4 2Ø C3 FF 2Ø CC FF 57
ØB19:4C 61 Ø8 2Ø 94 ØD 9Ø Ø3 AD
ØB21:4C 69 Ø8 AD Ø6 ØE C9 Ø1 9Ø
ØB29:DØ ØF A9 ØD 2Ø D2 FF A9 67
ØB31:CB 8D 56 C7 AØ ØØ 4C 48 BD
ØB39:ØB EE Ø8 ØE EE Ø8 ØE A9 CF
ØB41:C9 8D 56 C7 A9 Ø2 A8 AE 3C
ØB49:Ø6 ØE 2Ø BA FF AD Ø8 ØE 6A
ØB51:AE 56 C7 AØ Ø2 2Ø BD FF 63
ØB59:2Ø CØ FF A2 Ø2 2Ø C6 FF F7
ØB61:AØ ØØ 2Ø E4 FF AA 2Ø B7 BC
ØB69:FF 29 4Ø DØ 1F 8A 91 FB 21
ØB71:2Ø ØD Ø8 C9 ØD FØ Ø3 4C F6
ØB79:63 ØB 2Ø E4 FF 91 FB 2Ø B4
ØB81:B7 FF 29 4Ø DØ Ø6 2Ø ØD 88
ØB89:Ø8 4C 63 ØB A9 Ø2 2Ø C3 2D
ØB91:FF 2Ø 9B ØB 2Ø CC FF 4C 54
ØB99:61 Ø8 A9 Ø2 8D 86 Ø2 A9 EB
ØBA1:ØD 2Ø D2 FF A2 ØF 2Ø C6 F8
ØBA9:FF 2Ø CF FF C9 ØD FØ Ø6 2C
ØBB1:2Ø D2 FF 4C AA ØB 2Ø CC DF
ØBB9:FF A9 ØD 2Ø D2 FF A9 Ø3 CA
ØBC1:8D 86 Ø2 6Ø 2Ø 94 ØD 9Ø 84
ØBC9:Ø3 4C 69 Ø8 AD Ø8 ØE 18 E3
ØBD1:69 Ø6 8D Ø8 ØE A2 ØØ BD Ø9
ØBD9:6C ØE 99 CB Ø2 C8 E8 EØ 7F
ØBE1:Ø4 DØ F4 AØ ØØ B1 FB C9 5F
ØBE9:FF FØ 51 A9 Ø2 A8 AE Ø6 17
ØBF1:ØE 2Ø BA FF AD Ø6 ØE C9 D9
ØBF9:Ø1 DØ 12 A9 ØD 2Ø D2 FF 3Ø
ØC01:AD Ø8 ØE 38 E9 Ø6 A2 CB AF
ØC09:AØ Ø2 4C 15 ØC AD Ø8 ØE Ø2
ØC11:A2 C9 AØ Ø2 2Ø BD FF 2Ø 39
ØC19:CØ FF A2 Ø2 2Ø C9 FF AØ CE
ØC21:ØØ B1 FB 2Ø D2 FF C9 ØD 5E
ØC29:FØ Ø6 2Ø ØD Ø8 4C 22 ØC D1
ØC31:2Ø ØD Ø8 B1 FB C9 FF DØ 9Ø
ØC39:E8 2Ø D2 FF A9 Ø2 2Ø C3 81
ØC41:FF 2Ø CC FF 2Ø 9B ØB 4C CC
```

```
ØC49:61 Ø8 2Ø E2 FC A9 Ø4 AA 87
ØC51:AØ ØØ 2Ø BA FF 98 2Ø BD C9
ØC59:FF 2Ø CØ FF A2 Ø4 2Ø C9 CØ
ØC61:FF 4C DC ØA 2Ø DD ØC B1 ØB
ØC69:FB CD 53 C7 9Ø 14 FØ Ø3 93
ØC71:4C 61 Ø8 2Ø ØD Ø8 B1 FB F2
ØC79:CD 52 C7 FØ 14 9Ø Ø3 4C 4A
ØC81:61 Ø8 2Ø ØD Ø8 B1 FB C9 E9
ØC89:ØD DØ F7 2Ø ØD Ø8 4C 68 E6
ØC91:ØC 2Ø 13 Ø8 A5 FB 18 8C 74
ØC99:55 C7 6D 55 C7 8D 59 C7 4Ø
ØCA1:9Ø ØB A5 FC 8D 5A C7 EE 9D
ØCA9:5A C7 4C B3 ØC A5 FC 8D 24
ØCB1:5A C7 2Ø ØD Ø8 B1 FB C9 86
ØCB9:ØD FØ Ø3 4C B3 ØC 2Ø ØD D4
ØCC1:Ø8 A5 FB 18 8C 55 C7 6D FE
ØCC9:55 C7 8D 57 C7 9Ø Ø2 E6 11
ØCD1:FC A5 FC 8D 58 C7 2Ø 6E DA
ØCD9:ØD 4C 57 Ø8 A9 ØØ 85 FB 4B
ØCE1:A8 A9 5Ø 85 FC 6Ø 8C 55 F2
ØCE9:C7 A5 FB 8D ØA ØE A5 FC 78
ØCF1:8D ØB ØE 6Ø AC 55 C7 AD 53
ØCF9:ØA ØE 85 FB AD ØB ØE 85 46
ØDØ1:FC 6Ø AD 57 C7 85 FB AD D6
ØDØ9:58 C7 85 FC AD 59 C7 85 A9
ØD11:FD AD 5A C7 85 FE 6Ø 2Ø 66
ØD19:Ø3 ØD AD ØD ØE 18 65 FC 17
ØD21:85 FC AD ØD ØE 18 65 FE 5E
ØD29:85 FE AC ØC ØE FØ Ø9 88 EA
ØD31:B1 FB 91 FD CØ ØØ DØ F7 D4
ØD39:AE ØD ØE FØ 1Ø C6 FC C6 1B
ØD41:FE 88 B1 FB 91 FD CØ ØØ F8
ØD49:DØ F7 CA DØ FØ 6Ø A2 ØØ 7E
ØD51:8E Ø3 C5 2Ø 7B 12 AE Ø3 B2
ØD59:C5 2Ø F5 ØC 2Ø ØD Ø8 BD EØ
ØD61:3C Ø3 91 FB C9 ØD FØ Ø4 B4
ØD69:E8 4C 5D ØD 6Ø 2Ø Ø3 ØD 1E
ØD71:AØ ØØ AE ØD ØE FØ ØE B1 84
ØD79:FB 91 FD C8 DØ F9 E6 FC 7B
ØD81:E6 FE CA DØ F2 AE ØC ØE AD
ØD89:FØ Ø8 B1 FB 91 FE C8 CA F8
ØD91:DØ F8 6Ø A9 ØØ 8D Ø8 ØE 4D
ØD99:A9 Ø8 8D Ø6 ØE A9 3Ø 8D A1
```

```
ØDA1:C9 Ø2 A9 3A 8D CA Ø2 2Ø B5
ØDA9:DD ØC A9 ØØ 8D Ø3 C5 2Ø ØF
ØDB1:7B 12 9Ø Ø2 38 6Ø A9 2Ø F6
ØDB9:2Ø ØA 12 BØ F7 2Ø 7B 12 FC
ØDC1:AE Ø3 C5 AØ ØØ BD 3C Ø3 29
ØDC9:C9 ØD FØ 17 C9 2Ø FØ 13 5F
ØDD1:C9 2C FØ 19 99 CB Ø2 E8 74
ØDD9:C8 EE Ø8 ØE CØ 11 FØ D4 F6
ØDE1:4C C6 ØD 8E Ø3 C5 2Ø 7B 49
ØDE9:12 9Ø Ø2 18 6Ø E8 BD 3C 51
ØDF1:Ø3 C9 31 FØ Ø6 C9 38 FØ ED
ØDF9:Ø2 38 6Ø 38 E9 3Ø 8D Ø6 E3
ØE01:ØE 18 6Ø ØØ ØØ ØØ ØØ ØØ 36
ØE09:ØØ ØØ ØØ ØØ 5Ø 93 2Ø 2Ø 56
ØE11:2Ø 2Ø 1C 2A 2A 2A 2Ø 43 E8
ØE19:2D 36 34 2Ø 5Ø 41 53 43 53
ØE21:41 4C 2Ø 49 4E 54 45 52 2A
ØE29:5Ø 52 45 54 45 52 2Ø 2A CD
ØE31:2A 2A 9F ØD ØD ØØ ØD 1C 5Ø
ØE39:53 59 4E 54 41 58 2Ø 45 55
ØE41:52 52 4F 52 9F ØD ØØ 4C A7
ØE49:49 53 54 ØØ 52 55 4E ØØ ED
ØE51:4E 45 57 ØØ 4C 4F 41 44 37
ØE59:ØØ 4C 5Ø 52 49 4E 54 ØØ E3
ØE61:53 41 56 45 ØØ 58 ØØ 44 3C
ØE69:55 4D 2A 2C 53 2C 57 6Ø E5
ØE71:AE 13 Ø3 DØ Ø1 AA AØ ØØ ØB
ØE79:A9 ØD 2Ø D2 FF A9 1C 2Ø DD
ØE81:D2 FF B9 68 ØF 85 Ø2 C9 21
ØE89:FF FØ 1D 8A D9 68 ØF FØ AD
ØE91:Ø4 C8 4C 83 ØE 98 ØA AA 35
ØE99:BD 77 ØF E8 BC 77 ØF 2Ø E4
ØEA1:1E AB A5 Ø2 C9 Ø8 FØ ØA E6
ØEA9:A9 18 AØ 1Ø 2Ø 1E AB 2Ø A6
ØEB1:C5 ØE A9 ØØ 2Ø C3 FF AA 24
ØEB9:E8 8A C9 ØF FØ Ø3 4C B5 F8
ØEC1:ØE 4C 43 Ø8 AC 1Ø C5 AD BF
ØEC9:11 C5 85 FB AD 12 C5 85 17
ØED1:FC B1 FB 85 62 2Ø ØD Ø8 66
ØED9:B1 FB 85 63 A2 9Ø 38 2Ø 9C
ØEE1:49 BC 2Ø DD BD A9 ØØ AØ E8
ØEE9:Ø1 2Ø 1E AB A9 ØD 2Ø D2 A1
ØEF1:FF 2Ø D2 FF 6Ø 2Ø BD 1Ø 7F
```

```
ØEF9:9Ø Ø2 38 6Ø 2Ø E3 1D C9 8Ø
ØF01:Ø7 DØ Ø6 EE 17 C5 4C 61 5Ø
ØF09:ØF C9 Ø5 DØ Ø6 EE 17 C5 AE
ØF11:4C 61 ØF C9 Ø2 FØ 46 A9 36
ØF19:ØØ 8D Ø6 ØE AA BD 3C Ø3 Ø4
ØF21:C9 ØD FØ 3C AD Ø6 ØE DØ BB
ØF29:1C C9 27 DØ 11 AD Ø6 ØE 13
ØF31:DØ Ø6 EE Ø6 ØE 4C 46 ØF B4
ØF39:CE Ø6 ØE 4C 46 ØF BD 3C EC
ØF41:Ø3 C9 42 FØ Ø4 E8 4C 1E 25
ØF49:ØF 8A 48 2Ø E3 1D 68 AA AB
ØF51:AD Ø2 C5 C9 Ø9 DØ EE EE 74
ØF59:17 C5 4C 61 ØF CE 17 C5 BB
ØF61:AD 17 C5 DØ 9Ø 18 6Ø Ø1 88
ØF69:Ø2 Ø3 Ø4 Ø5 Ø6 Ø7 Ø8 Ø9 7F
ØF71:ØA ØE ØF 14 16 FF 93 ØF 22
ØF79:9A ØF A5 ØF B3 ØF BC ØF BØ
ØF81:C1 ØF A5 ØF Ø1 1Ø C7 ØF DØ
ØF89:A5 ØF CC ØF DD ØF E6 ØF DØ
ØF91:F7 ØF 53 59 4E 54 41 58 ØE
ØF99:ØØ 49 44 45 4E 54 49 46 83
ØFA1:49 45 52 ØØ 49 4E 54 45 71
ØFA9:47 45 52 2Ø 45 52 52 4F 7Ø
ØFB1:52 ØØ 53 55 42 52 41 4E E4
ØFB9:47 45 ØØ 46 49 4C 45 ØØ 37
ØFC1:53 54 41 43 4B ØØ 43 41 1D
ØFC9:53 45 ØØ 49 4C 4C 45 47 DC
ØFD1:41 4C 2Ø 51 55 41 4E 54 5D
ØFD9:49 54 59 ØØ 4F 56 45 52 8D
ØFE1:46 4C 4F 57 ØØ 44 49 56 8F
ØFE9:49 53 49 4F 4E 2Ø 42 59 7Ø
ØFF1:2Ø 5A 45 52 4F ØØ 44 41 C8
ØFF9:54 41 2Ø 54 59 5Ø 45 ØØ 72
1ØØ1:4D 49 53 53 49 4E 47 2Ø EB
1ØØ9:27 45 4E 44 2E 27 2Ø 45 AF
1Ø11:52 52 4F 52 9F ØD ØØ 1C 4B
1Ø19:2Ø 45 52 52 4F 52 2Ø 49 57
1Ø21:4E 9F ØØ EØ 16 FØ Ø3 8E 67
1Ø29:13 Ø3 6Ø 52 4F 55 54 31 6E
1Ø31:2Ø 4C 44 41 2Ø 28 24 46 41
1Ø39:42 29 2C 59 ØØ 4B 1Ø 4C 79
1Ø41:EA 12 4D 41 58 49 4E 54 F1
1Ø49:ØØ Ø1 8F 7F FE ØØ ØØ 5Ø DB
```

```
1051:52 4F 47 52 41 4D 00 43 FE
1059:4F 4E 53 54 00 54 59 50 B8
1061:45 00 56 41 52 00 50 52 88
1069:4F 43 45 44 55 52 00 46 29
1071:55 4E 43 54 49 4F 4E 00 A1
1079:42 45 47 49 4E 00 4F 46 E0
1081:00 49 4E 54 45 47 45 52 27
1089:00 52 45 41 4C 00 42 4F 31
1091:4F 4C 45 41 4E 00 43 48 6A
1099:41 52 00 54 45 58 54 00 68
10A1:41 52 52 41 59 00 50 41 02
10A9:43 4B 45 44 00 53 45 54 57
10B1:00 45 4E 44 00 4C 41 42 27
10B9:45 4C 00 2A A5 FB 8D E0 4B
10C1:CF A5 FC 8D E1 CF 8C E2 F5
10C9:CF 20 94 12 A5 FB 8D 11 D6
10D1:C5 A5 FC 8D 12 C5 8C 10 87
10D9:C5 A2 00 B1 FB C9 FF D0 78
10E1:07 A9 08 8D 13 03 38 60 3F
10E9:20 0D 08 20 0D 08 B1 FB 48
10F1:C9 0D F0 0A 9D 3C 03 E8 C5
10F9:20 0D 08 4C EF 10 9D 3C 6A
1101:03 A9 00 8D 03 C5 A9 20 8A
1109:20 0A 12 20 7B 12 20 0D 73
1111:08 20 A2 12 AD 03 C5 8D 47
1119:04 C5 AE 03 C5 BD 3C 03 55
1121:C9 28 D0 0B E8 BD 3C 03 B6
1129:C9 2A D0 03 4C CA 10 AD 60
1131:E0 CF 85 FB AD E1 CF 85 42
1139:FC AC E2 CF 18 60 20 44 25
1141:12 A9 50 A2 10 A0 00 4C 5A
1149:50 11 20 44 12 A0 00 85 B8
1151:FB 86 FC A9 00 8D 02 C5 4D
1159:8D ED 1A 8D 27 C3 AE 03 82
1161:C5 BD 3C 03 C9 20 F0 2D 6B
1169:C9 0D F0 30 C9 3B F0 2C 1E
1171:C9 28 F0 21 C9 2C F0 1D B0
1179:C9 5B F0 19 C9 29 F0 15 F0
1181:D1 FB D0 24 E8 20 0D 08 D1
1189:EE 27 C3 AD 27 C3 C9 08 24
1191:F0 03 4C 62 11 B1 FB D0 B4
1199:0F 4C 52 12 B1 FB D0 08 E8
11A1:A9 FF 8D ED 1A 4C 52 12 E1
```

```
11A9:A9 ØØ 8D 27 C3 EE Ø2 C5 68
11B1:AD Ø3 C3 C9 Ø7 DØ 33 B1 14
11B9:FB FØ Ø6 2Ø ØD Ø8 4C B8 B2
11C1:11 2Ø ØD Ø8 2Ø ØD Ø8 2Ø FB
11C9:ØD Ø8 2Ø ØD Ø8 B1 FB C9 12
11D1:2A FØ 2E A5 FC C5 4Ø 9Ø 75
11D9:22 DØ 26 A5 FB C5 3F 9Ø 66
11E1:1A DØ 1E C4 4E 9Ø 14 4C 7E
11E9:Ø2 12 B1 FB FØ ØA C9 2A F4
11F1:FØ ØF 2Ø ØD Ø8 4C EB 11 7F
11F9:2Ø ØD Ø8 AE Ø3 C5 4C 62 85
12Ø1:11 A9 FF 8D Ø2 C5 4C 52 Ø3
12Ø9:12 8D EC 1A AE Ø3 C5 BD A3
1211:3C Ø3 CD EC 1A FØ ØA C9 ØF
1219:ØD FØ Ø4 E8 4C 1Ø 12 38 ØE
1221:6Ø 8E Ø3 C5 18 6Ø 8D EC 2Ø
1229:1A AE Ø4 C5 BD 3C Ø3 CD 95
1231:EC 1A FØ ØA C9 ØD FØ Ø4 79
1239:E8 4C 2D 12 38 6Ø 8E Ø4 1Ø
1241:C5 18 6Ø 8E 12 C3 A6 FB 2C
1249:8E 1Ø C3 A6 FC 8E 11 C3 A3
1251:6Ø AD Ø3 C3 C9 Ø7 FØ Ø3 FC
1259:2Ø 6D 12 AD Ø2 C5 C9 FF CØ
1261:DØ Ø5 38 AD Ø2 C5 6Ø 18 11
1269:AD Ø2 C5 6Ø AD 1Ø C3 85 5E
1271:FB AD 11 C3 85 FC AC 12 E8
1279:C3 6Ø AE Ø3 C5 BD 3C Ø3 3E
1281:C9 2Ø DØ Ø4 E8 4C 7E 12 74
1289:C9 ØD DØ Ø2 38 6Ø 8E Ø3 73
1291:C5 18 6Ø AC AØ C5 AD A1 8E
1299:C5 85 FC AD A2 C5 85 FB AF
12A1:6Ø 98 8D AØ C5 A5 FC 8D 24
12A9:A1 C5 A5 FB 8D A2 C5 6Ø 67
12B1:AE Ø3 C5 BD 3C Ø3 C9 41 45
12B9:9Ø 21 C9 5B BØ 1D E8 EC 16
12C1:Ø4 C5 FØ 1C BD 3C Ø3 C9 E7
12C9:2Ø FØ 15 C9 3Ø 9Ø ØC C9 1F
12D1:5B BØ Ø8 C9 3A 9Ø E7 C9 1B
12D9:41 BØ E3 A9 Ø2 8D 13 Ø3 51
12E1:6Ø C8 84 4E DØ Ø2 E6 4Ø 7A
12E9:6Ø A9 ØØ 8D EE 1A 8D 13 8F
12F1:Ø3 8D Ø3 C3 85 Ø2 8D 5E 45
12F9:C1 8D 23 C3 8D 8Ø C1 8D 82
```

```
1301:81 C1 8D 01 C9 A9 2A 8D F0
1309:50 CA A9 00 AA A8 85 3F 81
1311:85 4E A9 60 85 40 BD 43 B4
1319:10 91 3F 20 E2 12 E8 E0 A7
1321:0D F0 03 4C 17 13 A9 00 87
1329:8D A0 C5 8D A2 C5 A9 50 9F
1331:8D A1 C5 A9 2A 8D 50 CA CC
1339:20 BD 10 90 03 4C B3 0E A8
1341:20 3F 11 F0 05 A9 07 4C A1
1349:71 0E 20 BD 10 90 03 4C A0
1351:B3 0E 20 3F 11 8E 03 C5 5B
1359:20 7B 12 AD 02 C5 C9 01 47
1361:D0 03 4C C8 13 A4 4E A9 38
1369:2A 91 3F 20 E2 12 AD 02 AF
1371:C5 C9 02 D0 03 4C B7 14 07
1379:A9 00 8D 08 C3 8D 09 C3 D0
1381:AD 02 C5 C9 03 D0 03 4C 02
1389:70 15 A5 3F 8D 0C C3 A5 9F
1391:40 8D 0D C3 A5 4E 8D 0E A8
1399:C3 A8 A9 2A 91 3F AD 02 8A
13A1:C5 C9 04 D0 03 4C 1F 19 4B
13A9:AD 02 C5 C9 05 D0 03 4C 3A
13B1:2A 19 AD 02 C5 C9 06 D0 3B
13B9:09 A9 2A A4 4E 91 3F 4C E1
13C1:E0 1D A9 02 4C 71 0E AD E6
13C9:ED 1A C9 FF D0 03 4C 91 63
13D1:14 A9 00 85 02 A0 00 8C E3
13D9:04 C5 A9 3D 20 27 12 90 CE
13E1:05 A9 01 4C 71 0E 20 B1 8F
13E9:12 AD 13 03 F0 03 4C 71 B4
13F1:0E A9 08 8D EC 1A AE 03 93
13F9:C5 A4 4E BD 3C 03 C9 20 73
1401:F0 0F C9 3D F0 0B 91 3F 88
1409:E8 20 E2 12 CE EC 1A D0 5A
1411:EA A9 00 91 3F 20 E2 12 84
1419:AE 04 C5 E8 8E 03 C5 20 0D
1421:7B 12 A5 02 C9 C8 D0 03 76
1429:4C D7 14 A9 3B 20 27 12 45
1431:90 05 A9 0D 20 27 12 20 CA
1439:00 30 AD 13 03 F0 03 4C 82
1441:71 0E A4 4E AD C5 02 91 39
1449:3F 20 E2 12 C9 03 B0 12 64
1451:A2 00 BD F2 03 91 3F 20 AE
```

```
1459:E2 12 E8 E0 05 D0 F3 4C 42
1461:91 14 C9 03 D0 0B AD A2 71
1469:03 91 3F 20 E2 12 4C 91 EA
1471:14 C9 04 D0 16 A2 00 BD 9C
1479:A2 03 91 3F F0 07 20 E2 A0
1481:12 E8 4C 78 14 20 E2 12 F6
1489:4C 91 14 A9 0E 4C 71 0E EB
1491:20 BD 10 90 03 4C 43 08 1C
1499:98 AA 20 3F 11 C9 08 B0 21
14A1:12 A4 4E A9 2A 91 3F 20 96
14A9:E2 12 8E 03 C5 20 7B 12 81
14B1:4C 6F 13 4C D2 13 A5 3F 70
14B9:8D 08 C3 A5 40 8D 09 C3 8B
14C1:A5 4E 8D 0A C3 AD ED 1A 6D
14C9:C9 FF D0 03 4C 4C 15 A9 88
14D1:C8 85 02 4C D6 13 20 3F 47
14D9:11 C9 08 90 07 C9 10 B0 37
14E1:03 4C 24 15 A9 2E 20 27 E1
14E9:12 90 38 A9 07 8D 03 C3 19
14F1:AD 08 C3 AE 09 C3 AC 0A 11
14F9:C3 20 50 11 A2 00 8E 03 5C
1501:C3 C9 FF F0 19 20 0D 08 F9
1509:B1 FB 8D EC 1A 20 0D 08 FE
1511:B1 FB 8D ED 1A 20 0D 08 17
1519:B1 FB AA 4C 34 15 A9 02 80
1521:4C 71 0E AD 10 C5 AA AD 05
1529:11 C5 8D EC 1A AD 12 C5 3F
1531:8D ED 1A A4 4E AD EC 1A 48
1539:91 3F 20 E2 12 AD ED 1A 6B
1541:91 3F 20 E2 12 8A 91 3F 53
1549:20 E2 12 20 BD 10 90 03 D2
1551:4C B3 0E 20 3F 11 C9 08 2C
1559:B0 12 A4 4E A9 2A 91 3F 32
1561:20 E2 12 8E 03 C5 20 7B 6A
1569:12 4C 81 13 4C D0 14 A5 84
1571:3F 8D 0C C3 A5 40 8D 0D B2
1579:C3 A5 4E 8D 0E C3 A9 00 64
1581:8D 23 C3 8D 20 C3 8D 17 CE
1589:C5 8D 22 C3 8D 21 C3 AD A0
1591:ED 1A C9 FF D0 03 4C B5 53
1599:18 A0 00 84 02 84 FD 8C EA
15A1:5F C1 8C 60 C1 8C 04 C5 91
15A9:A9 3A 20 27 12 90 05 A9 34
```

```
15B1:Ø1 4C 71 ØE AC Ø4 C5 8C ØC
15B9:5F CØ AE Ø3 C5 8E Ø4 C5 FF
15C1:A9 2C 2Ø 27 12 9Ø Ø9 AE D5
15C9:Ø3 C5 8E Ø4 C5 4C 3Ø 16 CE
15D1:AD Ø4 C5 CD 5F CØ 9Ø Ø9 91
15D9:AD Ø3 C5 8D Ø4 C5 4C 3Ø 2D
15E1:16 2Ø B1 12 AD 13 Ø3 FØ 27
15E9:Ø3 4C 71 ØE AE Ø3 C5 A4 69
15F1:FD A9 ØØ 85 FE BD 3C Ø3 48
15F9:C9 2Ø FØ 15 99 ØØ C1 E8 B9
16Ø1:C8 E6 FE EC Ø4 C5 FØ Ø9 1C
16Ø9:A5 FE C9 Ø8 FØ Ø3 4C F6 A4
1611:15 A9 ØØ 99 ØØ C1 C8 84 E8
1619:FD A5 Ø2 C9 Ø7 FØ 21 AC 75
1621:Ø4 C5 C8 8C Ø3 C5 2Ø 7B 8D
1629:12 EE Ø4 C5 4C C1 15 AØ 2B
1631:ØØ 8C Ø4 C5 A9 3A 2Ø 27 FA
1639:12 A9 Ø7 85 Ø2 4C E2 15 2E
1641:88 A9 2A 99 ØØ C1 A2 ØØ 47
1649:8E Ø3 C5 A9 3A 2Ø ØA 12 49
1651:EE Ø3 C5 2Ø 7B 12 AC Ø3 FØ
1659:C5 8C Ø4 C5 2Ø 3F 11 C9 52
1661:ØF DØ Ø3 4C F6 18 C9 Ø8 22
1669:9Ø Ø7 C9 ØD BØ Ø3 4C F6 CA
1671:16 A9 2E 2Ø 27 12 9Ø 5E DB
1679:A9 Ø7 8D Ø3 C3 AD Ø8 C3 C6
1681:AE Ø9 C3 AC ØA C3 2Ø 5Ø 7A
1689:11 A2 ØØ 8E Ø3 C3 C9 FF 8A
1691:DØ Ø5 A9 Ø2 4C 71 ØE AD AE
1699:AØ C5 8D 14 C3 AD A1 C5 58
16A1:8D 15 C3 AD A2 C5 8D 16 8A
16A9:C3 EE 21 C3 B1 FB FØ Ø6 39
16B1:2Ø ØD Ø8 4C AD 16 2Ø ØD ØA
16B9:Ø8 B1 FB 8D A2 C5 2Ø ØD 28
16C1:Ø8 B1 FB 8D A1 C5 2Ø ØD 28
16C9:Ø8 B1 FB 8D AØ C5 2Ø BD D8
16D1:1Ø A9 3D 4C 4E 16 AC Ø4 Ø5
16D9:C5 C8 B9 3C Ø3 C9 2E FØ A2
16E1:Ø5 A9 Ø1 4C 71 ØE AD Ø3 Ø2
16E9:C5 8D Ø4 C5 2Ø 11 17 2Ø CC
16F1:D9 18 4C F5 17 2Ø D9 18 FE
16F9:AD Ø2 C5 38 E9 Ø7 8D C5 Ø6
17Ø1:Ø2 C9 Ø5 BØ Ø3 4C F5 17 9A
```

```
1709:AA E8 8E C5 02 4C F5 17 39
1711:A9 2E 20 27 12 90 06 A9 9E
1719:01 8D 13 03 60 A9 00 85 EC
1721:41 85 42 85 4F 20 00 30 1D
1729:AD 13 03 F0 01 60 AD C5 0D
1731:02 A6 41 9D 5F C1 E8 86 66
1739:41 C9 01 F0 50 C9 04 D0 2C
1741:03 4C C8 17 A9 04 8D 13 1A
1749:03 60 EE 04 C5 EE 04 C5 E6
1751:AC 04 C5 8C 03 C5 20 7B 43
1759:12 AC 03 C5 B9 3C 03 C9 07
1761:0D F0 10 C9 5D F0 0C C9 81
1769:2C F0 08 C9 3B F0 04 C8 F5
1771:4C 5D 17 8C 04 C5 20 00 40
1779:30 AD 13 03 F0 01 60 AD B7
1781:C5 02 CD 5F C1 F0 B2 A9 A3
1789:0F 8D 13 03 60 E6 42 A2 FA
1791:00 A4 41 BD F2 03 99 5F 23
1799:C1 C8 E8 E0 05 90 F4 84 DE
17A1:41 A5 42 C9 02 B0 03 4C E3
17A9:4B 17 A9 60 A0 C1 20 A2 6D
17B1:BB A9 65 A0 C1 20 5B BC E0
17B9:C9 FF F0 05 A9 04 8D 13 C6
17C1:03 A9 07 8D C5 02 60 E6 73
17C9:42 A4 41 AD A3 03 99 5F 01
17D1:C1 E6 41 A5 42 C9 02 B0 0B
17D9:03 4C 4B 17 A9 00 8D 62 42
17E1:C1 AD 60 C1 CD 61 C1 90 8C
17E9:05 A9 04 8D 13 03 A9 08 5E
17F1:8D C5 02 60 AD 13 03 F0 4F
17F9:03 4C 71 0E A2 00 A4 4E 78
1801:86 FE AD 23 C3 91 3F 20 1F
1809:E2 12 86 02 BD 00 C1 F0 82
1811:0F C9 2A F0 09 91 3F E8 85
1819:20 E2 12 4C 0D 18 E6 FE AE
1821:A9 28 91 3F 20 E2 12 E8 EF
1829:86 FD AD C5 02 91 3F 20 23
1831:E2 12 C9 06 D0 0F A9 FE 06
1839:91 3F 20 E2 12 91 3F 20 A9
1841:E2 12 4C 6F 18 C9 0B 90 76
1849:11 A2 00 A9 FE 91 3F 20 22
1851:E2 12 E8 E0 FA F0 17 4C B8
1859:4E 18 A8 B9 E1 1A 8D 5F 59
```

```
1861:CØ A5 4E 18 6D 5F CØ 9Ø A1
1869:Ø2 E6 4Ø 85 4E 98 A2 ØØ CE
1871:A4 4E AD C5 Ø2 C9 Ø7 FØ CF
1879:22 C9 ØA FØ ØE BD 6Ø C1 67
1881:FØ 26 91 3F 2Ø E2 12 E8 73
1889:4C 7E 18 BD 6Ø C1 91 3F CA
1891:E8 2Ø E2 12 C9 Ø1 DØ F3 A3
1899:4C A9 18 BD 6Ø C1 91 3F A5
18A1:E8 2Ø E2 12 EØ ØA DØ F3 9Ø
18A9:2Ø E2 12 A6 FD A5 FE DØ A4
18B1:Ø3 4C Ø3 18 2Ø BD 1Ø 9Ø Ø1
18B9:Ø3 4C B3 ØE 2Ø 3F 11 C9 BF
18C1:Ø8 BØ 12 8E Ø3 C5 2Ø 7B 38
18C9:12 A4 4E A9 2A 91 3F 2Ø C6
18D1:E2 12 4C 9F 13 4C 9A 15 8F
18D9:AD 21 C3 FØ 17 A9 ØØ 8D 9D
18E1:21 C3 AD 14 C3 8D AØ C5 E5
18E9:AD 15 C3 8D A1 C5 AD 16 1D
18F1:C3 8D A2 C5 6Ø A9 43 2Ø 68
18F9:ØA 12 BØ 1Ø 2Ø 3F 11 C9 B4
1901:ØB DØ ØC 8D C5 Ø2 2Ø D9 97
1909:18 4C F5 17 A9 Ø4 2C A9 E9
1911:Ø1 4C 71 ØE C9 Ø1 DØ Ø3 DC
1919:4C Ø8 1A 4C 55 1A EE Ø3 6F
1921:C5 2Ø 7B 12 A9 ØØ 4C 2C EØ
1929:19 A9 Ø1 8D 28 C3 EE 23 9C
1931:C3 A9 ØØ 8D 29 C3 A9 28 5C
1939:2Ø 27 12 9Ø Ø8 A9 ØD 2Ø B1
1941:27 12 CE Ø4 C5 AE Ø3 C5 5A
1949:AC 5E C1 8C 5D C1 48 A9 96
1951:Ø8 85 FC AD 23 C3 99 5Ø ØF
1959:CA C8 BD 3C Ø3 C9 2Ø FØ ØF
1961:ØE 99 5Ø CA E8 C8 C6 FC AC
1969:FØ Ø5 EC Ø4 C5 DØ EB A9 26
1971:ØØ 99 5Ø CA C8 8C 5E C1 B7
1979:68 C9 28 FØ Ø3 4C 1E 1A Ø6
1981:EE Ø4 C5 AE Ø4 C5 8E Ø3 27
1989:C5 2Ø 7B 12 AE Ø3 C5 BD Ø2
1991:3C Ø3 C9 29 DØ Ø3 4C 1E B7
1999:1A A9 29 2Ø 27 12 9Ø Ø5 12
19A1:A9 Ø1 4C 71 ØE AC Ø4 C5 7A
19A9:84 Ø2 AC Ø3 C5 8C Ø4 C5 92
19B1:A9 3B 2Ø 27 12 9Ø 11 EE E1
```

```
19B9:29 C3 AE Ø3 C5 8E Ø4 C5 AD
19C1:A9 29 2Ø 27 12 4C D3 19 ØC
19C9:AD Ø4 C5 C5 Ø2 9Ø Ø3 4C 8D
19D1:B8 19 2Ø AD 1A AE Ø3 C5 DC
19D9:AØ ØØ BD 3C Ø3 99 3C Ø3 D1
19E1:C8 E8 EC Ø4 C5 FØ Ø3 4C D4
19E9:DB 19 A9 ØD 99 3C Ø3 A2 BC
19F1:ØØ 8E Ø3 C5 8E Ø4 C5 2Ø B4
19F9:7B 12 A9 Ø1 8D 2Ø C3 A9 D1
1AØ1:ØØ 8D ED 1A 4C 8A 15 A9 58
1AØ9:ØØ 8D 2Ø C3 AD 29 C3 DØ 4B
1A11:Ø6 2Ø C7 1A 4C 81 19 CE 54
1A19:29 C3 4C 1E 1A 2Ø BD 1Ø 1B
1A21:BØ ØD 2Ø 3F 11 C9 Ø3 FØ 8F
1A29:Ø9 C9 Ø6 FØ 3B A9 Ø2 4C F4
1A31:71 ØE AD ED 1A C9 FF DØ FE
1A39:Ø3 4C 4B 1A A9 2Ø 2Ø ØA 25
1A41:12 2Ø 7B 12 AD Ø3 C5 8D A9
1A49:Ø4 C5 A9 Ø3 8D 2Ø C3 A9 74
1A51:ØØ 4C 8A 15 A9 ØØ 8D 2Ø C3
1A59:C3 2Ø BD 1Ø 9Ø Ø3 4C B3 ØD
1A61:ØE 2Ø 3F 11 C9 Ø6 DØ E2 88
1A69:AC 5E C1 AD 1Ø C5 99 5Ø B9
1A71:CA C8 AD 11 C5 99 5Ø CA Ø4
1A79:C8 AD 12 C5 99 5Ø CA C8 88
1A81:8C 5E C1 A9 2A 99 5Ø CA 89
1A89:A9 Ø1 8D 17 C5 2Ø F6 ØE AØ
1A91:9Ø Ø3 4C 71 ØE 2Ø BD 1Ø EB
1A99:9Ø Ø3 4C B3 ØE 2Ø 3F 11 1C
1AA1:8E Ø3 C5 2Ø 7B 12 AD Ø2 1A
1AA9:C5 4C 9F 13 A2 ØØ BD 3C C5
1AB1:Ø3 9D 8Ø C6 E8 C9 ØD DØ A4
1AB9:F5 AD Ø4 C5 8D 2A C3 AD 7B
1AC1:Ø3 C5 8D 5C C1 6Ø A2 ØØ 35
1AC9:BD 8Ø C6 9D 3C Ø3 E8 C9 39
1AD1:ØD DØ F5 AC 5C C1 8C Ø3 5Ø
1AD9:C5 AC 2A C3 8C Ø4 C5 6Ø FD
1AE1:ØØ Ø5 Ø5 Ø1 Ø1 Ø9 Ø2 Ø5 3D
1AE9:Ø1 Ø9 Ø9 ØØ ØØ ØØ E4 16 E1
1AF1:FØ Ø3 8E 13 Ø3 6Ø 2B 31 83
1AF9:ØØ ØB 1B 2C ØB 44 45 2Ø 2B
1BØ1:AB 2F A9 2Ø 2Ø ØA 12 2Ø 7D
1BØ9:7B 12 8E C9 Ø2 A9 3A 2Ø 3B
```

```
1B11:27 12 20 72 26 20 E7 0C 18
1B19:AD 19 C5 D0 05 A9 02 4C 51
1B21:71 0E 84 02 AD 01 C9 38 81
1B29:E9 0A 90 1B AA BD 02 C9 B4
1B31:C5 FB D0 13 E8 BD 02 C9 A0
1B39:C5 FC D0 0B E8 BD 02 C9 68
1B41:C5 02 D0 03 4C 8D 1C AE A4
1B49:03 C5 8E 98 03 A9 44 20 35
1B51:0A 12 B0 0D 20 E3 1D C9 8C
1B59:12 F0 28 EE 03 C5 4C 4E DE
1B61:1B AE 98 03 8E 03 C5 A9 C9
1B69:54 20 0A 12 B0 0D 20 E3 12
1B71:1D C9 11 F0 0B EE 03 C5 B9
1B79:4C 68 1B A9 01 4C 71 0E 18
1B81:A9 00 2C A9 06 8D F8 C9 CE
1B89:AE 03 C5 CA 8E F7 C9 AD D2
1B91:C9 02 8D 03 C5 8D 04 C5 41
1B99:A9 04 8D 41 C5 4C 71 2B D8
1BA1:A9 00 8D 41 C5 AD 19 C5 4F
1BA9:C9 01 F0 11 C9 07 F0 0D 8D
1BB1:C9 04 F0 11 C9 08 F0 0D 5A
1BB9:A9 02 4C 71 0E A9 00 8D 8A
1BC1:80 C1 4C CB 1B A9 01 8D FD
1BC9:80 C1 AE 01 C9 20 F5 0C 5D
1BD1:A5 FB 9D 02 C9 E8 A5 FC E7
1BD9:9D 02 C9 E8 98 9D 02 C9 30
1BE1:E8 AD F8 C9 9D 02 C9 E8 25
1BE9:AD 80 C1 9D 02 C9 E8 8E C0
1BF1:01 C9 AC F7 C9 C8 8C 03 BD
1BF9:C5 A9 20 20 0A 12 20 7B D7
1C01:12 AE 03 C5 8E C9 02 A9 F3
1C09:44 20 0A 12 B0 0D 20 E3 AB
1C11:1D C9 0B F0 0B EE 03 C5 9A
1C19:4C 08 1C A9 01 4C 71 0E C1
1C21:AE 03 C5 8E 80 C1 CA 8E 42
1C29:04 C5 AD C9 02 8D 03 C5 39
1C31:20 00 30 AD 13 03 D0 E5 86
1C39:AD C5 02 C9 01 F0 09 C9 3E
1C41:04 F0 19 A9 16 4C 71 0E 48
1C49:A0 00 AE 01 C9 B9 F2 03 D5
1C51:9D 02 C9 E8 C8 C0 05 D0 C4
1C59:F4 4C 6B 1C AE 01 C9 AD 09
1C61:A3 03 9D 02 C9 E8 E8 E8 AC
```

```
1C69:E8 E8 8E Ø1 C9 AC 8Ø C1 F5
1C71:C8 8C Ø4 C5 8C Ø3 C5 2Ø 2A
1C79:BD 1Ø 2Ø E3 1D C9 Ø9 FØ E9
1C81:Ø8 A9 Ø1 8D ØØ C9 8D 85 E8
1C89:C1 4C 3F 1E AD Ø1 C9 38 BC
1C91:E9 Ø7 AA 48 BD Ø2 C9 AA 8E
1C99:AØ ØØ BD BF 1D FØ Ø8 99 2C
1CA1:3C Ø3 E8 C8 4C 9B 1C 68 D3
1CA9:AA E8 E8 8E 8Ø C1 CA BD D5
1CB1:Ø2 C9 FØ 2A 84 Ø2 A6 Ø2 99
1CB9:A9 27 9D 3C Ø3 E8 2Ø F5 F9
1CC1:ØC B1 FB 9D 3C Ø3 E8 A9 2F
1CC9:27 9D 3C Ø3 A9 29 E8 9D 16
1CD1:3C Ø3 E8 8E Ø4 C5 A2 ØØ 6B
1CD9:8E Ø3 C5 4C Ø5 1D 84 Ø2 3F
1CE1:2Ø F5 ØC 98 18 65 FB A4 A5
1CE9:FC 2Ø A2 BB 2Ø DD BD AØ 4D
1CF1:ØØ A6 Ø2 B9 ØØ Ø1 FØ Ø8 9D
1CF9:9D 3C Ø3 E8 C8 4C F4 1C 7C
1DØ1:CA 4C CD 1C A9 Ø5 8D 41 EC
1DØ9:C5 4C 1B 2C A9 ØØ 8D 41 Ø9
1D11:C5 AD Ø1 C9 38 E9 Ø7 AA 78
1D19:BD Ø2 C9 8D F8 C9 E8 BD 43
1D21:Ø2 C9 FØ 24 E8 2Ø F5 ØC EE
1D29:B1 FB DD Ø2 C9 FØ 42 AD 5B
1D31:F8 C9 FØ ØA B1 FB DD Ø2 54
1D39:C9 BØ 36 4C 9A 1D B1 FB B8
1D41:DD Ø2 C9 9Ø 2C 4C 9A 1D 12
1D49:E8 2Ø F5 ØC 98 18 65 FB 6B
1D51:A4 FC 2Ø A2 BB AØ C9 A9 E8
1D59:Ø2 86 Ø2 18 65 Ø2 2Ø 5B C6
1D61:BC AE F8 C9 FØ Ø7 C9 FF 98
1D69:FØ 2F 4C 72 1D C9 Ø1 FØ 9B
1D71:28 EA EA EA A9 44 2Ø ØA 2F
1D79:12 BØ ØD 2Ø E3 1D C9 ØB BE
1D81:FØ ØB EE Ø3 C5 4C 75 1D 6C
1D89:A9 Ø1 4C 71 ØE EE Ø3 C5 71
1D91:AD Ø3 C5 8D Ø4 C5 4C 78 3D
1D99:1C EA EA EA AD Ø1 C9 38 E5
1DA1:E9 ØA 8D Ø1 C9 A9 ØØ 8D 97
1DA9:17 C5 2Ø F6 ØE 9Ø Ø3 4C 59
1DB1:71 ØE AD 15 C5 38 E9 Ø3 15
1DB9:8D 15 C5 4C 1F 1E 53 55 EA
```

```
1DC1:43 43 28 ØØ 5Ø 52 45 44 ØE
1DC9:28 ØØ 9E 16 58 32 2Ø 4A 63
1DD1:53 52 2Ø 49 4E 43 59 3A 4F
1DD9:4C 44 41 2Ø 28 24 46 4C 2Ø
1DE1:ED 1D A9 D5 A2 2E AØ ØØ FB
1DE9:2Ø 5Ø 11 6Ø A9 ØØ 8D 15 ED
1DF1:C5 8D Ø1 C5 8D ØØ C5 8D 74
1DF9:58 C7 8D 23 C3 AA 9D Ø2 3C
1EØ1:C9 E8 E4 64 BØ Ø4 E8 4C F2
1EØ9:FF 1D 8D 41 C5 8D Ø6 C9 8C
1E11:8D 3Ø C5 8D 31 C5 8D ØØ 6D
1E19:C9 A9 Ø9 4C 5B 1E 2Ø BD DB
1E21:1Ø 9Ø Ø3 4C 71 ØE AD ØØ CD
1E29:C5 FØ Ø8 A9 ØØ 8D ØØ C5 1C
1E31:4C 19 2A AD ØØ C9 FØ Ø6 Ø9
1E39:CE ØØ C9 4C 19 2A AD Ø1 A8
1E41:C5 C9 Ø1 DØ Ø3 4C 4B 2B ØB
1E49:2Ø E1 FF DØ ØA A9 9F AØ F1
1E51:2F 2Ø 1E AB 4C B3 ØE 2Ø 19
1E59:E3 1D C9 ØA BØ 12 ØA AA 35
1E61:BD 75 2F 8D C9 Ø2 E8 BD 7E
1E69:75 2F 8D CA Ø2 6C C9 Ø2 E1
1E71:AD Ø3 C5 8D Ø4 C5 A9 28 89
1E79:2Ø 27 12 9Ø ØC A9 3B 2Ø 78
1E81:27 12 9Ø Ø5 A9 ØD 2Ø 27 21
1E89:12 2Ø 3D 1F BØ Ø3 4C BD 58
1E91:1E AE Ø3 C5 8E Ø4 C5 A9 FE
1E99:28 2Ø 27 12 A9 3B A2 2F A6
1EA1:AØ ØØ 2Ø 5Ø 11 9Ø Ø3 4C 54
1EA9:71 2B ØA A8 B9 8D 2F 8D 25
1EB1:C9 Ø2 C8 B9 8D 2F 8D CA 17
1EB9:Ø2 6C C9 Ø2 EE 58 C7 EE C2
1EC1:Ø6 C9 2Ø 72 26 CE Ø6 C9 EØ
1EC9:AD 19 C5 8D 22 C5 FØ 4F ØE
1ED1:AE Ø4 C5 E8 8E Ø3 C5 8E 48
1ED9:Ø4 C5 2Ø 7B 12 BØ 4Ø A9 C2
1EE1:ØØ 8D Ø5 C5 A9 2C 2Ø 27 E3
1EE9:12 9Ø ØF EE Ø5 C5 A9 29 DF
1EF1:2Ø 27 12 9Ø Ø5 A9 Ø1 4C 7Ø
1EF9:71 ØE A9 Ø3 8D 41 C5 AE 83
1FØ1:Ø3 C5 4C 1B 2C EE Ø6 C9 6Ø
1FØ9:2Ø F5 ØC 2Ø 13 Ø8 AE ØØ 6E
1F11:CB 2Ø FD 26 CE Ø6 C9 AD 2F
```

```
1F19:Ø5 C5 DØ Ø3 4C C6 1E 2Ø 6F
1F21:AB 2F AE 5D C1 BD 5Ø CA 1D
1F29:8D AØ C5 E8 BD 5Ø CA 8D EF
1F31:A2 C5 E8 BD 5Ø CA 8D A1 95
1F39:C5 4C 1F 1E AØ ØØ B9 5Ø FB
1F41:CA C9 2A FØ 4A 8D C9 Ø2 C9
1F49:C8 AE Ø3 C5 A9 Ø8 85 FE CB
1F51:BD 3C Ø3 D9 5Ø CA DØ 13 DD
1F59:E8 C8 EC Ø4 C5 FØ 1C C6 ØD
1F61:FE FØ 18 B9 5Ø CA FØ Ø3 8C
1F69:4C 51 1F B9 5Ø CA FØ Ø4 35
1F71:C8 4C 6C 1F C8 C8 C8 C8 6A
1F79:4C 3F 1F B9 5Ø CA DØ EB 68
1F81:C8 8C 5D C1 AD C9 Ø2 8D 35
1F89:Ø5 C9 8D 23 C3 18 6Ø 38 18
1F91:6Ø A9 Ø1 2C A9 ØØ 8D 16 CB
1F99:C5 A9 Ø1 8D AØ Ø3 2Ø C3 33
1FA1:FF A8 A2 Ø3 2Ø BA FF 2Ø 9A
1FA9:CØ FF A9 ØØ 8D Ø3 C5 A9 2B
1FB1:28 2Ø ØA 12 9Ø Ø3 4C DØ 68
1FB9:21 A9 ØØ 8D Ø5 C5 EE Ø3 EB
1FC1:C5 2Ø 7B 12 AD Ø3 C5 8D ØE
1FC9:Ø4 C5 A9 2C 2Ø 13 22 9Ø 95
1FD1:ØF EE Ø5 C5 A9 29 2Ø 13 95
1FD9:22 9Ø Ø5 A9 Ø1 4C 71 ØE B2
1FE1:2Ø E3 1D C9 ØD FØ 1D C9 99
1FE9:ØE DØ 1C A9 Ø1 2Ø C3 FF 91
1FF1:A8 A2 Ø4 8D AØ Ø3 2Ø BA 92
1FF9:FF A9 ØØ 2Ø BD FF 2Ø CØ 93
2ØØ1:FF 4C 8C 22 4C E5 21 2Ø 64
2ØØ9:72 26 AD 19 C5 C9 Ø6 DØ 85
2Ø11:11 B1 FB C9 Ø4 FØ Ø5 A9 F9
2Ø19:Ø5 4C 71 ØE 8D AØ Ø3 4C 3F
2Ø21:8C 22 AE AØ Ø3 2Ø C9 FF 3C
2Ø29:A9 ØØ 8D Ø6 C5 8D Ø7 C5 88
2Ø31:AC Ø4 C5 8C Ø8 C5 AE Ø3 Ø2
2Ø39:C5 8E Ø4 C5 A9 3A 2Ø 13 66
2Ø41:22 9Ø Ø3 4C EB 2Ø AC Ø4 19
2Ø49:C5 CC Ø8 C5 9Ø Ø3 4C EB 12
2Ø51:2Ø AD Ø3 C5 8D Ø9 C5 EE D4
2Ø59:Ø4 C5 AC Ø4 C5 8C Ø3 C5 ØF
2Ø61:A9 ØØ 8D ØA C5 A9 3A 2Ø 32
2Ø69:13 22 BØ Ø8 AC Ø4 C5 CC 2Ø
```

```
2071:08 C5 90 0B A9 01 8D 0A 60
2079:C5 AD 08 C5 8D 04 C5 20 8D
2081:00 30 AD C5 02 C9 01 D0 E9
2089:4C AD 13 03 D0 49 A9 F2 DF
2091:A0 03 20 A2 BB 20 BF B1 A0
2099:A6 65 8E 06 C5 AD 0A C5 77
20A1:D0 38 EE 04 C5 AC 04 C5 25
20A9:8C 03 C5 AD 08 C5 8D 04 FA
20B1:C5 20 00 30 AD C5 02 C9 32
20B9:01 D0 1A AD 13 03 D0 17 2A
20C1:A9 F2 A0 03 20 A2 BB 20 FA
20C9:BF B1 A6 65 8E 07 C5 4C E9
20D1:DB 20 A9 16 2C A9 01 4C F4
20D9:71 0E AE 09 C5 8E 03 C5 F0
20E1:8E 04 C5 A9 3A 20 13 22 58
20E9:90 06 AD 08 C5 8D 04 C5 5C
20F1:20 00 30 AD 13 03 D0 DF 49
20F9:AD C5 02 C9 04 D0 03 4C 15
2101:6A 21 C9 03 D0 28 A2 00 96
2109:AD A2 03 C9 20 F0 11 BD 6C
2111:85 22 9D A2 03 E8 E0 07 01
2119:F0 03 4C 10 21 4C 6A 21 4F
2121:BD 7D 22 9D A2 03 E8 E0 93
2129:08 F0 F2 4C 21 21 C9 01 F0
2131:F0 03 4C 95 22 A0 00 B9 DC
2139:A2 03 F0 04 C8 4C 38 21 F4
2141:CC 06 C5 B0 13 AD 06 C5 50
2149:8C 06 C5 38 ED 06 C5 AA 4D
2151:A9 20 20 D2 FF CA D0 F8 67
2159:A2 00 BD A2 03 F0 07 20 D8
2161:D2 FF E8 4C 5B 21 4C C3 AA
2169:21 AD 07 C5 F0 05 A9 01 D4
2171:4C 71 0E A2 00 BD A2 03 61
2179:F0 04 E8 4C 76 21 CA CA AF
2181:8E 0B C5 AC 06 C5 CC 0B 3D
2189:C5 F0 17 CC 0B C5 90 12 3D
2191:AC 0B C5 A9 20 C8 20 D2 77
2199:FF C8 CC 06 C5 B0 03 4C 4B
21A1:97 21 A2 01 BD A2 03 C9 A4
21A9:27 F0 0C 20 D2 FF EC 06 B5
21B1:C5 F0 0F E8 4C A5 21 E8 A7
21B9:BD A2 03 C9 27 D0 03 4C 4F
21C1:AC 21 AD 0C C5 C9 01 D0 41
```

```
21C9:Ø3 4C CB 22 4C E5 21 AD 26
21D1:16 C5 FØ Ø5 A9 ØD 2Ø D2 93
21D9:FF A9 Ø1 2Ø C3 FF 2Ø CC D3
21E1:FF 4C 1F 1E AD Ø5 C5 FØ FA
21E9:Ø3 4C DØ 21 AC Ø8 C5 C8 C6
21F1:8C Ø3 C5 8C Ø4 C5 A9 2C 73
21F9:2Ø 13 22 BØ Ø3 4C 29 2Ø 1C
22Ø1:EE Ø5 C5 A9 29 2Ø 13 22 63
22Ø9:BØ Ø3 4C 29 2Ø A9 Ø1 4C 78
2211:71 ØE 8D 25 C3 AC Ø4 C5 34
2219:A9 ØØ 8D ØD C5 8D ØE C5 FA
2221:8D ØF C5 BE 3C Ø3 EC 25 81
2229:C3 FØ 3B EØ ØD FØ 4B EØ A4
2231:27 FØ 21 AD ØF C5 DØ 2A 9F
2239:EØ 28 DØ Ø3 EE ØD C5 EØ 5A
2241:29 DØ Ø3 CE ØD C5 EØ 5B 38
2249:DØ Ø3 EE ØE C5 EØ 5D DØ B2
2251:11 CE ØE C5 AD ØF C5 FØ 16
2259:Ø6 CE ØF C5 4C 63 22 EE B5
2261:ØF C5 C8 4C 24 22 AD ØD 8E
2269:C5 18 6D ØE C5 6D ØF C5 EC
2271:FØ Ø3 4C 3Ø 22 18 8C Ø4 ØA
2279:C5 6Ø 38 6Ø 27 46 41 4C E6
2281:53 45 27 ØØ 27 54 52 55 2A
2289:45 27 ØØ AE AØ Ø3 2Ø C9 4Ø
2291:FF 4C E5 21 AD Ø7 C5 FØ BD
2299:6B A9 Ø1 8D ØC C5 A2 ØØ B3
22A1:BD A2 Ø3 9D ØØ CD FØ Ø4 C4
22A9:E8 4C A1 22 A9 F2 AØ Ø3 29
22B1:2Ø A2 BB 2Ø CC BC 2Ø DD 9F
22B9:BD A2 ØØ BD ØØ Ø1 9D A2 43
22C1:Ø3 FØ Ø4 E8 4C BC 22 4C B8
22C9:36 21 CE ØC C5 A9 2E 2Ø 5D
22D1:D2 FF A2 ØØ BD ØØ CD C9 27
22D9:2E FØ Ø4 E8 4C D5 22 E8 67
22E1:EE Ø7 C5 BD ØØ CD FØ ØC 19
22E9:CE Ø7 C5 FØ 14 2Ø D2 FF E5
22F1:E8 4C E4 22 CE Ø7 C5 FØ 8B
22F9:Ø8 A9 3Ø 2Ø D2 FF 4C F5 D9
2301:22 4C E5 21 A2 ØØ 8E C9 36
2309:Ø2 BD A2 Ø3 9D ØØ CD FØ BD
2311:Ø4 E8 4C ØA 23 A9 27 8D 59
2319:A2 Ø3 8D B2 Ø3 AD ØØ CD EA
```

```
2321:8D A3 Ø3 A9 ØØ 8D B3 Ø3 B2
2329:AD Ø1 CD C9 2E DØ Ø3 4C E3
2331:D9 23 8D A4 Ø3 A9 2E 8D D1
2339:A5 Ø3 A9 2B 8D AF Ø3 A2 CE
2341:Ø2 AØ Ø4 BD ØØ CD FØ 3E 64
2349:C9 45 FØ ØC C9 2E FØ 4B D8
2351:99 A2 Ø3 E8 C8 4C 44 23 1F
2359:CØ ØC FØ Ø9 A9 3Ø 99 A2 95
2361:Ø3 C8 4C 59 23 BD ØØ CD 58
2369:99 A2 Ø3 C8 E8 BD ØØ CD 1E
2371:99 A2 Ø3 E8 C8 BD ØØ CD 27
2379:99 A2 Ø3 E8 C8 BD ØØ CD 2F
2381:99 A2 Ø3 4C F9 23 AD C9 E3
2389:Ø2 DØ Ø3 8E C9 Ø2 A9 3Ø 28
2391:CØ ØC FØ ØE 99 A2 Ø3 C8 6Ø
2399:4C 87 23 8E C9 Ø2 E8 4C A9
23A1:44 23 A9 45 99 A2 Ø3 C8 82
23A9:C8 AD C9 Ø2 38 E9 Ø2 A8 2F
23B1:2Ø A2 B3 2Ø DD BD AD Ø2 6C
23B9:Ø1 DØ ØE A9 3Ø 8D BØ Ø3 2D
23C1:AD Ø1 Ø1 8D B1 Ø3 4C F9 44
23C9:23 AD Ø1 Ø1 8D BØ Ø3 AD 2Ø
23D1:Ø2 Ø1 8D B1 Ø3 4C F9 23 86
23D9:A9 2D 8D AF Ø3 A2 Ø1 E8 7A
23E1:BD ØØ CD C9 3Ø FØ F8 8D 22
23E9:A4 Ø3 A9 2E 8D A5 Ø3 AØ Ø5
23F1:Ø4 E8 8E C9 Ø2 4C 44 23 CF
23F9:A9 12 8D Ø6 C5 4C A3 21 73
2401:A9 Ø1 2C A9 ØØ 8D 16 C5 A6
2409:A9 ØØ 8D Ø5 C5 8D AØ Ø3 DØ
2411:A9 28 2Ø ØA 12 9Ø Ø3 4C Ø2
2419:E8 25 EE Ø3 C5 2Ø 7B 12 E4
2421:AE Ø3 C5 8E Ø4 C5 A9 2C D9
2429:2Ø 27 12 9Ø ØF EE Ø5 C5 9A
2431:A9 29 2Ø 27 12 9Ø Ø5 A9 95
2439:Ø1 4C 71 ØE 2Ø E3 1D C9 B8
2441:ØC DØ Ø3 4C AA 25 2Ø 72 85
2449:26 2Ø E7 ØC AD 19 C5 8D 55
2451:22 C5 DØ Ø5 A9 Ø2 4C 71 E5
2459:ØE C9 Ø6 DØ 19 B1 FB C9 3A
2461:Ø3 FØ Ø5 A9 Ø5 4C 71 ØE EC
2469:8D AØ Ø3 A9 ØØ 8D 31 C5 F9
2471:8D 3Ø C5 4C AA 25 85 Ø2 Ø1
```

```
2479:AD 3Ø C5 DØ Ø3 4C 13 25 FE
2481:2Ø 7Ø 25 2Ø 15 26 A5 Ø2 2B
2489:C9 Ø3 9Ø 11 C9 Ø7 FØ ØD F3
2491:C9 Ø4 FØ 42 C9 Ø8 FØ 3E 9Ø
2499:A9 16 4C 71 ØE 2Ø 7B 12 D6
24A1:9Ø Ø6 AD 8D 25 4C 13 25 E7
24A9:AD Ø3 C5 8D Ø4 C5 A9 2Ø C5
24B1:2Ø 27 12 9Ø ØD A9 ØD 2Ø 68
24B9:27 12 2Ø 8F 2E A9 ØØ 8D BC
24C1:3Ø C5 A9 Ø1 8D 41 C5 4C 22
24C9:1B 2C EE Ø4 C5 AC Ø4 C5 77
24D1:8C Ø3 C5 4C Ø5 25 A9 Ø2 BØ
24D9:8D 41 C5 A9 Ø4 8D C5 Ø2 7Ø
24E1:A9 ØØ 8D A5 Ø3 2Ø F5 ØC 9B
24E9:AE Ø3 C5 BD 3C Ø3 E8 8D 2C
24F1:A3 Ø3 EE Ø3 C5 BD 3C Ø3 7B
24F9:C9 ØD DØ Ø5 A9 ØØ 8D 3Ø 6D
25Ø1:C5 4C 29 2C 2Ø F8 25 2Ø 78
25Ø9:8D 25 A9 ØØ 8D 41 C5 4C E1
2511:AA 25 AD AØ Ø3 DØ 1C 2Ø 6D
2519:6Ø A5 A2 ØØ BD ØØ Ø2 FØ 34
2521:Ø7 9D 8Ø C9 E8 4C 1D 25 DA
2529:A9 ØD 9D 8Ø C9 EE 3Ø C5 77
2531:4C 62 25 AE AØ Ø3 AD 31 67
2539:C5 FØ Ø5 A9 Ø5 4C 71 ØE 28
2541:2Ø C6 FF A2 ØØ 2Ø E4 FF C1
2549:9D 8Ø C9 A8 2Ø B7 FF 29 4F
2551:4Ø DØ Ø8 CØ ØD FØ Ø7 E8 2Ø
2559:4C 46 25 EE 31 C5 2Ø CC 9C
2561:FF A2 ØØ 8E 54 C7 8E 5A 76
2569:C7 EE 3Ø C5 4C 81 24 A2 Ø9
2571:ØØ BD 3C Ø3 9D 1Ø C9 C9 6D
2579:ØD FØ Ø4 E8 4C 72 25 AD B9
2581:Ø3 C5 8D 56 C7 AD Ø4 C5 98
2589:8D 57 C7 6Ø A2 ØØ BD 1Ø 1Ø
2591:C9 9D 3C Ø3 C9 ØD FØ Ø4 48
2599:E8 4C 8F 25 AD 56 C7 8D 93
25A1:Ø3 C5 AD 57 C7 8D Ø4 C5 4C
25A9:6Ø AD Ø5 C5 DØ 39 AC Ø4 55
25B1:C5 C8 8C Ø3 C5 8C Ø4 C5 Ø1
25B9:2Ø 7B 12 A9 2C 2Ø 27 12 12
25C1:9Ø ØF EE Ø5 C5 A9 29 2Ø 8D
25C9:27 12 9Ø Ø5 A9 Ø1 4C 71 E9
```

```
25D1:0E 20 72 26 20 E7 0C AD 42
25D9:19 C5 8D 22 C5 F0 03 4C 3A
25E1:77 24 A9 02 4C 71 0E AD 38
25E9:16 C5 F0 05 A9 00 8D 30 B7
25F1:C5 20 CC FF 4C 1F 1E A2 7E
25F9:00 BD 3C 03 9D 80 C9 C9 B7
2601:0D F0 04 E8 4C FA 25 AD 65
2609:03 C5 8D 54 C7 AD 04 C5 02
2611:8D 5A C7 60 A2 00 BD 80 CA
2619:C9 9D 3C 03 C9 0D F0 04 D1
2621:E8 4C 17 26 AD 54 C7 8D 16
2629:03 C5 AD 5A C7 8D 04 C5 06
2631:60 A9 00 8D 04 C5 A9 28 A3
2639:20 27 12 B0 2C AC 04 C5 8E
2641:C8 8C 03 C5 20 7B 12 A9 8E
2649:29 20 27 12 90 05 A9 01 25
2651:4C 71 0E 20 E3 1D C9 0D 18
2659:F0 0F 20 72 26 AD 19 C5 EC
2661:C9 06 F0 0A A9 05 4C 71 3E
2669:0E A9 93 20 D2 FF 4C 1F E9
2671:1E AD 0C C3 85 FB AC 0E 79
2679:C3 AD 0D C3 D0 03 4C 4B 67
2681:27 85 FC AD 23 C3 8D 1C 9C
2689:C5 A9 00 8D A1 03 A9 08 70
2691:8D 1A C5 8D 1B C5 AE 03 0D
2699:C5 B1 FB C9 2A D0 03 4C 38
26A1:3B 27 CD 23 C3 D0 43 AD D6
26A9:06 C9 F0 1B 20 0D 08 B1 32
26B1:FB C9 28 D0 F7 20 0D 08 E2
26B9:B1 FB 8D 19 C5 8D 00 CB 51
26C1:20 0D 08 20 E7 0C 60 20 B4
26C9:0D 08 BD 3C 03 C9 29 F0 9C
26D1:42 C9 20 F0 3E D1 FB D0 C6
26D9:11 20 0D 08 E8 EC 04 C5 A1
26E1:F0 31 CE 1A C5 F0 2C 4C 05
26E9:CB 26 B1 FB C9 28 F0 06 72
26F1:20 0D 08 4C EB 26 20 0D 9C
26F9:08 B1 FB AA BD C8 2E 8D DB
2701:1A C5 20 0D 08 CE 1A C5 18
2709:D0 F8 AD 1B C5 8D 1A C5 C3
2711:4C 97 26 B1 FB C9 28 F0 93
2719:03 4C EB 26 AD 1C C5 8D D2
2721:23 C3 B1 FB C9 28 F0 06 BE
```

```
2729:20 0D 08 4C 23 27 20 0D 93
2731:08 B1 FB 8D 19 C5 20 0D 75
2739:08 60 AD 23 C3 D0 11 AD BC
2741:1C C5 8D 23 C3 AD A1 03 0E
2749:D0 00 A9 00 8D 19 C5 60 F1
2751:A9 00 8D 23 C3 AD 0C C3 09
2759:85 FB AD 0D C3 85 FC AC CA
2761:0E C3 AD 1B C5 8D 1A C5 6D
2769:4C 97 26 A9 00 8D 03 C5 25
2771:A9 20 20 0A 12 20 7B 12 5B
2779:A9 20 20 0A 12 B0 05 20 C6
2781:7B 12 90 03 4C 1F 1E AD 1D
2789:03 C5 8D 04 C5 4C 27 1E 88
2791:20 AB 2F A9 20 20 0A 12 03
2799:B0 0E 20 7B 12 B0 09 AD 92
27A1:03 C5 8D 04 C5 4C 3F 1E D0
27A9:4C 1F 1E A9 20 20 0A 12 EB
27B1:20 7B 12 90 08 20 BD 10 86
27B9:90 03 4C 71 0E A9 3B 20 5F
27C1:27 12 90 05 A9 0D 20 27 73
27C9:12 20 FB 2D AD 13 03 F0 2C
27D1:03 4C 71 0E E0 20 F0 13 40
27D9:AD 15 C5 38 E9 03 8D 15 0C
27E1:C5 B0 05 A9 09 4C 71 0E E4
27E9:4C 1F 1E AE 15 C5 BD 00 10
27F1:C6 8D A0 C5 E8 BD 00 C6 7C
27F9:8D A2 C5 E8 BD 00 C6 8D 08
2801:A1 C5 AD 15 C5 38 E9 03 80
2809:8D 15 C5 4C 1F 1E A9 00 A7
2811:8D 17 C5 A9 20 20 0A 12 E8
2819:B0 05 20 7B 12 90 06 20 BD
2821:F6 0E 4C 1F 1E A9 20 20 E3
2829:27 12 20 FD 0E 4C 1F 1E 73
2831:A9 20 20 0A 12 B0 1A 20 AA
2839:7B 12 B0 15 AE 03 C5 8E CE
2841:C9 02 BD 3C 03 C9 54 F0 4B
2849:0D C9 0D F0 04 E8 4C 43 E2
2851:28 A9 01 4C 71 0E 8E 03 E8
2859:C5 CA 8E 04 C5 20 E3 1D E4
2861:AD 02 C5 C9 0A F0 05 E8 65
2869:E8 4C 43 28 AE C9 02 8E 5B
2871:03 C5 20 FB 2D AD 13 03 C1
2879:F0 03 4C 71 0E E0 20 F0 C8
2881:36 AD 04 C5 18 69 05 8D 33
```

```
2889:Ø3 C5 8D Ø4 C5 2Ø 7B 12 76
2891:BØ Ø3 4C 99 28 2Ø BD 1Ø 6B
2899:2Ø E3 1D C9 Ø9 FØ Ø3 4C 91
28A1:34 1E AD 15 C5 18 69 Ø3 FE
28A9:8D 15 C5 18 69 Ø2 AA A9 92
28B1:FF 9D ØØ C6 4C 34 1E A9 EE
28B9:ØØ 8D 17 C5 AD Ø4 C5 18 CD
28C1:69 Ø5 8D Ø3 C5 8D Ø4 C5 1C
28C9:2Ø 7B 12 BØ Ø6 2Ø FD ØE 11
28D1:4C D7 28 2Ø F6 ØE 2Ø BD 33
28D9:1Ø 9Ø Ø3 4C 71 ØE 2Ø E3 63
28E1:1D C9 Ø5 FØ Ø3 4C 34 1E B2
28E9:A9 2Ø 2Ø ØA 12 2Ø 7B 12 D5
28F1:AD Ø3 C5 8D Ø4 C5 BØ Ø3 Ø7
28F9:4C Ø4 29 2Ø BD 1Ø 9Ø Ø3 EA
2901:4C 71 ØE 2Ø E3 1D C9 Ø9 C9
2909:FØ Ø3 4C 34 1E 4C A3 28 F2
2911:A9 2Ø 2Ø ØA 12 BØ 1A 2Ø 8C
2919:7B 12 BØ 15 AE Ø3 C5 8E BØ
2921:C9 Ø2 BD 3C Ø3 C9 44 FØ ØD
2929:ØD C9 ØD FØ Ø4 E8 4C 23 A4
2931:29 A9 Ø1 4C 71 ØE 8E Ø3 4B
2939:C5 CA 8E Ø4 C5 2Ø E3 1D C6
2941:C9 ØB FØ Ø5 E8 E8 4C 23 5Ø
2949:29 AE C9 Ø2 8E Ø3 C5 2Ø 61
2951:FB 2D AD 13 Ø3 FØ Ø3 4C Ø2
2959:71 ØE EØ 2Ø FØ 2E 2Ø AB 32
2961:2F AD Ø4 C5 18 69 Ø3 8D 8D
2969:Ø3 C5 8D Ø4 C5 2Ø 7B 12 58
2971:BØ Ø3 4C 7E 29 2Ø BD 1Ø A3
2979:9Ø Ø3 4C 71 ØE 2Ø E3 1D 4B
2981:C9 Ø9 FØ Ø5 A9 Ø1 8D ØØ D5
2989:C5 4C 34 1E AD ØØ C5 FØ 24
2991:Ø6 2Ø BD 1Ø 4C B7 29 A9 E4
2999:ØØ 8D 17 C5 AD Ø4 C5 18 AF
29A1:69 Ø3 8D Ø3 C5 8D Ø4 C5 7D
29A9:2Ø 7B 12 BØ Ø6 2Ø FD ØE F2
29B1:4C B7 29 2Ø F6 ØE A9 ØØ 82
29B9:8D ØØ C5 4C 1F 1E AD 15 32
29C1:C5 FØ Ø5 A9 Ø6 4C 71 ØE CØ
29C9:2Ø BD 1Ø 9Ø F6 A9 76 AØ 92
29D1:A3 2Ø 1E AB 4C B3 ØE AD 77
29D9:15 C5 18 69 Ø2 AA BD ØØ F7
```

```
29E1:C6 C9 FF D0 0C AD 15 C5 1E
29E9:38 E9 03 8D 15 C5 4C 1F 83
29F1:1E C9 FE D0 11 20 BD 10 47
29F9:B0 5A AD 15 C5 38 E9 03 28
2A01:8D 15 C5 4C 4B 2B AD A0 E1
2A09:C5 8D 14 C3 AD A1 C5 8D 6F
2A11:15 C3 AD A2 C5 8D 16 C3 15
2A19:AE 15 C5 BD 00 C6 8D A0 75
2A21:C5 E8 BD 00 C6 8D A2 C5 C1
2A29:E8 BD 00 C6 8D A1 C5 AD F9
2A31:15 C5 38 E9 03 B0 05 A9 B5
2A39:06 4C 71 0E 8D 15 C5 AD AC
2A41:58 C7 F0 0B A9 00 8D 58 43
2A49:C7 8D 23 C3 20 BD 10 20 BD
2A51:BD 10 90 03 4C 71 0E 4C 5B
2A59:34 1E A9 20 20 0A 12 8E 62
2A61:C9 02 A9 4F 20 0A 12 B0 43
2A69:0D 20 E3 1D C9 10 F0 0B 16
2A71:EE 03 C5 4C 63 2A A9 01 93
2A79:4C 71 0E AC 03 C5 88 8C A9
2A81:04 C5 AE C9 02 8E 03 C5 D1
2A89:20 7B 12 20 00 30 AD 13 40
2A91:03 F0 03 4C 71 0E AD C5 AD
2A99:02 C9 03 D0 1A A2 00 A0 CA
2AA1:00 AD A2 03 C9 2D F0 02 CC
2AA9:A2 05 BD 66 2B 99 A2 03 B6
2AB1:F0 05 E8 C8 4C AB 2A A9 78
2AB9:00 8D 05 C5 20 BD 10 90 17
2AC1:05 A9 07 4C 71 0E A9 2C EB
2AC9:20 27 12 B0 3D AE 03 C5 B5
2AD1:AD 04 C5 8D 42 C5 20 AF A8
2AD9:2E A0 00 AD A2 03 C9 20 1D
2AE1:D0 01 C8 B9 A2 03 DD 3C AC
2AE9:03 D0 0A E8 C8 EC 04 C5 8B
2AF1:F0 25 4C E4 2A AD 05 C5 B7
2AF9:D0 BD AC 42 C5 C8 8C 03 4D
2B01:C5 8C 04 C5 20 7B 12 4C 99
2B09:C7 2A EE 05 C5 A9 3A 20 65
2B11:27 12 90 B9 4C B8 2A A9 70
2B19:3A 20 0A 12 EE 03 C5 20 26
2B21:7B 12 B0 21 20 E3 1D C9 76
2B29:09 D0 14 AD 15 C5 18 69 EE
2B31:03 8D 15 C5 AA E8 E8 A9 DF
2B39:FE 9D 00 C6 4C 34 1E EE 41
```

```
2B41:Ø1 C5 4C 49 1E A9 Ø1 4C 8D
2B49:71 ØE A9 Ø1 8D 17 C5 2Ø 95
2B51:FD ØE 2C 2Ø F6 ØE 9Ø Ø5 C7
2B59:A9 Ø7 4C 71 ØE A9 ØØ 8D 8B
2B61:Ø1 C5 4C 1F 1E 54 52 55 61
2B69:45 ØØ 46 41 4C 53 45 ØØ 79
2B71:AØ ØØ 8C Ø4 C5 A9 3A 2Ø 53
2B79:27 12 9Ø Ø5 A9 Ø1 4C 71 A5
2B81:ØE AC Ø4 C5 C8 B9 3C Ø3 8F
2B89:C9 3D FØ Ø3 4C DØ 2F 2Ø 86
2B91:72 26 AD 19 C5 DØ Ø5 A9 17
2B99:Ø2 4C 71 ØE 8D 22 C5 2Ø B3
2BA1:E7 ØC C9 ØB DØ 41 AC Ø4 C1
2BA9:C5 C8 C8 8C Ø3 C5 A9 5B D4
2BB1:2Ø ØA 12 2Ø F5 ØC A9 27 39
2BB9:2Ø ØA 12 BØ 19 EE Ø3 C5 4Ø
2BC1:AE Ø3 C5 BD 3C Ø3 91 FB D1
2BC9:2Ø ØD Ø8 A9 27 2Ø ØA 12 EE
2BD1:EE Ø3 C5 4C B7 2B A9 5D F8
2BD9:2Ø ØA 12 9Ø Ø3 4C 7D 2B 7D
2BE1:A9 ØØ 91 FB 4C 31 2D AD 2E
2BE9:41 C5 C9 Ø4 DØ 11 AC Ø4 F3
2BF1:C5 C8 C8 8C Ø3 C5 AC F7 BF
2BF9:C9 8C Ø4 C5 4C 1B 2C AC Ø9
2CØ1:Ø4 C5 C8 C8 8C Ø3 C5 A9 18
2CØ9:ØD 2Ø 27 12 AC Ø4 C5 88 7F
2C11:B9 3C Ø3 C9 3B DØ Ø3 CE 44
2C19:Ø4 C5 2Ø 7B 12 2Ø ØØ 3Ø E1
2C21:AD 13 Ø3 FØ Ø3 4C 71 ØE BE
2C29:2Ø F5 ØC AD C5 Ø2 C9 Ø2 37
2C31:BØ 16 AD 22 C5 C9 Ø3 BØ 4B
2C39:Ø3 4C 9A 2C C9 Ø7 DØ Ø3 4B
2C41:4C B9 2C A9 Ø2 4C 71 ØE 8Ø
2C49:C9 Ø2 DØ ØF AD 22 C5 C9 5D
2C51:Ø2 DØ Ø3 4C 9A 2C A9 Ø2 DE
2C59:4C 71 ØE C9 Ø3 DØ ØF AD B9
2C61:22 C5 C9 Ø3 DØ Ø3 4C AC 7D
2C69:2C A9 Ø3 4C 71 ØE C9 Ø4 C2
2C71:DØ 22 AD A5 Ø3 FØ ØØ AD 54
2C79:22 C5 C9 Ø4 DØ Ø3 4C AF A8
2C81:2C C9 Ø8 DØ Ø3 4C 1Ø 2D Ø7
2C89:C9 ØC DØ Ø3 4C 58 2D A9 DB
2C91:Ø2 4C 71 ØE A9 Ø2 4C 71 6C
```

```
2C99:ØE A2 ØØ BD F2 Ø3 91 FB 4Ø
2CA1:E8 2Ø ØD Ø8 EØ Ø6 DØ F3 4D
2CA9:4C 31 2D A2 ØØ 2C A2 Ø1 3B
2CB1:BD A2 Ø3 91 FB 4C 31 2D AB
2CB9:A9 F2 AØ Ø3 2Ø A2 BB A2 8D
2CC1:Ø5 2Ø F5 ØC 2Ø ØD Ø8 CA 34
2CC9:DØ FA A5 FB 84 Ø2 18 65 7F
2CD1:Ø2 A4 FC 2Ø 5B BC C9 FF 57
2CD9:FØ 3Ø A2 ØA 2Ø F5 ØC 2Ø BC
2CE1:ØD Ø8 CA DØ FA A5 FB 84 14
2CE9:Ø2 18 65 Ø2 A4 FC 2Ø 5B CA
2CF1:BC C9 Ø1 FØ 15 A2 ØØ 2Ø 9D
2CF9:F5 ØC BD F2 Ø3 91 FB 2Ø AD
2DØ1:ØD Ø8 E8 EØ Ø5 DØ F3 4C AE
2DØ9:31 2D A9 Ø4 4C 71 ØE AD AE
2D11:A5 Ø3 DØ F6 AD A3 Ø3 2Ø AA
2D19:ØD Ø8 D1 FB BØ Ø3 4C ØB 2B
2D21:2D 2Ø ØD Ø8 D1 FB 9Ø ØØ DB
2D29:2Ø 13 Ø8 2Ø 13 Ø8 91 FB 33
2D31:AD 41 C5 FØ 1F C9 Ø1 FØ 8D
2D39:ØF C9 Ø3 FØ ØE C9 Ø4 FØ 8D
2D41:ØD C9 Ø5 FØ ØC 4C Ø5 25 Ø5
2D49:4C CB 24 4C Ø6 1F 4C A1 EC
2D51:1B 4C ØD 1D 4C 1F 1E 2Ø FA
2D59:F5 ØC A2 ØØ BD A2 Ø3 91 16
2D61:FB FØ Ø7 E8 2Ø ØD Ø8 4C F6
2D69:5D 2D CA CA CA 2Ø ØD Ø8 BC
2D71:B1 FB C9 Ø1 DØ F7 2Ø 13 A6
2D79:Ø8 8A D1 FB FØ ØØ 4C 31 C5
2D81:2D A9 ØØ A8 A9 6Ø 85 FC 3E
2D89:A9 Ø8 8D 1A C5 AE Ø3 C5 C2
2D91:B1 FB C9 2A FØ 59 BD 3C 44
2D99:Ø3 C9 2Ø FØ 42 D1 FB DØ 1D
2DA1:ØC CE 1A C5 FØ 39 E8 2Ø B3
2DA9:ØD Ø8 4C 97 2D B1 FB FØ A8
2DB1:Ø6 2Ø ØD Ø8 4C AE 2D 2Ø DØ
2DB9:ØD Ø8 B1 FB C9 Ø5 9Ø ØD 23
2DC1:2Ø ØD Ø8 B1 FB DØ F9 2Ø C2
2DC9:ØD Ø8 4C 89 2D AA BD F6 55
2DD1:2D 8D 1A C5 2Ø ØD Ø8 CE D9
2DD9:1A C5 DØ F8 4C 89 2D B1 FØ
2DE1:FB DØ CA 2Ø ØD Ø8 B1 FB B1
2DE9:8D 19 C5 2Ø ØD Ø8 6Ø A9 FE
2DF1:ØØ 8D 19 C5 6Ø ØØ Ø6 Ø6 44
```

```
2DF9:Ø2 Ø4 AD Ø3 C5 8D 5B C7 1F
2EØ1:AD Ø4 C5 8D 5C C7 A9 49 65
2EØ9:2Ø ØA 12 BØ 66 2Ø E3 1D DD
2E11:C9 ØF FØ Ø6 EE Ø3 C5 4C EF
2E19:Ø7 2E AD Ø4 C5 8D 21 C5 E6
2E21:AD Ø3 C5 8D Ø4 C5 AD 5B 94
2E29:C7 8D Ø3 C5 2Ø ØØ 3Ø AD 98
2E31:C5 Ø2 C9 Ø4 DØ 4D AE A3 27
2E39:Ø3 8E C9 Ø2 AD Ø4 C5 18 35
2E41:69 Ø4 8D Ø3 C5 2Ø 7B 12 EC
2E49:2Ø 72 26 AD 19 C5 C9 ØB 7Ø
2E51:DØ 31 B1 FB FØ ØE CD C9 7D
2E59:Ø2 FØ Ø6 2Ø ØD Ø8 4C 53 2A
2E61:2E A9 2D 2C A9 2Ø AD 5B 2C
2E69:C7 8D Ø3 C5 AD 5C C7 8D C5
2E71:Ø4 C5 6Ø AD 5B C7 8D Ø3 4Ø
2E79:C5 2Ø ØØ 3Ø AD C5 Ø2 C9 16
2E81:Ø3 FØ Ø5 A9 ØE 8D 13 Ø3 A6
2E89:AE A2 Ø3 4C 67 2E AC Ø4 5C
2E91:C5 B9 3C Ø3 C9 3B DØ 15 E8
2E99:AC Ø4 C5 B9 3C Ø3 C9 2Ø 43
2EA1:DØ Ø8 88 CØ Ø1 FØ Ø3 4C A3
2EA9:9C 2E 8C Ø4 C5 6Ø AC Ø4 BE
2EB1:C5 88 B9 3C Ø3 C9 2Ø DØ 5E
2EB9:Ø9 88 4C B3 2E C8 8C Ø4 33
2EC1:C5 6Ø C8 8C Ø4 C5 6Ø ØØ F2
2EC9:Ø7 Ø7 Ø3 Ø3 ØB Ø4 11 Ø5 8B
2ED1:1D ØØ FB FC 52 45 5Ø 45 99
2ED9:41 54 ØØ 55 4E 54 49 4C E3
2EE1:ØØ 45 4E 44 ØØ 45 4E 44 93
2EE9:2E ØØ 49 46 ØØ 45 4C 53 EB
2EF1:45 ØØ 46 4F 52 ØØ 43 41 Ø9
2EF9:53 45 ØØ 57 48 49 4C 45 ØC
2FØ1:ØØ 42 45 47 49 4E ØØ 54 E4
2FØ9:48 45 4E ØØ 44 4F ØØ 49 4F
2F11:4E 5Ø 55 54 ØØ 4F 55 54 D6
2F19:5Ø 55 54 ØØ 5Ø 52 49 4E 2C
2F21:54 45 52 ØØ 49 4E ØØ 4F 18
2F29:46 ØØ 54 4F ØØ 44 4F 57 31
2F31:4E 54 4F ØØ 47 4F 54 4F 25
2F39:ØØ 2A 52 45 53 45 54 ØØ 19
2F41:52 45 57 52 49 54 45 ØØ 5Ø
2F49:52 45 41 44 ØØ 52 45 41 A3
2F51:44 4C 4E ØØ 57 52 49 54 99
```

```
2F59:45 ØØ 57 52 49 54 45 4C DC
2F61:4E ØØ 5Ø 41 47 45 ØØ 5Ø A4
2F69:41 43 4B ØØ 55 4E 5Ø 41 68
2F71:43 4B ØØ 2A 91 27 AC 27 9Ø
2F79:D8 29 BF 29 31 28 ØF 28 89
2F81:ØØ 1B 5B 2A 11 29 6C 27 E1
2F89:CA 2F CD 2F DØ 2F DØ 2F D9
2F91:Ø4 24 Ø1 24 95 1F 92 1F CA
2F99:32 26 CA 2F CD 2F ØD ØD 39
2FA1:1C 42 52 45 41 4B 9F ØD CØ
2FA9:ØD ØØ AD 15 C5 18 69 Ø3 F9
2FB1:8D 15 C5 AA AD 1Ø C5 9D 56
2FB9:ØØ C6 AD 11 C5 E8 9D ØØ 9D
2FC1:C6 E8 AD 12 C5 9D ØØ C6 FF
2FC9:6Ø 4C 1F 1E 4C 1F 1E A9 F5
2FD1:Ø1 4C 71 ØE 33 ØØ E4 2F 65
2FD9:52 26 4C 44 41 2Ø 23 22 AB
2FE1:3A 22 ØØ FØ 2F 5C 26 53 7F
2FE9:54 41 2Ø 37 31 34 ØØ FF 94
2FF1:2F 66 26 4A 53 52 2Ø 53 62
2FF9:54 49 4E 5Ø 5A ØØ 2A AE 79
3ØØ1:Ø3 C5 AC Ø4 C5 88 8C 37 CA
3ØØ9:Ø3 A9 Ø1 8D C5 Ø2 EE 37 99
3Ø11:Ø3 A9 28 8D A2 Ø3 A9 ØØ AF
3Ø19:8D 34 Ø3 8D 13 Ø3 8D F2 39
3Ø21:CC 8D FF CF AØ Ø1 8C EE 59
3Ø29:CC BD 3C Ø3 99 A2 Ø3 E8 5D
3Ø31:C8 EC 37 Ø3 FØ Ø3 4C 2A 9E
3Ø39:3Ø A9 29 99 A2 Ø3 8C 35 4A
3Ø41:Ø3 A9 ØØ 8D 37 Ø3 8D FA 42
3Ø49:Ø3 8D BF Ø2 A2 Ø1 C8 99 EA
3Ø51:A2 Ø3 AD 35 Ø3 C9 Ø2 BØ CØ
3Ø59:Ø6 A9 Ø1 8D 13 Ø3 6Ø A9 2F
3Ø61:ØØ 8D F2 Ø3 8D F3 Ø3 8D 83
3Ø69:F4 Ø3 8D F5 Ø3 8D F6 Ø3 55
3Ø71:8D F7 Ø3 BD A2 Ø3 C9 27 AE
3Ø79:DØ Ø6 2Ø 8E 3Ø 4C 85 3Ø 9E
3Ø81:C9 29 FØ 16 E8 EC 35 Ø3 F8
3Ø89:DØ E9 4C AØ 3Ø AD 37 Ø3 ØA
3Ø91:FØ Ø4 CE 37 Ø3 6Ø EE 37 67
3Ø99:Ø3 6Ø AD 37 Ø3 DØ E5 8E 72
3ØA1:39 Ø3 A9 ØØ 8D 37 Ø3 CA AE
3ØA9:EC 34 Ø3 FØ 19 BD A2 Ø3 Ø5
3ØB1:C9 27 DØ Ø6 2Ø 8E 3Ø 4C 23
```

```
30B9:A8 30 C9 28 F0 03 4C A8 0B
30C1:30 AD 37 03 D0 E1 A9 00 1E
30C9:8D 3A 03 8E FF CC 8E 38 51
30D1:03 8E C2 02 A9 00 8D 37 6F
30D9:03 E8 EC 39 03 F0 67 BD 8F
30E1:A2 03 C9 27 D0 06 20 8E 6D
30E9:30 4C DA 30 AC 37 03 F0 0D
30F1:03 4C DA 30 C9 2B F0 5C 7E
30F9:C9 2D F0 58 C9 3C D0 03 12
3101:4C C8 31 C9 3D D0 03 4C FD
3109:C8 31 C9 3E D0 03 4C C8 2D
3111:31 86 02 A4 02 C8 C8 20 1D
3119:1F 3C AD 36 03 D0 13 8E 43
3121:3B 03 8E C3 02 8A 18 69 C3
3129:02 8D 3B 03 BD A2 03 4C 52
3131:EB 31 C8 20 1F 3C AD 36 6C
3139:03 C9 01 D0 9C 8E 3B 03 55
3141:8E C3 02 4C 26 31 A9 54 7E
3149:8E 3B 03 8E C3 02 EE 3A 49
3151:03 4C EB 31 8E 3B 03 8E CE
3159:C3 02 CA BC A2 03 C0 20 06
3161:F0 F8 C0 2A F0 5B C0 44 EF
3169:F0 57 C0 52 F0 53 C0 2B D8
3171:D0 06 EE C3 02 4C C2 31 CF
3179:C0 2F F0 45 C0 28 F0 41 44
3181:C0 45 D0 0F CA BC A2 03 32
3189:C0 30 90 2C C0 3A B0 28 A5
3191:4C C2 31 86 02 A4 02 EC EC
3199:34 03 F0 25 CA EC 34 03 BC
31A1:90 1F BD A2 03 C9 20 D0 46
31A9:F3 E8 20 1F 3C AD 36 03 3E
31B1:C9 02 F0 0D C9 03 F0 09 AD
31B9:AE C3 02 BD A2 03 4C EB 26
31C1:31 AE C3 02 4C DA 30 8E BD
31C9:C3 02 E8 BC A2 03 CA C0 EE
31D1:3E F0 0E C0 3D F0 0A C0 DF
31D9:3C F0 06 8E 3B 03 4C EB AA
31E1:31 8E 3B 03 8C BF 02 EE 6E
31E9:3B 03 8D FC 03 A9 00 8D 78
31F1:FF 03 AE C2 02 E8 EC C3 68
31F9:02 D0 14 A9 00 8D C6 02 74
3201:8D C7 02 8D C8 02 8D C9 6A
3209:02 8D CA 02 8D CB 02 AE 99
3211:C2 02 A9 00 8D 37 03 E8 C4
```

```
3219:EC C3 Ø2 BØ 35 BD A2 Ø3 19
3221:C9 27 DØ Ø6 2Ø 8E 3Ø 4C 96
3229:18 32 AD 37 Ø3 DØ E8 BD 3A
3231:A2 Ø3 C9 2F FØ 3A C9 2A Ø2
3239:FØ 36 86 Ø2 A4 Ø2 C8 C8 1C
3241:C8 2Ø 1F 3C AD 36 Ø3 C9 CF
3249:Ø2 FØ 25 C9 Ø3 FØ 21 4C 96
3251:18 32 AD FF Ø3 DØ 14 A9 31
3259:Ø2 8D FF Ø3 A9 ØØ 8D C1 7C
3261:Ø2 A9 FB 8D FF CF 86 Ø2 D7
3269:4C 8A 32 A9 Ø2 8D FF Ø3 CØ
3271:86 Ø2 BD A2 Ø3 8D C4 Ø2 55
3279:AD FF Ø3 FØ Ø3 4C 46 34 2E
3281:EE FF Ø3 BD A2 Ø3 8D C1 97
3289:Ø2 AE C2 Ø2 E8 E4 Ø2 DØ C2
3291:ØD A9 3Ø 8D C9 Ø2 A9 ØØ 6F
3299:8D CA Ø2 4C CA 32 AØ ØØ DC
32A1:BD A2 Ø3 99 C9 Ø2 E8 C8 78
32A9:E4 Ø2 FØ Ø3 4C A1 32 A9 46
32B1:ØØ 99 C9 Ø2 AØ ØØ B9 C9 18
32B9:Ø2 C9 2Ø DØ Ø4 C8 4C B7 36
32C1:32 C9 3Ø 9Ø 6A C9 3A BØ 6Ø
32C9:66 A5 7A 8D 36 Ø3 A5 7B 77
32D1:8D CØ Ø2 A9 C9 AØ Ø2 85 62
32D9:7A 84 7B 2Ø 79 ØØ 2Ø F3 ØE
32E1:BC AD 36 Ø3 85 7A AD CØ 39
32E9:Ø2 85 7B AD EE CC C9 46 7F
32F1:DØ Ø3 4C A1 36 AD 37 Ø3 FC
32F9:C9 FD DØ Ø3 4C 4E 34 A2 B3
3301:8E AØ Ø3 2Ø D4 BB A5 Ø2 1C
3309:8D C2 Ø2 AD FF CF C9 FB DØ
3311:DØ Ø6 CE FF CF 4C 8B 35 37
3319:AD C1 Ø2 C9 44 FØ Ø7 C9 61
3321:4D FØ Ø3 4C 1Ø 32 EE C2 79
3329:Ø2 EE C2 Ø2 4C 1Ø 32 C9 95
3331:2B FØ 96 C9 2D FØ 92 C9 F4
3339:2E FØ 13 C9 3Ø FØ ØF C9 1F
3341:27 FØ 11 C9 54 FØ 15 C9 9Ø
3349:46 FØ 11 4C 7D 3A A9 Ø1 1F
3351:8D 13 Ø3 6Ø A9 Ø4 8D C5 E7
3359:Ø2 4C B5 33 8C 98 Ø3 A2 2D
3361:Ø5 B9 C9 Ø2 C9 45 FØ Ø7 5E
3369:C8 CA FØ Ø3 4C 62 33 AE 36
3371:98 Ø3 A9 C9 85 FB A9 Ø2 28
```

```
3379:85 FC 20 27 3C AD 36 03 60
3381:C9 04 F0 07 C9 05 F0 1A BA
3389:4C 7D 3A A9 03 8D C5 02 33
3391:A9 2D 8D C9 02 A9 31 8D 0D
3399:CA 02 A9 00 8D CB 02 4C 07
33A1:CA 32 A9 30 8D C9 02 A9 73
33A9:00 8D CA 02 A9 03 8D C5 27
33B1:02 4C CA 32 AD FA 03 D0 D8
33B9:17 A0 00 A2 00 B9 C9 02 7A
33C1:F0 08 9D 00 CF C8 E8 4C 16
33C9:BE 33 9D 00 CF 4C A9 36 49
33D1:C9 3C F0 0E C9 3E F0 0A 5E
33D9:C9 3D F0 06 A9 05 8D 13 82
33E1:03 60 A0 00 A2 00 B9 C9 48
33E9:02 F0 08 9D 50 CF E8 C8 C4
33F1:4C E7 33 9D 50 CF 4C 44 57
33F9:35 A9 03 8D F5 CC AD BF 9C
3401:02 C9 3E D0 11 AD FA 03 E9
3409:C9 3C D0 0A AD 36 03 C9 36
3411:3D F0 0B 4C 34 34 AD 36 7E
3419:03 CD FA 03 F0 15 AD BF FC
3421:02 CD 36 03 F0 0D A9 30 34
3429:8D C9 02 A9 00 8D CA 02 73
3431:4C CA 32 A9 2D 8D C9 02 88
3439:A9 31 8D CA 02 A9 00 8D 65
3441:CB 02 4C CA 32 A9 FD 8D 08
3449:37 03 4C 8A 32 AD C1 02 0E
3451:C9 2A F0 03 4C 9A 34 AD 5A
3459:C5 02 C9 02 90 00 A9 8E E4
3461:A0 03 20 28 BA A2 8E A0 7F
3469:03 20 D4 BB AD C5 02 C9 04
3471:02 90 00 AD FF 03 C9 02 7B
3479:D0 03 4C 8B 35 AD C4 02 39
3481:8D C1 02 C9 2A F0 0A C9 F0
3489:2F F0 06 E6 02 E6 02 E6 8B
3491:02 A5 02 8D C2 02 4C 10 44
3499:32 C9 2F D0 43 A9 02 8D D2
34A1:C5 02 A9 61 D0 06 A9 25 CF
34A9:8D 13 03 60 A2 98 A0 03 BF
34B1:20 D4 BB A9 8E A0 03 20 8E
34B9:0F BB AD 36 03 C9 FA F0 D7
34C1:5E 20 83 35 AD C5 02 C9 77
34C9:03 90 06 A9 0A 8D 13 03 E2
34D1:60 AD C0 02 C9 F3 F0 2A 38
```

```
34D9:A9 Ø2 8D C5 Ø2 4C 74 34 Ø4
34E1:C9 44 DØ 33 AD C5 Ø2 C9 DF
34E9:Ø1 FØ Ø6 A9 14 8D 13 Ø3 6A
34F1:6Ø 2Ø B6 39 AD 13 Ø3 FØ AD
34F9:Ø1 6Ø A9 F3 8D CØ Ø2 4C 2F
35Ø1:9E 34 A9 Ø1 8D C5 Ø2 2Ø B4
35Ø9:CC BC AD 36 Ø3 C9 FA FØ 48
3511:ØE 2Ø 83 35 4C 74 34 A9 94
3519:FA 8D 36 Ø3 4C 9E 34 2Ø CØ
3521:CC BC A9 98 AØ Ø3 2Ø 28 59
3529:BA A9 8E AØ Ø3 2Ø 8C BA A3
3531:A5 61 2Ø 53 B8 2Ø 58 BC B3
3539:2Ø 83 35 A9 Ø1 8D C5 Ø2 A1
3541:4C 74 34 AØ ØØ A9 3D 8D 2E
3549:36 Ø3 B9 5Ø CF FØ 3Ø B9 28
3551:ØØ CF FØ 2B D9 5Ø CF 9Ø CØ
3559:Ø8 FØ Ø2 BØ 13 C8 4C 4B EE
3561:35 A9 3D CD 36 Ø3 DØ F5 AA
3569:A9 3C 8D 36 Ø3 4C 5E 35 Ø8
3571:A9 3D CD 36 Ø3 DØ E6 A9 EF
3579:3E 8D 36 Ø3 4C 5E 35 4C EF
3581:FA 33 A2 F2 AØ Ø3 2Ø D4 DF
3589:BB 6Ø AD C5 Ø2 C9 Ø4 BØ EB
3591:Ø3 4C A5 35 AD F5 CC C9 41
3599:Ø3 DØ Ø3 8D C5 Ø2 2Ø 83 EC
35A1:35 4C A9 36 AD FA Ø3 C9 7B
35A9:2B DØ ØD A9 F2 AØ Ø3 2Ø 5A
35B1:67 B8 2Ø 83 35 4C A9 36 9E
35B9:C9 2D DØ ØD A9 F2 AØ Ø3 9C
35C1:2Ø 5Ø B8 2Ø 83 35 4C A9 9C
35C9:36 AD FA Ø3 C9 3C DØ Ø8 33
35D1:A9 Ø1 8D CØ Ø2 4C F4 35 6F
35D9:C9 3E DØ Ø8 A9 FF 8D CØ 7C
35E1:Ø2 4C F4 35 C9 3D DØ Ø8 3F
35E9:A9 ØØ 8D CØ Ø2 4C F4 35 47
35F1:4C 72 36 A9 Ø3 8D C5 Ø2 5C
35F9:AD BF Ø2 C9 3D DØ Ø8 A9 EE
3601:ØØ 8D 36 Ø3 4C 25 36 C9 F4
3609:3E DØ Ø8 A9 FF 8D 36 Ø3 ØA
3611:4C 25 36 C9 3C DØ Ø8 A9 2F
3619:Ø1 8D 36 Ø3 4C 25 36 A9 6D
3621:52 8D 36 Ø3 A9 F2 AØ Ø3 6E
3629:2Ø 5B BC CD CØ Ø2 FØ ØD ED
3631:CD 36 Ø3 FØ Ø8 A9 ØØ 8D F5
```

```
3639:CØ Ø2 4C 46 36 A9 FF 8D 5A
3641:CØ Ø2 4C 46 36 AD CØ Ø2 68
3649:DØ ØD A9 3Ø 8D C9 Ø2 A9 DA
3651:ØØ 8D CA Ø2 4C 67 36 A9 BØ
3659:2D 8D C9 Ø2 A9 31 8D CA 11
3661:Ø2 A9 ØØ 8D CB Ø2 A9 46 12
3669:8D EE CC 4C CA 32 4C A1 1Ø
3671:36 C9 41 DØ Ø3 4C 7D 36 1B
3679:C9 4F DØ 24 2Ø ØF BC A9 5B
3681:F2 AØ Ø3 2Ø A2 BB AD FA 4C
3689:Ø3 C9 41 DØ ØB 2Ø E9 AF 7B
3691:A9 Ø3 8D C5 Ø2 4C A1 36 5C
3699:2Ø E6 AF A9 Ø3 8D C5 Ø2 3C
36A1:A9 ØØ 8D EE CC 2Ø 83 35 A6
36A9:AD FC Ø3 8D FA Ø3 AD 3A DE
36B1:Ø3 DØ Ø6 AE 3B Ø3 4C CF CD
36B9:3Ø AD C5 Ø2 C9 Ø4 DØ 17 99
36C1:AD FF CC CD 34 Ø3 DØ ØF D9
36C9:A2 ØØ BD ØØ CF 9D A2 Ø3 7C
36D1:FØ Ø4 E8 4C CB 36 6Ø 4C DD
36D9:9A 37 A2 ØØ AC 39 Ø3 C8 CE
36E1:B9 A2 Ø3 FØ Ø8 9D ØØ CD C7
36E9:E8 C8 4C E1 36 9D ØØ CD 9A
36F1:AD C5 Ø2 C9 28 DØ ØE AC DØ
36F9:FF CC C8 C8 C8 A9 Ø4 8D C1
3701:C5 Ø2 4C 25 37 A9 F2 AØ 95
3709:Ø3 2Ø A2 BB 2Ø DD BD A2 A7
3711:ØØ AC FF CC BD ØØ Ø1 FØ 58
3719:Ø8 99 A2 Ø3 C8 E8 4C 15 ØE
3721:37 99 A2 Ø3 A2 ØØ BD ØØ A6
3729:CD FØ Ø8 99 A2 Ø3 C8 E8 FØ
3731:4C 27 37 99 A2 Ø3 88 AD EF
3739:A2 Ø3 C9 28 DØ Ø3 4C 3F DF
3741:3Ø AE 13 Ø3 8E 37 Ø3 AD ØB
3749:C5 Ø2 C9 Ø1 FØ Ø1 6Ø 2Ø DØ
3751:B6 39 AD 13 Ø3 C9 15 FØ AA
3759:28 AE 37 Ø3 8E 13 Ø3 A9 ØF
3761:F2 AØ Ø3 2Ø A2 BB 2Ø CC E4
3769:BC A2 F2 AØ Ø3 2Ø D4 BB 45
3771:2Ø DD BD A2 ØØ BD ØØ Ø1 41
3779:9D A2 Ø3 FØ ØF E8 4C 8E 12
3781:37 A9 Ø2 8D C5 Ø2 AE 37 D9
3789:Ø3 8E 13 Ø3 6Ø AD F2 Ø3 52
3791:C9 9Ø BØ Ø1 6Ø EE C5 Ø2 7B
```

```
3799:60 AD FF CC CD 34 03 D0 86
37A1:03 4C DB 36 CE FF CC AA 3E
37A9:CA BD A2 03 C9 20 D0 12 F3
37B1:CA EC 34 03 D0 06 EE FF F3
37B9:CC 4C DB 36 CE FF CC 4C DC
37C1:AA 37 86 02 EE FF CC BD 13
37C9:A2 03 C9 2A F0 2C C9 2F 21
37D1:F0 28 C9 2B F0 24 C9 2D 87
37D9:F0 20 C9 20 F0 1C C9 28 B7
37E1:F0 18 C9 3E F0 14 C9 3C 93
37E9:F0 10 C9 3D F0 0C EC 34 A7
37F1:03 D0 03 4C CF 30 CA 4C 5C
37F9:C8 37 A4 02 E4 02 D0 03 23
3801:4C DB 36 AD FF CC 8D C0 3F
3809:02 8E FF CC EE FF CC E8 E4
3811:86 02 20 1F 3C AD 36 03 43
3819:C9 06 B0 03 4C 24 38 C9 63
3821:18 90 0A C9 02 90 0C A9 B3
3829:01 8D 13 03 60 20 49 38 5E
3831:4C DB 36 AD C0 02 8D FF 89
3839:CC EE FF CC 4C DB 36 AD 84
3841:13 03 F0 01 60 4C DB 36 4C
3849:38 E9 06 0A A8 B9 8B 3B 30
3851:8D C9 02 B9 8C 3B 8D CA 0E
3859:02 A9 F2 A0 03 20 A2 BB 37
3861:6C C9 02 20 58 BC 20 83 36
3869:35 60 20 6B E2 20 83 35 1B
3871:AD C5 02 C9 03 90 05 A9 15
3879:15 8D 13 03 A9 02 8D C5 A0
3881:02 60 20 CC BC 20 83 35 7E
3889:AD C5 02 C9 03 90 05 A9 2D
3891:14 8D 13 03 A9 01 8D C5 34
3899:02 60 20 0E E3 4C 6E 38 6D
38A1:A9 4F 8D FF CF 20 F7 39 94
38A9:AD C5 02 C9 04 D0 03 4C F4
38B1:E0 38 20 83 35 60 20 C7 10
38B9:1A AD 14 C3 8D A0 C5 AD 89
38C1:15 C3 8D A1 C5 AD 16 C3 4E
38C9:8D A2 C5 60 A9 F2 A0 03 C5
38D1:20 28 BA 20 83 35 60 20 87
38D9:B6 39 20 A1 B7 86 02 AC 9A
38E1:FF CC A9 27 99 A2 03 C8 53
38E9:A5 02 99 A2 03 C8 A9 27 C0
38F1:99 A2 03 A9 28 8D C5 02 D7
```

```
38F9:60 20 ED BF 4C 6E 38 A9 92
3901:50 8D FF CF 4C A6 38 20 89
3909:71 BF 4C 6E 38 A9 80 8D 8B
3911:C9 02 A9 00 8D CA 02 8D 47
3919:CB 02 8D CC 02 8D CD 02 54
3921:A9 C9 A0 02 20 67 B8 4C 6B
3929:83 38 A9 53 8D FF CF 4C 2E
3931:A6 38 A2 01 2C A2 00 BD 13
3939:30 C5 F0 59 4C 80 39 20 DF
3941:EA B9 20 83 35 60 A9 03 55
3949:8D C5 02 A9 82 8D C9 02 AE
3951:A9 00 8D CA 02 8D CB 02 D6
3959:8D CC 02 8D CD 02 A9 C9 72
3961:A0 02 20 A2 BB A9 F2 A0 DD
3969:03 20 0F BB A2 98 A0 03 BE
3971:20 D4 BB 20 CC BC A9 98 E7
3979:A0 03 20 5B BC F0 16 A9 36
3981:81 8D F2 03 A9 80 8D F3 05
3989:03 A9 00 8D F4 03 8D F5 85
3991:03 8D F6 03 60 A9 00 8D 2F
3999:F2 03 8D F3 03 8D F4 03 72
39A1:8D F5 03 8D F6 03 60 20 36
39A9:64 E2 20 83 35 60 20 D4 83
39B1:AE 20 83 35 60 A2 00 A0 75
39B9:CD 20 D4 BB A9 8E A0 03 3D
39C1:20 A2 BB 20 CC BC A9 8E A1
39C9:A0 03 20 5B BC F0 0E A9 76
39D1:15 8D 13 03 60 A9 00 A0 0F
39D9:CD 20 A2 BB 60 A9 00 A0 95
39E1:CD 20 A2 BB 20 CC BC A9 AA
39E9:00 A0 CD 20 5B BC F0 E5 D5
39F1:A9 14 8D 13 03 60 AD F5 0C
39F9:CC 4C FF 39 C9 04 AD FF 33
3A01:CF C9 4F D0 11 AD C5 02 93
3A09:C9 04 F0 2A 60 20 A2 B3 A0
3A11:A9 01 8D C5 02 60 AE C5 5D
3A19:02 E0 04 D0 06 8D C5 02 48
3A21:4C 37 3A C9 50 D0 46 A9 69
3A29:BC A0 B9 20 A2 BB A9 F2 A7
3A31:A0 03 20 50 B8 60 AC C2 23
3A39:02 B9 A2 03 C9 27 F0 06 74
3A41:C8 CC 35 03 D0 F3 8C F5 89
3A49:CC AD F5 CC 18 69 01 AA 2E
3A51:BD A2 03 A8 AD C5 02 C9 8A
```

```
3A59:5Ø FØ ØE C9 Ø4 FØ AE C8 9A
3A61:98 85 Ø2 A9 Ø4 8D C5 Ø2 42
3A69:6Ø 88 4C 61 3A A9 BC AØ 62
3A71:B9 2Ø A2 BB A9 F2 AØ Ø3 38
3A79:2Ø 67 B8 6Ø AE C2 Ø2 E8 62
3A81:AØ ØØ A5 Ø2 8D F2 CC BD AA
3A89:A2 Ø3 E4 Ø2 FØ Ø8 99 3C E3
3A91:Ø3 E8 C8 4C 88 3A A9 2Ø 4Ø
3A99:99 3C Ø3 A9 ØØ 8D Ø3 C5 E6
3AA1:2Ø 7B 12 2Ø 72 26 AD 19 E9
3AA9:C5 FØ 75 C9 Ø1 FØ 23 C9 64
3AB1:Ø7 FØ 1F C9 Ø2 FØ 16 C9 3Ø
3AB9:Ø3 FØ 25 C9 Ø4 FØ 2D C9 35
3AC1:Ø8 FØ 29 C9 ØC FØ Ø3 4C AE
3AC9:9A 37 4C 5B 3B A9 Ø2 8D AA
3AD1:C5 Ø2 A2 Ø1 98 18 65 FB F9
3AD9:A4 FC 2Ø A2 BB 4C EC 32 29
3AE1:B1 FB C9 2D DØ Ø3 4C 8C F1
3AE9:33 4C A3 33 B1 FB 8D C9 15
3AF1:Ø2 AC F2 CC A2 ØØ B9 A2 E8
3AF9:Ø3 FØ Ø8 9D ØØ CD E8 C8 D8
3BØ1:4C F7 3A 9D ØØ CD AC C2 1Ø
3BØ9:Ø2 C8 A9 27 99 A2 Ø3 C8 8Ø
3B11:AD C9 Ø2 99 A2 Ø3 C8 A9 Ø7
3B19:27 99 A2 Ø3 C8 4C 25 37 Ø7
3B21:A9 ØØ 8D Ø3 C5 2Ø 7B 12 Ø6
3B29:2Ø 82 2D AD 19 C5 FØ 24 B6
3B31:C9 Ø4 FØ Ø3 4C AC 3A A2 Ø8
3B39:ØØ B1 FB 9D A2 Ø3 9D C9 9B
3B41:Ø2 FØ Ø7 E8 2Ø ØD Ø8 4C F5
3B49:3A 3B EØ Ø3 FØ 29 A9 Ø4 7B
3B51:8D C5 Ø2 6Ø A9 Ø1 8D 13 C5
3B59:Ø3 6Ø A2 Ø1 A9 27 8D C9 9C
3B61:Ø2 B1 FB FØ Ø7 9D C9 Ø2 18
3B69:E8 4C 62 3B A9 27 9D C9 56
3B71:Ø2 E8 A9 ØØ 9D C9 Ø2 4C BC
3B79:55 33 AØ ØØ B1 FB 91 FD 1A
3B81:C8 CC 37 Ø3 FØ Ø3 4C 7D 5Ø
3B89:3B 6Ø 64 38 6B 38 83 38 41
3B91:9B 38 A1 38 CD 38 D8 38 D4
3B99:FA 38 ØØ 39 Ø8 39 A8 39 DE
3BA1:4Ø 39 ØE 39 2B 39 33 39 B9
3BA9:36 39 47 39 AF 39 4F 52 59
```

```
3BB1:20 00 41 4E 44 20 00 44 2C
3BB9:49 56 20 00 4D 4F 44 20 BE
3BC1:00 54 52 55 45 00 46 41 E4
3BC9:4C 53 45 00 41 42 53 00 9D
3BD1:53 49 4E 00 54 52 55 4E F2
3BD9:43 00 41 52 43 54 41 4E 7B
3BE1:00 4F 52 44 00 53 51 52 FC
3BE9:00 43 48 52 00 45 58 50 75
3BF1:00 50 52 45 44 00 53 51 35
3BF9:52 54 00 43 4F 53 00 4C F6
3C01:4E 00 52 4F 55 4E 44 00 4C
3C09:53 55 43 43 00 45 4F 46 17
3C11:00 45 4F 4C 4E 00 4F 44 DE
3C19:44 00 4E 4F 54 2A A9 A2 B3
3C21:85 FB A9 03 85 FC 8E FD FB
3C29:03 C8 8C FE 03 8A A8 A9 14
3C31:00 8D 36 03 AA B1 FB DD F5
3C39:AF 3B D0 12 E8 C8 CC FE 96
3C41:03 B0 03 4C 36 3C AE FD 8A
3C49:03 AC FE 03 88 60 EE 36 58
3C51:03 BD AF 3B F0 08 C9 2A C9
3C59:F0 0B E8 4C 52 3C E8 AC F0
3C61:FD 03 4C 36 3C A9 FF 8D 9C
3C69:36 03 4C 47 3C 3A 00 00 86
```

Index

To order your copy of the *Pascal for Beginners Disk*, call our toll-free US order line: 1-800-346-6767 (in NY 212-887-8525) or send your prepaid order to:

Pascal for Beginners Disk
COMPUTE! Publications
P.O. Box 5038
F.D.R. Station
New York, NY 10150

All orders must be prepaid (check, charge, or money order). NC residents add 5% sales tax. NY residents add 8.25% sales tax.

Send _____ copies of *Pascal for Beginners Disk* at $12.95 per copy. (688BDSK)

Subtotal $_____

Shipping and Handling: $2.00/disk $_____

Sales tax (if applicable) $_____

Total payment enclosed $_____

□ Payment enclosed
□ Charge □ Visa □ MasterCard □ American Express

Acct. No. _____ Exp. Date _____
(Required)

Name _____

Address _____

City _____ State _____ Zip _____

Please allow 4-5 weeks for delivery.

COMPUTE! Books

Ask your retailer for these **COMPUTE! Books** or order directly from **COMPUTE!**.

Call toll free (in US) **800-346-6767** (in NY 212-887-8525) or write COMPUTE! Books, P.O. Box 5038, F.D.R. Station, New York, NY 10150.

Quantity	Title	Price*	Total
_____	SpeedScript: The Word Processor for the Commodore 64 and VIC-20 (94-9)	**$ 9.95**	_____
_____	Commodore SpeedScript Disk	**$12.95**	_____
_____	128 Machine Language for Beginners (033-5)	**$16.95**	_____
_____	COMPUTE!'s First Book of the Commodore 128 (059-9)	**$14.95**	_____
_____	COMPUTE!'s Commodore 64/128 Collection (97-3)	**$12.95**	_____
_____	All About the Commodore 64, Volume Two (45-0)	**$16.95**	_____
_____	All About the Commodore 64, Volume One (40-X)	**$12.95**	_____
_____	Programming the Commodore 64: The Definitive Guide (50-7)	**$24.95**	_____
_____	COMPUTE!'s Data File Handler for the Commodore 64 (86-8)	**$12.95**	_____
_____	COMPUTE!'s Kids and the Commodore 128 (032)	**$14.95**	_____
_____	COMPUTE!'s Kids and the Commodore 64 (77-9)	**$12.95**	_____
_____	COMPUTE!'s Commodore Collection, Volume 1 (55-8)	**$12.95**	_____
_____	COMPUTE!'s Commodore Collection, Volume 2 (70-1)	**$12.95**	_____
_____	COMPUTE!'s Personal Accounting Manager for the Commodore 64 (014-9)	**$12.95**	_____
_____	COMPUTE!'s VIC-20 and Commodore 64 Tool Kit: BASIC (32-9)	**$16.95**	_____
_____	COMPUTE!'s VIC-20 and Commodore 64 Tool Kit: Kernal (33-7)	**$16.95**	_____
_____	COMPUTE!'s Telecomputing on the Commodore 64 (009)	**$12.95**	_____
_____	COMPUTE!'s 128 Programmer's Guide (0319)	**$16.95**	_____
_____	The Complete 64 (Book/Disk Combination) (0629)	**$29.95**	_____
_____	COMPUTE!'s Machine Language Games for the Commodore 64 (0610)	**$16.95**	_____
_____	Mapping the 128 (060-2)	**$16.95**	_____

*Add $2.00 per book for shipping and handling.
Outside US add $5.00 air mail or $2.00 surface mail.

NC residents add 5% sales tax _____
NY residents add 8.25% sales tax _____
Shipping & handling: $2.00/book _____
Total payment _____

All orders must be prepaid (check, charge, or money order).
All payments must be in US funds.
☐ Payment enclosed.
Charge ☐ Visa ☐ MasterCard ☐ American Express

Acct. No._____ Exp. Date_____
(Required)

Name_____

Address_____

City_____ State _____ Zip_____

*Allow 4–5 weeks for delivery.
Prices and availability subject to change.
Current catalog available upon request.

Learn Pascal

The complete guide to standard Pascal, *COMPUTE!'s Pascal for Beginners* explains, with selected program examples, everything you'll need to know to begin writing your own Pascal applications on any computer with a standard Pascal interpreter or compiler. And, if you own a Commodore 64 or 128, *COMPUTE!'s Pascal for Beginners* includes an introductory (not full-featured) machine language Pascal interpreter that understands most of the Pascal keywords a beginner needs to start programming in Pascal on the Commodore 64 and 128.

COMPUTE!'s Pascal for Beginners is more than a book that effectively teaches Pascal. It also shows you how to write structured programs that take advantage of the advanced features of Pascal.

Here's a sample of what's included:

- Clear explanations, with numerous program examples, of every standard Pascal keyword
- How to declare your own data types
- Save memory by using packed arrays
- When and how to use each of the three types of program loops
- Dozens of program examples that you can study and modify
- How to declare and use record variables
- Write your own procedures and functions
- How to take advantage of dynamic data structures
- An introductory (not full-featured) machine language Pascal interpreter for the Commodore 64 and 128

Whether you're new to programming or have been programming for years, you'll find that *COMPUTE!'s Pascal for Beginners* is the perfect introductory guide to this increasingly popular computer language. And, since it's from COMPUTE! Books, you know it's written in the clear, concise style that has become the hallmark of all COMPUTE! publications.

Most of the programs in this book, including the introductory Pascal interpreter, are available on a companion disk for the Commodore 64 and 128. See the coupon in the back for details.

ISBN 0-87455-068-8

$14.95