Introduction to the TURBO Pascal Tutorial

Assuming you know nothing at all about Pascal, and in fact, that you may know nothing about programming in general, we will begin to study Pascal. If you are already somewhat familiar with programming and especially Pascal, you will probably want to skip very quickly through the first few chapters. You should at least skim the first few chapters, and you should read the remainder of this introduction.

A few comments are in order to get us started in the right direction. The sample programs included on the disks are designed to teach you the basics of Pascal and they do not include any clever or tricky code. Nearly all of the programs are really quite dumb as far as being useful programs, but all will teach one or more principles of Pascal. I have seen one tutorial that included a 12 page program as the first example. In fact there were only 2 example programs in the entire tutorial, and it was impossible to glean the essentials of programming from that system. For this reason, I will completely bypass any long programs until the very end of this tutorial. In order to illustrate fundamental concepts used in Pascal programming, all programs will be very short and concise until we reach the last chapter.

The last chapter has some rather large programs to illustrate to you how to write a large program. It would be a disservice to you to show you all of the constructs of Pascal and not show you how to put them together in a meaningful way to build a large program. After completing all of the fundamentals of Pascal, it will then be very easy for you to use the tools learned to build as large a program as you desire.

Another problem I have noticed in example programs is the use of one word for all definitions. For example, a sort program is stored in a file called SORT, the program is named Sort, and various parts of the program are referred to as Sort1, Sort2, etc. This can be confusing since you have no idea if the program name must be the same as the filename, or if any of the other names were chosen to be the same because of some obscure rule not clearly documented. For this reason, the example programs use completely

arbitrary names whenever the choice of a name adds nothing to the readability or clarity of a program. As an illustration of this, the first program is named Puppy_Dog. This adds nothing to the understanding of the program but does illustrate that the program name means nothing to the Pascal compiler concerning what the program does.

Introduction to the TURBO Pascal Tutorial

Due to the fundamental design of the Pascal language, certain words are "reserved" and can only be used for their defined purposes. These are listed in your TURBO Pascal reference manual (page 37 - version 3.0; page 196 - version 4.0). All of the sample programs in this tutorial are written with the reserved words in all lower-case letters, and the user variables in lower case with the first letter capitalized since this is becoming the accepted industry standard. Don't worry about what reserved words are yet, they will be completely defined later.

WHAT IS A COMPILER?

There are two primary methods used in running any computer program that is written in a readable form of English. The first method is an interpreter. An interpreter is a program that looks at each line of the "English" program, decides what the "English" on that line means, and does what it says to do. If one of the lines is executed repeatedly, it must be scanned and analyzed each time, greatly slowing down the solution of the problem at hand. A compiler, on the other hand, is a program that looks at each statement one time and converts it into a code that the computer understands directly. When the compiled program is actually run, the computer does not have to figure out what each statement means, it is already in a form that the computer can run directly, hence a much faster execution of the program.

This tutorial is written especially for Borland International's TURBO Pascal compilers version 3.0 or version 4.0. These are very high quality compilers that can do nearly anything you will ask them to do since they are so flexible. The original intent of this tutorial was to write it in such a way that it would be completely generic and

usable with any good Pascal compiler. The programmers at Borland included a great many nonstandard aids for the Pascal language and resulted in a very good product that has dominated the market for microcomputers. To completely omit all of the extensions would do those of you with the Borland compiler a real disservice, and to include the extensions would not allow other compilers to be used effectively with this tutorial.

The decision was made to stay with the Borland extensions and make the tutorial very difficult to use with other compilers. TURBO Pascal is so inexpensive that it would be a wise decision to purchase a copy solely for the purpose of learning the Pascal programming language then moving to a larger compiler on a minicomputer or a mainframe using the accumulated knowledge to very quickly learn the

extensions provided by that particular compiler. At any rate, this tutorial will not teach you everything you will ever need to know about Pascal. It will, however, teach you the fundamentals and the terminology needed to progress on your own into more advanced topics of Pascal and programming in general. You will find that experience will be your best teacher.

Some of the example files will only work properly with TURBO Pascal version 3.0 and some will only work properly with version 4.0, but most will work with either. It will be clearly indicated to you which files will work with each of the two versions of TURBO Pascal.

WHAT ABOUT TURBO PASCAL VERSION 2.0?

Most of the files will compile properly with TURBO Pascal version 2.0, but no warning will be given since that version has been superseded for so long. It will pay you to purchase a newer version because of the flexibility. If you choose not to however, this tutorial will work fine in most cases if you follow the instructions for TURBO Pascal version 3.0.

PREPARATION FOR USE OF THIS TUTORIAL.

Copy the example files onto your TURBO working disk and you are ready to begin, provided of course that you have already learned how to start the TURBO system and how to edit a Pascal file. Be sure you make a backup copy of the Pascal tutorial disks so you cannot accidentally lose all information on the distribution disks. If you are using TURBO Pascal version 3.0, you should read Chapter 1 of the reference manual to be ready to use this tutorial, and if you are using TURBO Pascal version 4.0, you should read parts of chapters 1, 2, & 11 of your reference manual. You should be familiar with use of the editor supplied with TURBO Pascal before beginning.

If you are not using TURBO Pascal, you will still be able to compile and execute many of these Pascal files, since most of the examples use standard Pascal syntax. There will be some statements used which are unique to TURBO Pascal and will probably not work with your compiler. This will be especially true when you come to the chapter on standard input and output since this is where most compilers differ. Unfortunately, this is one of the most important aspects of any programming language, since it is required to get data into and out of the computer to do anything useful.

Introduction to the TURBO Pascal Tutorial

It is highly suggested that you do the programming exercises after you complete the study for each chapter. They are carefully selected to test your understanding of the material covered in that chapter. If you do not write,

enter, debug, and run these programs, you will only be proficient at reading Pascal. If you do the exercises completely, you will have a good start at being a Pascal program writer.

It should also be mentioned that this tutorial will not teach you everything you will ever need to know about Pascal. You will continue to learn new techniques as long as you continue to write programs. Experience is the best teacher here just as it is in any endeavor. This tutorial will teach you enough about Pascal that you should feel very comfortable as you search through the reference manual for some topic. You will also be able to read and understand any Pascal program you find in textbooks or magazines.

When you are ready, I will meet you in Chapter 1.

CHAPTER 1 - What is a computer program?

If you are a complete novice to computers you will find the information in this chapter useful. If however, you have had some experience with programming, you can completely ignore this chapter. It will deal with a few fundamentals of computers in general and will introduce nothing that is specific to Pascal.

WHAT IS A COMPUTER PROGRAM?

A computer is nothing but a very dumb machine that has the ability to perform mathematical operations very rapidly and very accurately, but it can do nothing without the aid of a program written by a human being. Moreover, if the human being writes a program that turns good data into garbage, the computer will very obediently, and very rapidly, turn the good data into garbage. It is possible to write a computer program with one small error in it that will do that very thing, and in some cases appearing to be generating good data. It is up to the human programmer to design a program to achieve the desired results.

A computer program is simply a "recipe" which the computer will use on the input data to derive the desired output data. It is similar to the recipe for baking a cake. The input data is comparable to the ingredients, including the heat supplied by the oven. The program is comparable to the recipe instructions to mix, stir, wait, heat, cool, and all other possible operations on the ingredients. The output of the computer program can be compared to the final cake sitting on the counter ready to be cut and served. A computer program then is composed of two parts, the data upon which the program operates, and the program that operates on the data. The data and program are inseparable as implied by the last sentence.

WHAT ARE CONSTANTS?

Nearly any computer program requires some numbers that never change throughout the program. They can be defined once and used as often as needed during the operation of the

program. To return to the recipe analogy, once you have defined how big a tablespoon is, you can use the same tablespoon without regard to what you are measuring with it. When writing a computer program, you can define the value of PI = 3.141592, and continue to use it wherever it makes sense knowing that it is available, and correct.

WHAT ARE VARIABLES?

In addition to constants, nearly every computer program uses some numbers that change in value throughout the

program. They can be defined as variables, then changed to any values that make sense to the proper operation of the program. An example would be the number of eggs in the above recipe. If a single layer of cake required 2 eggs, then a triple layer cake would require 6 eggs. The number of eggs would therefore be a variable.

HOW DO WE DEFINE CONSTANTS OR VARIABLES?

All constants and variables have a name and a value. In the last example, the name of the variable was "eggs", and the value was either 2 or 6 depending on when we looked at the stored data. In a computer program the constants and variables are given names in much the same manner, after which they can store any value within the defined range. Any computer programming language has a means by which constants or variables can be first named, then assigned a value. The means for doing this in Pascal will be given throughout the remainder of this tutorial.

WHAT IS SO GOOD ABOUT PASCAL?

Some computer languages allow the programmer to define constants and variables in a very haphazard manner and then combine data in an even more haphazard manner. For example, if you added the number of eggs, in the above recipe, to the number of cups of flour, you would arrive a a valid mathematical addition, but a totally meaningless number. Some programming languages would allow you to do just such an addition and obediently print out the meaningless answer. Since Pascal requires you to set up your constants and variables in a very precise manner, the possibility of such a meaningless answer is minimized. A well written Pascal program has many cross checks to minimize the possibility of a completely scrambled and meaningless output.

Notice however, in the last statement, that a "well written" Pascal program was under discussion. It is still up to the programmer to define the data structure in such a way that the program can prevent garbage generation. In the end, the program will be no better than the analysis that went into the program design.

If you are a novice programmer, do not be intimidated by any of the above statements. Pascal is a well designed, useful tool that has been used successfully by many computer novices and professionals. With these few warnings, you are ready to begin.

CHAPTER 2 - Getting started in Pascal

YOUR FIRST PASCAL PROGRAM

Lets get right into a program that really does nothing but is an example of the most trivial Pascal program. Load Turbo Pascal, select TRIVIAL as a Work file, and select Edit. This assumes that you have been successful in learning how to use the TURBO Pascal system. If you are using TURBO Pascal 4.0, you will need to load TRIVIAL.PAS from the File menu.

You should now have the most trivial Pascal program possible on your display, and we can take a look at each part to define what it does.

The first line is required in the standard Pascal definition and is the program name which can be any name you like, as long as it follows the rules for an identifier given in the next paragraph. It can have no blanks, otherwise it would be considered as two words and it would confuse the compiler. The first word "program" is the first of the reserved words mentioned earlier and it is the indicator to the Pascal compiler that this is the name of the program. Notice that the line ends with a semicolon. Pascal uses the semicolon as the statement separator and although all statements do not actually end in a semicolon, most do, and use of the semicolon will clear up later in your mind.

TURBO Pascal version 3.0 does not require the "program" statement, but to remain compatible with standard Pascal, it will simply ignore the entire statement. I like to include a program name both to keep me thinking in standard Pascal, and to add a little more indication of the purpose of each program.

WHAT IS AN IDENTIFIER?

All identifiers, including program name, procedure and function names, type definitions, and constant and variable names, will start with an alphabetical character and be composed of any combination of alphabetic and numeric characters with no embedded blanks. Upper or lower case alphabetic characters are not significant and may be mixed at will. (If you find this definition confusing at this point, don't worry about it, it will be clear later but it must be defined early). The standard definition of Pascal requires that any implementation (i.e. any compiler written by some company) must use at least 8 characters of the identifier as significant and may ignore the remaining characters if more are used. Most implementations use far

more than 8. TURBO Pascal uses at least 63 characters in an identifier as being significant.

Standard Pascal does not allow the use of underlines in an identifier but most implementations of Pascal allow its use after the first character. Both TURBO Pascal compilers allow the use of the underline as an allowable character in an identifier, so it will be freely used throughout this tutorial. The underline is used in the program name "Puppy_Dog" which should be on your display at this time.

Returning to the example program, the next line is a blank line which is ignored by all Pascal compilers. More will be said about that at the end of this chapter.

NOW FOR THE PROGRAM

The next two lines comprise the actual Pascal program, which in this case does absolutely nothing. It is an illustration of the minimum Pascal program. The two words "begin" and "end" are the next two reserved words we will consider. Any logical grouping of Pascal code can be isolated by bracketing it with the two reserved words "begin" and "end". You will use this construct repeatedly as you write Pascal code so it is well to learn it thoroughly. Code to be executed by conditional jumps will be bracketed by "begin" and "end", as will code within a loop, and code contained within a subroutine (although they are called "procedures" in Pascal), and in many other ways. In the present program, the "begin" and "end" are used to bracket the main program and every Pascal program will have the main program bracketed in this manner. Because there is nothing to do in this program, there are no statements.

Finally, although it could be very easily overlooked, there is one more very important part of the program, the period following "end". The period is the signal to the

compiler that it has reached the end of the executable statements and is therefore finished compiling. Every Pascal program will have one, and only one period in it and that one period will be at the end of the program. I must qualify that statement in this regard, a period can be used in comments, and in text to be output. In fact there are some data formats that require using a period as part of their structure. The statement is true however, that there is only one period in the executable part of a Pascal program. Think of a Pascal program as one long sentence with one period at the end.

That should pretty well describe our first program. Now it is time to compile and run it. To do so you must

exit the editor using Ctrl-K-D, unless you modified the exit command.

Compile the program, and run it to observe the result. Since this program doesn't do anything, it is not very interesting, so let's get one that does something.

A PROGRAM THAT DOES SOMETHING

Load the Pascal program WRITESM and view it on your monitor. The filename is sort of cryptic for "Write Some" and it will give a little output to the monitor. The program name is "Kitty_Cat" which says nothing about the program itself but can be any identifier we choose. We still have the begin and end to define the main program area followed by the period. However, now we have two additional statements between the begin and end. "Writeln" is a special word and it is probably not surprising that it means

to write a line of data somewhere. Without a modifier, (to be explained in due time), it will write to the default device which, in the case of our IBM compatible, is the video display. The data within the parentheses is the data to be output to the display and although there are many possibilities of display information, we will restrict ourselves to the simplest for the time being. Any data between apostrophes will simply be output as text information.

The special word "Writeln" is not a reserved word but is defined by the system to do a very special job for you, namely to output a line of data to the monitor. It is, in fact, a procedure supplied for you by the writers of TURBO Pascal as a programming aid for you. You can, if you so desire, use this name for some other purpose in your program but doing so will not allow you to use the standard output procedure. It will then be up to you to somehow get your data out of the program.

Note carefully that some words are reserved and cannot be redefined and used for some other purpose, and some are special since they can be redefined. You will probably not want to redefine any of the special words for a long time so simply use them as tools.

Notice the semicolon at the end of line 4. This is the statement separator referred to earlier and tells Pascal that this line is complete as it stands, nothing more is coming that could be considered part of this statement. The next statement, in line 5, is another statement that will be executed sequentially following the statement in line 4. This program will output the two lines of text and stop.

Now it is time to go try it. Exit the editor, then compile and run the program.

You should get the two lines of text output to the video display every time you run it. When you grow bored of running WRITESM lets go on to another example.

ANOTHER PROGRAM WITH MORE OUTPUT

Load and edit WRITEMR. This new program has three lines of output but the first two are different because another special word is introduced to us, namely Write. Write causes the text to be output in exactly the same manner as Writeln, but Write does not cause a carriage return. Writeln causes its output to take place then returns the "carriage" to the first character of the next line. The end result is that all three of the lines will be output on the same line when the program is run. Notice that there is a blank at the end of each of the first two lines so that the formatting will look nice. Exit the editor now and try the new program.

Now might be a good time for you to return to editing WRITEMR and add a few more output commands to see if they do what you think they should do. When you tire of that, we will go on to the next file and learn about comments within a Pascal program.

ADDING COMMENTS IN THE PROGRAM

The file named PASCOMS is similar to the others except that comments have been added to illustrate their use. Pascal defines comments as anything between (* and *) or anything between { and }. Originally only the wiggly brackets were defined but since many keyboards didn't have them available, the parenthesis star combination was defined as an extension and is universal by now, so you can use either. Most of the comments are self explanatory except for the one within the code. Since comments can go from line to line, the two lines that would print "send money" are not Pascal code but are commented out. Try compiling and running this program, then edit the comments out so that "send money" is printed also.

A fine point should be mentioned here. Even though some compilers allow comments to start with (* and end with

}, or to start with { and end with *), it is very poor programming practice and should be discouraged. The ANSI Pascal standard allows such usage but TURBO Pascal does not allow this funny use of comment delimiters.

TURBO Pascal does not allow you to nest comments using the same delimiters but it does allow you to nest one type within the other. This could be used as a debugging aid. If you generally use the (* and *) for comments, you could use the { and } in TURBO Pascal to comment out an entire section of code during debugging even if it had a few comments in it. This is a trick you should remember when you reach the point of writing programs of significant size.

When you have successfully modified and run the program with comments, we will go on to explain good formatting practice and how Pascal actually searches through your source file (Pascal program) for its executable statements.

It should be mentioned that the program PASCOMS does not indicate good commenting style. The program is meant to illustrate where and how comments can be used and looks very choppy and unorganized. Further examples will illustrate good use of comments to you as you progress through this tutorial.

GOOD FORMATTING PRACTICE

Observe GOODFORM to see an example of good formatting style. It is important to note that Pascal doesn't give a hoot where you put carriage returns or how many blanks you put in when a blank is called for as a delimiter. Pascal

only uses the combination of reserved words and end-of-statement semicolons to determine the logical structure of the program. Since we have really only covered two executable statements, I have used them to build a nice looking program that can be easily understood at a glance. Compile and run this program to see that it really does what you think it should do.

VERY POOR FORMATTING PRACTICE

Edit UGLYFORM now to see an example of terrible formatting style. It is not really apparent at a glance but the program you are looking at is exactly the same program as the last one. Pascal doesn't care which one you ask it to run because to Pascal, they are identical. To you they are considerably different, and the second one would be a mess to try to modify or maintain sometime in the future.

UGLYFORM should be a good indication to you that Pascal doesn't care about programming style or form. Pascal only cares about the structure, including reserved words and delimiters such as blanks and semicolons. Carriage returns are completely ignored as are extra blanks. You can put extra blanks nearly anywhere except within reserved words or

variable names. You should pay some attention to programming style but don't get too worried about it yet. As time goes by you will develop a style of statement indentation, adding blank lines for clarity, and a method of adding clear comments to Pascal source code. Programs are available to read your source code, and put it in a "pretty" format, but that is not important now.

Not only is the form of the program important, the names used for variables can be very helpful or hindering as we will see in the next chapter. Feel free to move things around and modify the format of any of the programs we have covered so far and when you are ready, we will start on variables in the next chapter.

Be sure you compile and run UGLYFORM.

PROGRAMMING EXERCISES

1. Write a program that displays your name on the video monitor.

2. Modify your program to display your name and address on one line, then modify it by changing the Write's to Writeln's so that the name and address are on different lines.

CHAPTER 3 - The simple Pascal data types

TURBO Pascal has 5 basic data types which are predefined and can be used anywhere in a program provided you use them properly. The five types and a very brief description follows;

integer Whole numbers from -32768 to 32767
byte The integers from 0 to 255
real Floating point numbers from 1E-38 to 1E+38
boolean Can only have the value TRUE or FALSE
char Any character in the ASCII character set

Please note that the byte type of data is not a part of the standard Pascal definition but is included as an extension to the TURBO Pascal compiler.

TURBO Pascal version 4.0 has three additional "integer" types available which are not available with version 3.0, and they are defined as follows;

shortint The integers from -128 to 127
word The integers from 0 to 65535
longint The integers from -2147483648 to 2147483647

In addition to the above data types TURBO Pascal version 4.0 has the following data types available but in order to use them, you must have an 80X87 math coprocessor installed in your system;

single Real type with 7 significant digits
double Real type with 15 significant digits
extended Real type with 19 significant digits
comp The integers from about -10E18 to 10E18

A complete definition of the available types for each compiler can be found on pages 41 and 42 of the TURBO Pascal version 3.0 reference manual, and on pages 39 through 44 of the reference manual for version 4.0. It would be good to read these pages now for a good definition prior to learning how to define and use them in a program. Note that all of these will be used in example programs in this chapter.

The integers are by far the easiest to understand so we will start with a simple program that uses some integers in a very simple way. Load INTVAR into your TURBO system and

lets take a look at it.

OUR FIRST VARIABLES

Immediately following the program statement is another reserved word, "var". This reserved word is used to define a variable before it can be used anywhere in the program.

CHAPTER 3 - The simple Pascal data types

There is an unbroken rule of Pascal that states "Nothing can be used until it is defined." The compiler will complain by indicating a compilation error if you try to use a variable without properly defining it. It seems a bit bothersome to have to define every variable prior to its use, but this rule will catch many spelling errors of variables before they cause trouble. Some other languages will simply define a new variable with the new name and go merrily on its way producing some well formatted garbage for you.

Notice that there is only one "var", but it is used to define three different variables, Count, X, and Y. Once a var is recognized, the compiler will continue to recognize variable definitions line after line until it finds another reserved word. It would be permissible to put a var on the second line also but it is not necessary. It would also be permissible to put all three variables on one line but your particular programming style will dictate where you put the three variables. Following the colon on each line is the word "integer" which is a standard identifier which is different from a reserved word. An identifier is predefined like a reserved word but you can redefine it thereby losing its original purpose and meaning. For now and for a long time, don't do that. Page 38 contains a list of standard identifiers in TURBO Pascal 3.0. There is no corresponding list in the reference manual for TURBO Pascal 4.0.

OUR FIRST ARITHMETIC

Now that we have three variables defined as integer type variables, we are free to use them in a program in any way we desire as long as we use them properly. If we tried to assign a real value to X, the compiler will generate an error, once again preventing a garbage output. Observe the start of the main body of the program. There are three statements assigning values to X, Y, and Count. A fine point of mathematics would state that Count is only equal to the value of X+Y until one of them was modified, therefore the equal sign used in so many other languages is not used here. The sign := is used, and can be read as "is replaced by the value of", when reading a listing to preserve the mathematical purity of Pascal. Another quicker way is to use the word "gets". Thus X := X + 1 would be read "X gets the value of X plus 1". We will see later that the simple equal sign is reserved for use in a different manner.

The first three statements give X the value of 12, Y the value of 13, and Count the value of 12+13 or 25. We need to get those values out of the computer, so we need another extension to the Writeln statement. The first part of the data within the parentheses should be familiar to you

now, but the second part is new. Multiple outputs can be handled within one Writeln if the fields are separated by a comma. To output a variable, simply write the variable's name in the output field. The number following the variable in each case is the number of output columns to be used by the output data. This number is optional and can be omitted allowing the system to use as many columns as it needs. For

purposes of illustration, they have all been assigned different numbers of columns. At this point, you can compile and run INTVAR and examine its output.

To illustrate the various ways to output data, load INTVAR2 and observe that even though the output is identical, it is output in a completely different manner. Observe especially that a Writeln all by itself simply moves the cursor to the beginning of a new line on the video monitor.

Compile and run this program and observe its output.

NOW LET'S USE LOTS OF VARIABLES

Load ALLVAR to observe a short program using all 5 of the basic data types. The variables are simply assigned values and the values are printed. A complete and detailed description of the options available in the Write statement is given in the TURBO reference manual version 3.0 on pages 111 through 113, and on pages 500 through 502 for version 4.0. It would be to your advantage to read this section at this time since very little explanation will be given about Write statements from this point on. We will discuss the method by which we can write to disk files or other output devices when the time comes.

Back to the basic types. Pascal does lots of cross checking for obvious errors. It is illegal to assign the value of any variable with a value that is of the wrong type or outside the allowable range of that variable. There are routines to convert from one system to another when that is necessary. Suppose, for example, that you wished to use the value of an integer in a calculation of real numbers. That is possible by first converting the integer into a real number of the same value and using the new real type variable in the desired calculations. The new real type variable must of course be defined in a var statement as a real type variable before it can be used. Details of how to do the conversion will be given later.

Since we have some variables defined, it would be nice to use the properties of computers for which they are famous, namely some mathematics. Two programs are available

for your observation to illustrate the various kinds of math available, REALMATH using real variables, and INTMATH using integer variables. You can edit, compile, and run these on your own with no comment from me except the comments embedded into the source files. Chapter 6 on pages 51 to 54 of your version 3.0 TURBO reference manual completely defines the simple mathematics available. The corresponding list for version 4.0 is found in chapter 3 on pages 46 through 49.

A byte type variable is used just like an integer variable but with a much smaller value. Only one byte of computer memory is used for each variable defined as a byte type variable, but 2 are used for each integer type variable.

BOOLEAN VARIABLES

Lets take a look at the boolean variable which is only allowed to take on two different values, TRUE or FALSE. This variable is used for loop controls, end of file indicators or any other TRUE or FALSE conditions in the program. Variables can be compared to determine a boolean value. Following is a complete list of the relational operators available with Pascal.

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

These operators can be used to compare any of the simple types of data including integer, char, byte, and real

type variables or constants. An illustration is the best way to learn about the boolean variable so load BOOLMATH and observe it.

In BOOLMATH we define a few boolean variables and two integer type variables for use in the program and begin by assigning values to the two integer variables. The expression "Junk = Who" in line 14 is actually a boolean operation that is not true since the value of Junk is not equal to the value of Who. The result is therefore FALSE and that value is assigned to the boolean variable A. The boolean variable B is assigned the value of TRUE because the expression "Junk = (Who - 1)" is true. The boolean variables C and D are likewise assigned some values in a manner that should not need any comment. After assigning a

value to the variable with the big name, the values are all printed out.

WHERE DO WE USE THE BOOLEAN VARIABLES?

We will find many uses for the boolean type variable when we study the loops and conditional statements soon, but until then we can only learn what they are. Often, in a conditional statement, you will want to do something if either of two things are true, in which case you will use the reserved word "and" with two boolean expressions. If either of the two are true, the result will be true. Line 29 is an example of this. If the boolean variables B, C, and D, are all true, then the result will be true and A will be assigned the value of TRUE. If any one of them is false, the result will be false and A will be assigned the value of

FALSE.

In Line 31, where the "or" operator is illustrated, if any of the three boolean variables is true, the result will be true, and if all three are false, the result will be false. Another boolean operator is the "not" which is illustrated in line 30. Examine line 33 which says the result is true only if the variable Junk is one less than Who, or if Junk is equal to Who.

Compile and run this program, then add some additional printout to see if the boolean variables change the way you think they should in the last few statements.

LETS LOOK AT THE CHAR TYPE VARIABLE

A char type variable is a very useful variable, but usually not when used alone. It is very powerful when used in an array or some other user defined data structure which is beyond the scope of this chapter. A very simple program, CHARDEMO is included to give you an idea of how a char type variable can be used. Study then compile and run CHARDEMO for a very brief idea of what the char type variable is used for.

Examine the sample program CONVERT for several examples of converting data from one simple variable to another. The program is self explanatory.

THIS IS FOR TURBO PASCAL 4.0 USERS

If you are using TURBO Pascal version 3.0, you are finished with this chapter because the data types illustrated in the last two programs are not available with that compiler.

If you are using TURBO Pascal 4.0, display the program NEWINT4 for an example of using the extended integer types available with that compiler. Four variables are defined and values assigned to each, then the results are displayed. When you compile and run the program, you will see that the variable Big_int can indeed handle a rather large number.

It must be pointed out that the calculation in lines 13 and 21 result in a different answer even though they appear to be calculating the same thing. An explanation is in order. The quantity named MaxInt used in lines 10 and 13 is a constant built into the system that represents the largest value that an integer type variable can store. On the first page of this chapter we defined that as 32767 and when running the program you will find that Index displays that value as it should. The constant MaxInt has a type that is of a universal_integer type as do all of the numeric constants in line 13. The result then is calculated to the number of significant digits dictated by the left hand side of the assignment statement which is of type longint resulting in a very large number.

When we get to line 21, however, the variable Index is of type integer so the calculations are done as though the constants were of type integer also which causes some of the more significant digits to be truncated. The truncated result is converted to type longint and assigned to the variable Big_int and the truncated value is displayed by line 22.

After that discussion it should be apparent to you that it is important what types you use for your variables. It must be emphasized that it would not be wise to use all large type variables because they use more storage space and slow down calculations. Experience will dictate the proper data types to use for each application.

NOW FOR THE NEW REAL TYPES

If you are using TURBO Pascal 4.0, display the program NEWREAL4 for an example using the new "real" types available with version 4.0. Note that you must have an 80X87 math coprocessor installed to compile and run this program.

There is a note given in the file to aid you in selecting it for use.

This program should be self explanatory so nothing will be said except that when you run it you can observe the relative accuracy of each of the variable types. Once

again, you should keep in mind that use of the larger "real" types costs you a bit in storage space and run-time speed.

PROGRAMMING EXERCISE

1. Write a program containing several variable definitions and do some math on them, printing out the results.

CHAPTER 4 - The Pascal loops and control structures

Every program we have examined to this point has been a simple one pass through with no statements being repeated. As in all other languages, Pascal has extensive capabilities to do looping and conditional branching. We will look at these now.

THE FOR LOOP

We will start with what may be the easiest structure to understand, the "for" loop. This is used to repeat a single Pascal statement any number of times we desire. Load LOOPDEMO and we will discuss the loops presented there.

The first example is the simplest and is simply a

repeat of a Writeln 7 times. We have three new reserved words, "for", "to", and "do" which are used as shown. Any simple variable of type integer, byte, or char can be used for the loop index and it must be defined in a var statement. Following the "do" reserved word is any single Pascal statement that will be repeated the specified number of times. Note that the loop is an incrementing loop but substitution of "downto" for "to" will make it a decrementing loop as is illustrated in the last example in this program. It should be pointed out that the loop control variable can only be incremented or decremented by 1 each time through the loop in Pascal.

A COMPOUND PASCAL STATEMENT

The second example contains our first compound Pascal statement. It was mentioned in Chapter 1 that the begin end pair of reserved words could be used to mark the limits of a compound statement. In this case, the single statement starting with the "begin" at the end of line 17 and extending through and including the end statement in line 21 is the single Pascal statement that will be executed 10 times. A second variable Total has been introduced to simply add another operation to the loop. Any valid Pascal operation can be performed within the "begin end" pair, including another for loop, resulting in nested loops to whatever depth you desire.

The third example shows how the char type variable could be used in a for loop. Pascal requires that the loop variable, the starting point, and the ending point all be of the same type or it will generate an error message. In addition, it must be a variable of type integer, byte, or char. The starting point and ending point can be constants or expressions of arbitrary complexity.

The fourth example is a decrementing loop as mentioned earlier. It uses the reserved word "downto".

THE IF STATEMENT

Pascal has two conditional branching capabilities, the "if" and the "case". We will look at one of them now, the if statement. Load IFDEMO for an onscreen look at the "if then" pair of reserved words. Any condition that can be reduced to a boolean answer is put between the "if then" pair of words. If the resulting expression resolves to TRUE, then the following single Pascal statement is executed, and if it resolves to FALSE, then the following single statement is skipped over. Of course, you can probably guess that the single statement can be replaced with a compound statement bracketed with a "begin end" pair and you are correct. Study example 1 and you will see that the line will always be printed in this particular fragment because Three is equal to One + Two. It is very difficult to come up with a good example without combining some of the other control structures but we will do so in the next file.

The second example in lines 14 through 19, is similar to the first but has the single statement replaced with a compound statement and should be easy to understand.

The third example in lines 21 through 24, contains a new reserved word, "else". When the if condition is FALSE, the single statement is skipped and if a semicolon is encountered, the if clause is totally complete. If instead of a semicolon, the reserved word "else" is encountered, then the single Pascal statement following else is executed. One and only one of the two statements will be executed every time the if statement is encountered in the program. Examination of the third example should clear this up in your mind.

Notice that the Pascal compiler is looking for either a semicolon to end the if, or the reserved word "else" to continue the logic. It is therefore not legal to use a semicolon immediately preceding the reserved word "else". You will get a compiler error if you do so.

THE IF-THEN-ELSE block

Put on your thinking cap because the next principle is difficult to grasp at first but will suddenly clear up and be one of the most useful facts of Pascal programming. Since the entire "if then else" block of code is itself a single Pascal statement by definition, it can be used anywhere that an executable statement is legal without begin

end separators. This is shown in the fourth example of the IFDEMO Pascal example program. Lines 27 through 30 comprise a single Pascal statement, and lines 32 through 35 comprise another. The if statement begun in line 26 therefore has a single statement in each of its branches.

The "if then else" construct is one of the most used, most useful, and therefore most important aspects of Pascal. For this reason you should become very familiar with it.

Try changing some of the conditions in the example program to see if you can get it to print when you expect it to for your own practice. When you are ready, we will go on to a program with loops and conditional statements combined and working together.

LOOPS AND IFS TOGETHER

Load LOOPIF and observe it for a few minutes. It contains most of what you have studied so far and should be understandable to you at this point. It contains a loop (lines 7 & 17) with two if statements within it (lines 8 & 9 and lines 10 through 16), and another loop (lines 11 through 15) within one of the if statements.

You should make careful note of the formatting used here. The "begin" is at the end of the line which starts the control and the "end" is lined up under the control word such that it is very clear which control word it is associated with. You will develop your own clear method of formatting your code in time but until then it is suggested that you follow this example.

An easily made error should be pointed out at this time. If an extraneous semicolon were put at the end of the if statement in line 8, the code following the statement would always be executed because the "null" statement (the nothing statement between the "then" and the semicolon) would be the conditional statement. The compiler would not generate an error and you would get no warning. Add a semicolon at the end of line 8 to see the error.

FINALLY, A MEANINGFUL PROGRAM

Load TEMPCONV and study its structure. Notice the header block that defines the program and gives a very brief explanation of what the program does. This program should pose no problem to you in understanding what it does since it is so clearly documented. Run it and you will have a list of Centigrade to Fahrenheit temperature conversions with a few added notes.

CHAPTER 4 - The Pascal loops and control structures

Load, examine, and run DUMBCONV for a good example of poor variable naming. The structure of the program is identical to the last program and when you run it, you will see that it is identical in output, but compared to the last

program, it is difficult to understand what it does by studying the listing. This program, like the last should be easily understood by you, so we will go on to our next Pascal control structure.

THE REPEAT UNTIL LOOP

The next two Pascal constructs are very similar because they are both indefinite loops (indefinite because they are not executed a fixed number of times). One of the loops is evaluated at the top and the other at the bottom. It will probably be easier to start with the "repeat" "until" construct which is the loop that is evaluated at the bottom.

Retrieve the file REPEATLP to see an example of a repeat loop. Two more reserved words are defined here, namely "repeat" and "until". This rather simple construct simply repeats all statements between the two reserved words until the boolean expression following the "until" is found to be TRUE. This is the only expression I know of that operates on a range of statements rather than a single statement and begin end delimiters are not required.

A word of caution is in order here. Since the loop is executed until some condition becomes TRUE, it is possible that the condition will never be TRUE and the loop will never terminate. It is up to you, the programmer, to insure that the loop will eventually terminate.

Compile and run REPEATLP to observe the output.

THE WHILE LOOP

The file WHILELP contains an example of another new construct, the "while" loop. This uses the "while" "do" reserved words and will execute one Pascal statement (or one compound statement bounded with begin and end) continuously until the boolean expression between the two words becomes FALSE.

This loop is also indeterminate and could, like the repeat until loop, never terminate. You should therefore exercise care in using it.

There are two basic differences in the last two loops. The repeat until loop is evaluated at the bottom of the loop

and must therefore always go through the loop at least one time. The while loop is evaluated at the top and may not go through even once. This gives you flexibility when choosing the loop to do the job at hand.

Compile, run, and examine the output from the example program WHILELP.

THE CASE STATEMENT

The final control structure introduces one more reserved word, "case". The case construct actually should be included with the if statement since it is a conditional execution statement, but I chose to save it for last because it is rather unusual and will probably be used less than the others we have discussed in this chapter.

The case statement is used to select one of many possible simple Pascal statements to execute based on the value of a simple variable. Load the file CASEDEMO and observe the program for an example of a case statement. The variable between the "case" and "of" reserved words in line 9 is the variable used to make the selection. Following that, the various selections are listed as a possible value or range, followed by a colon, a single Pascal statement, and a semicolon for each selector. Following the list of selections, an "else" can be added to cover the possibility that none of the selections were executed. Finally, an end statement is used to terminate the case construct. Note that this is one of the few places in Pascal that an end is used without a corresponding begin.

The example file uses Count for a variable and prints

the numbers one through five in text form, and declares that numbers outside this range are not in the allowable list. The program should be self explanatory beyond that point. Be sure to compile and run this example program.

Load and display the sample program BIGCASE for another example of a case statement with a few more added features. This program uses the identical structure as the previous program but in line 11 a range is used as the selector so that if the value of Count is 7, 8, or 9 this selection will be made. In line 12, three different listed values will cause selection of this part of the code. Of greater importance are the compound statements used in some of the selections. If the variable Count has the value of 2, 4, or 6, a compound statement will be executed and if the value is 3, a for loop is executed. If the value is 1, an if statement is executed which will cause a compound statement to be executed. In this case the if statement will always

be executed because TRUE will always be true, but any Boolean expression could be used in the expression.

Be sure to compile and run this program, then study the output until you understand the result thoroughly.

This brings us to the end of chapter 4 and you now have enough information to write essentially any program desired in Pascal. You would find that you would have a few difficulties if you attempted to try to write a very big program without the topics coming up in the next few chapters. The additional topics will greatly add to the flexibility of Pascal and will greatly ease programming in it.

PROGRAMMING EXERCISES

1. Write a program that lists the numbers from 1 to 12 and writes a special message beside the number representing your month of birth.

2. Write a program that lists all of the numbers from 1 to 12 except for the numbers 2 and 9.

CHAPTER 5 - The Pascal procedures and functions

In order to properly define procedures and functions we need to lay some groundwork in the form of a few definitions. These are important concepts, so pay close attention.

Program Heading - This is the easiest part since it is only one line, at least it has been in all of our programs up to this point. It is simply the "program" line, and it never needs to be any more involved than it has been up to this point in TURBO Pascal.

Declaration Part - This is the part of the Pascal source code in which all constants, variables, and user defined auxiliary operations are defined. In some of the programs we have examined, there have been one or more var declarations. These are the only components of the declaration part we have used to this point. There are actually five components in the declaration part, and the procedures and functions are the fifth part. We will cover the others in the next chapter.

Statement Part - This is the last part of any Pascal program, and it is what we have been calling the main program. It is one compound statement bracketed with the reserved words "begin" and "end".

It is very important that you grasp the above definitions because we will be referring to them constantly during this chapter, and throughout the remainder of this tutorial. With that introduction, lets go on to our first Pascal program with a procedure in it, in fact, it will have three procedures.

THE FIRST PROCEDURES

Load PROCED1 as your first example file with a procedure and display it on your monitor. You will notice that it doesn't look like anything you have seen up to this point because it has procedures in it. Lets go back to our definitions from above. The first line is the Program Heading which should pose no difficulty. The Declaration Part begins with the var statement in line 4 and continues down through and including all three procedures ending in line 19. Lines 21 through 26 constitute the Statement Part. It may seem strange that what appears to be executable Pascal statements, and indeed they are executable statements, are contained in the Declaration Part rather than the Statement Part. This is because of the Pascal

definition and it will make sense when we have completed our study of procedures and functions.

Continuing to examine PROCED1, we will make note of the program itself, which is the Statement Part. The program, due to the nature of Pascal and the carefully chosen procedure names, clearly tells us what it will do. It will write a header, eight messages, and an ending. The only problem we are faced with is, how will it write these messages? This is where the Declaration Part is called upon to define these operations in detail. The Declaration Part contains the three procedures which will completely define what is to be done by the procedure calls in the main program.

It should be clear to you that the definitions of the procedures should be in the Definition Part of the program because that is exactly what they do. In the case of a var, a variable is defined for later use by the main program, and in the case of a procedure, the procedure itself is defined for later use by the main program.

Lets arbitrarily pick one of the procedures, the first, and examine it in detail. The first executable statement we come to in the main program is line 22 and says simply, Write_A_Header, followed by the usual end of statement, the semicolon. This is a simple procedure call. When the compiler finds this statement it goes looking for a predefined procedure of that name which it can execute. If it finds one in the Declaration Part of the program, it will execute that procedure. If it doesn't find a user defined procedure, it will search the Pascal library for a system defined procedure and execute it. The Write and Writeln

statements are system procedures, and you have already been using them quite a bit, so procedures are not completely new to you. If it doesn't find the procedure defined in either place, it will generate an error message.

HOW TO CALL A PROCEDURE

To call a procedure, we simply need to state its name. To define a simple procedure, we use the reserved word "procedure" followed by its calling name, with a semicolon as a terminator. Following the Procedure Heading, there is the Declaration Part of the procedure followed by a body which is nothing more than a compound statement bracketed by the reserved words "begin" and "end". This is identical to the Statement Part of the main program except that the procedure ends with a semicolon instead of a period. Any valid Pascal statements can be put between the begin and

end, and in fact, there is no difference in what can be put in a procedure and what can be put in the main program.

The program we are examining would be no different if we would eliminate the first procedure completely and move the Writeln contained in it down to the Statement Part in place of Write_A_Header. If that is not clear, go back and reread the last two paragraphs until it is.

Lines 23 and 24 will cause the procedure named Write_A_Message to be called 8 times, each time writing a line of output. Suffice it to say at this time that the value of the variable Count, as defined here, is available globally, meaning anywhere in the entire Pascal program. We will define the scope of variables shortly. Finally, the

last procedure call is made, causing the ending message to be displayed, and the program execution is complete.

Having examined your first Pascal procedures, there is a fine point that is obvious but could be easily overlooked. We mentioned the unbroken rule of Pascal in an earlier chapter and it must be followed here too. "Nothing can be used in Pascal until it has been defined". The procedures must all be defined ahead of any calls to them, once again emphasizing the fact that they are part of the Declaration Part of the program, not the Statement Part.

Compile and run PROCED1 to verify that it does what you expect it to do.

MORE PROCEDURE CALLS

Assuming you have run PROCED1 successfully and understand its output, lets go on to PROCED2 and examine it. In this program we will see how to call a procedure and take along some data for use within the procedure. To begin with, notice that there are three procedure calls in the Statement Part of the program and each has an additional term not contained in the calls in the last program, namely the variable name Index within brackets. This is Pascals way of taking a variable parameter to the procedure when it is called.

You will notice that the variable Index is defined as an integer variable in the very top of the Declaration Part. Since we are taking an integer type variable along when we visit the procedure Print_Data_Out, it had better be expecting an integer variable as input or we will have a type mismatch. In fact, observing the procedure heading itself in line 7, indicates that it is indeed expecting an integer variable but it prefers to call the variable Puppy

inside of the procedure. Calling it something different poses no problem as long as the main program doesn't try to call its variable Puppy, and the procedure doesn't try to use the name Index. Both are actually referring to the same piece of data but they simply wish to refer to it by different names.

Observe that the next procedure is called with Index as a parameter and the procedure prefers to call it by the name Cat. In both cases, the procedures simply print out the parameter passed to it, and each then try to modify the value passed to it before passing it back. We will see that one will be successful and the other will not.

We are in a loop in which Count is incremented from 1 to 3 and Pascal does not allow us to modify the loop variable so we make a copy of the value in line 21 and call it Index. We can then modify Index in the main program if we desire.

CALL BY VALUE

In line 7, the procedure heading does not contain a var in front of the passed parameter and therefore the parameter passing is only one way because of the way Pascal is defined. Without the reserved word var in front of the variable Puppy, the system makes a copy of Index, and passes the copy to the procedure which can do anything with it, using its new name, Puppy, but when control returns to the main program, the original value of Index is still there. The copy of Index named Puppy is modified in the procedure, but the original variable Index remains unchanged. So you can think of the passed parameter without the var as one way parameter passing. This is a "call by value" because only the value of the variable is passed to the procedure.

CALL BY REFERENCE

In line 13, the second procedure has the reserved word "var" in front of its desired name for the variable, namely Cat, so it can not only receive the variable, it can modify it, and return the modified value to the main program. A copy is not made, but the original variable named Index is

actually passed to this procedure and the procedure can modify it, therefore communicating with the main program. The name Catin the procedure is actually another name for the variable named Index in the main program. A passed parameter with a var in front of it is therefore a two way situation. This is a "call by reference" since the reference to the original variable is passed to the procedure.

CHAPTER 5 - The Pascal procedures and functions

SOME NEW TERMINOLOGY

The parameter name in the calling program is referred to as the actual parameter, and the parameter name in the procedure is referred to as the formal parameter. In the last example then, the actual parameter is named Index and the formal parameter in the procedure is named Cat. It should be pointed out that it is called a formal parameter whether it is a "call by reference" or a "call by value". This terminology is used in many other programming languages, not only in Pascal.

When you run this program, you will find that the first procedure is unable to return the value of 12 back to the main program, but the second procedure does in fact succeed in returning its value of 35 to the main program. Spend as much time as you like studying this program until you fully understand it. It should be noted that as many parameters as desired can be passed to and from a procedure by simply making a list separated by commas in the calls, and separated by semicolons in the procedure. This will be illustrated in the next example program.

Compile and run PROCED2 and study the output. You

should be able to comprehend all of the output. If it is not clear, reread the last few paragraphs.

For your own enlightenment, examine PROCED3 for an example of a procedure call with more than one variable in the call. Normally, you would group the three input values together to make the program more readable, but for purposes of illustration, they are separated. Observe that the variable Fruit is a two way variable because it is the 3rd variable in the actual parameter list and corresponds to the 3rd formal parameter in the procedure header.

Compile and run PROCED3 to see that it does what you expect it to do based on the above explanation.

"CALL BY REFERENCE" OR "CALL BY VALUE"?

It may seem to you that it would be a good idea to simply put the word var in front of every formal parameter in every procedure header to gain maximum flexibility, but using all "call by references" could actually limit your flexibility. There are two reasons to use "call by value" variables when you can. First is simply to shield some data from being corrupted by the procedure. This is becoming a very important topic is Software Engineering known as "information hiding" and is the primary basis behind Object

Oriented Programming which is far beyond the scope of this tutorial.

Secondly is the ability to use a constant in the procedure call. Modify line 17 of PROCED3 as follows;

```
Add_The_Fruit(12,Orange,Fruit,Pear);
```

and compile and run the program. Since Value1 is a "call by value", the constant 12 can be used and the program will compile and run. However, if you change line 17 to;

```
Add_The_Fruit(Apple,Orange,32,Pear);
```

you will find that it will not compile because Total is a "call by reference" and the system must be able to return a value for the formal parameter Total. It cannot do this because 32 is a constant, not a variable.

The prior discussion should indicate to you that both "call by value" and "call by reference" have a useful place in Pascal programming and it is up to you to decide which you should use.

When you are satisfied with the present illustration, we will go on to study the scope of variables using PROCED4.

A MULTIPLY DEFINED VARIABLE

If you will examine PROCED4, you will notice that the variable Count is defined twice, once in the main program var block and once in the var block contained within the procedure named Print_Some_Data. This is perfectly legal and is within the Pascal definition.

The variable Index is defined only in the main program var block and is valid anywhere within the entire Pascal program, including the procedures. The variable Count is also defined in the main program var block and is valid anywhere within the entire Pascal program, except within the procedure where another variable is defined with the same name Count. The two variables with the same name are in fact, two completely different variables, one being available only outside of the procedure and the other being available only within the procedure. The variable More_Stuff is defined within the procedure, so it is invisible to the main program, since it is defined at a lower level than that of the main program.

Any variable is available at any point in the program following its definition but only at the level of definition

or below. This means that any procedure in the Declaration Part of a program can use any variable defined in the Declaration Part of the program provided that the definition occurs prior to the procedure. Any variable defined in a procedure cannot be used by the main program since the definition is at a lower level than the main program.

Be sure to compile and run PROCED4 before continuing on to the next example program.

PROCEDURES CALLING OTHER PROCEDURES

Load and examine PROCED5 to see an example of procedures that call other procedures. Keep in mind that, "Nothing can be used in Pascal until it has been previously defined", and the order of procedures will be clear in this example. Note that procedure Three calls procedure Two which in turn calls procedure One.

Compile and run PROCED5 and study the output until you understand why it outputs each line in the order that it does.

Now that you have a good working knowledge of procedures, we need to make another important point. Remember that any Pascal program is made up of three parts, the Program Heading, the Declaration Part, and the Statement Part. The Declaration Part is composed of five unique components, four of which we will discuss in detail in the next chapter, and the last component, which is composed of some number of procedures and functions. We will cover functions in the next example, so for now simply accept the fact that it is like a procedure. A procedure is also composed of three parts, a Procedure Heading, a Declaration

Part, and a Statement Part. A procedure, by definition, is therefore nothing more or less than another complete Pascal program embedded within the main program, and any number of procedures can be located in the Declaration Part of the main program. These procedures are all in a line, one right after another.

Since a procedure is defined like the main program, it would seem to be possible to embed another procedure within the Declaration Part of any procedure. This is perfectly valid and is often done, but remember that the embedded procedure can only be called by the procedure in which it is embedded, not by the main program. This is a form of information hiding which is becoming popular in modern software engineering.

CHAPTER 5 - The Pascal procedures and functions

The previous paragraph is probably a bit difficult to grasp. Don't worry about it too much now, as you become proficient as a Pascal programmer, you will very clearly see how that is used.

NOW LET'S LOOK AT A FUNCTION

Now to keep a promise, lets examine the program named FUNCTION to see what a function is and how to use it. In this very simple program, we have a function that simply multiplies the sum of two variables by 4 and returns the result. The major difference between a function and a procedure is that the function returns a single value and is called from within a mathematical expression, a Writeln command, or anywhere that it is valid to use a variable,

since it is really a variable itself. Observing the Function Heading of the function, in line 6, reveals the two input variables inside the parenthesis pair being defined as integer variables, and following the parenthesis is a colon and another "integer". The last "integer" is used to define the type of the variable being returned to the main program.

Any call to this function is actually replaced by an integer upon completion of the call. Therefore in line 14, the function is evaluated and the value returned is used in place of the function call. The result of the function is assigned to the variable Feet.

Note that a function always returns a value and it may return additional values if some of its parameters are defined as "call by reference". Be sure to compile and run this program.

NOW FOR THE MYSTERY OF RECURSION

One of the great mysteries of Pascal to many people, is the recursion of procedure calls. Simply defined, recursion is the ability of a procedure to call itself. Examine the Pascal example file RECURSON for an example of recursion. The main program is very simple, it sets the variable Count to the value 7 and calls the procedure Print_And_Decrement. The procedure prefers to refer to the variable by the name Index but that poses no problem for us because we understand that the name of the formal parameter can be any legal identifier. The procedure writes a line to the video display with the value of Index written within the line, and decrements the variable.

The if statement introduces the interesting part of this program. If the variable is greater than zero, and it is now 6, then the procedure Print_And_Decrement is called

once again. This might seem to create a problem except for the fact that this is perfectly legal in Pascal. Upon entering the procedure the second time, the value of Index is printed as 6, and it is once again decremented. Since it is now 5, the same procedure will be called again, and it will continue until the value of Index is reduced to zero when each procedure call will be completed one at a time and control will return to the main program.

ABOUT RECURSIVE PROCEDURES

This is really a stupid way to implement this particular program, but it is the simplest recursive program that can be written and therefore the easiest to understand. You will have occasional use for recursive procedures, so don't be afraid to try them. Remember that the recursive procedure must have some variable converging to something, or you will have an infinite loop.

Compile and run this program and observe the value decrementing as the recursion takes place.

THE FORWARD REFERENCE

Occasionally you will have a need to refer to a procedure before you can define it. In that case you will need a forward reference. The program FORWARD has an example of a forward reference in it. In this program, each one of the procedures calls the other, a form of recursion. This program, like the last, is a very stupid way to count from 7 to 0, but it is the simplest program possible with the forward reference.

The first procedure, Write_A_Line, has its header defined in exactly the same manner as any other procedure but instead of the normal procedure body, only the reserved word "forward" is given. This tells the compiler that the procedure will be defined later. The next procedure is defined as usual, then the body of Write_A_Line is given with only the reserved word "procedure" and the procedure name. The variable reference has been defined earlier. In this way, each of the procedure names are defined before they are called.

It would be possible, by using the forward reference in great numbers, to move the main program ahead of all procedure definitions and have the program structured like some other languages. This style of programming would be perfectly legal as far as the compiler is concerned, but the resulting program would be very nonstandard and confusing. You would do well to stick with conventional Pascal

formatting techniques and use the forward reference sparingly.

Be sure you compile and run this program.

PROGRAMMING EXERCISES

1. Write a program to write your name, address, and phone number with each Writeln in a different procedure.

2. Add a statement to the procedure in RECURSON to display the value of "Index" after the call to itself so you can see the value increasing as the recurring calls are returned to the next higher level.

3. Rewrite TEMPCONV putting the centigrade to fahrenheit formulas in a function call.

CHAPTER 6 - Arrays, types, constants, and labels

ARRAYS

At the beginning of this tutorial we said that a computer program is composed of data and executable statements to do something with that data. Having covered nearly all of the programming statements, we must now go back and fill in some gaps in our data definition and look at the array in particular.

One of the most useful Pascal data structures is the array, which is, in the simplest terms, a group of 2 or more identical terms, all having the same type. Lets go directly to an example to see what an array looks like. Display the Pascal program ARRAYS and notice line 5 starting with the

word Automobiles. The variable Automobiles is defined as an integer variable but in addition, it is defined to have twelve different integer variables, namely Automobile[1], Automobile[2], Automobile[3], .. Automobile[12].

The square braces are used in Pascal to denote a subscript for an array variable. The array definition given in line 5 is the standard definition for an array, namely a variable name followed by a colon and the reserved word "array", with the range of the array given in square brackets followed by another reserved word "of" and finally the type of variable for each element of the array.

In using the elements of the array in a program, each of the elements of the array are required to be used in exactly the same manner as any simple variable having the same type. Each time one of the variables is used, it must have the subscript since the subscript is now part of the variable name. The subscript moreover, must be of the type used in the definition and it must be within the range defined or it will be listed as an error.

Now consider the program itself. As Index is varied from 1 to 12, the range of the variable Automobile, the 12 variables are set to the series of values 11 to 22. Any integer values could be used, this was only a convenient way to set the values to some well defined numbers. With the values stored, a header is now printed and the list of values contained in the array is printed. Note carefully that, although the subscripts are limited to 1 through 12, the values stored in each of the 12 variables are limited only by the range of integers, namely -32768 to 32767. Review this material and this program as long as needed to fully understand it, as it is very important.

Keep in mind that the array is actually composed of 12 different integer type variables that can be used in any way

that it is legal to use any other integer type variable. Compile and run this program.

DOUBLY INDEXED ARRAYS

After understanding the above, load the program ARRAYS2 to see the next level of complexity of arrays. You will see that Checkerboard is defined as an array from 1 to 8, but instead of it being a simple data type, it is itself another array from 1 to 8 of type integer. The variable Checkerboard is actually composed of 8 elements, each of which is 8 elements, leading to a total of 64 elements, each of which is a simple integer variable. This is called a doubly subscripted array and it can be envisioned in exactly the same manner as a real checker board, an 8 by 8 matrix. Another way to achieve the same end is to define the double array as in the next line of the program where Value is defined as a total of 64 elements.

To use either of the two variables in a program, we must add two subscripts to the variable name to tell the program which element of the 64 we desire to use. Examining the program will reveal two loops, one nested within the other, and both ranging in value from 1 to 8. The two loop indices can therefore be used as subscripts of the defined array variables. The variable Checkerboard is subscripted by both of the loop indices and each of the 64 variables is assigned a value as a function of the indices. The assigned value has no real meaning other than to illustrate to you how it is done. Since the value of Checkerboard is now available, it is used to define some values to be used for the variable Value in line 12 of the program.

After defining all of those variables, and you should understand that we have defined a total of 128 variables in the double loop, 64 of Checkerboard and 64 of Value, they can be printed out. The next section of the program does just that, by using another doubly nested loop, with a Write statement in the center. Each time we go through the center of the loop we tell it to print out one of the 64 variables in the Checkerboard matrix with the indices Index and Count defining which of the variables to write each time. Careful study of the loop should reveal its exact operation.

After printing out the matrix defined by the variable Checkerboard we still have the matrix defined by the variable Value intact (In fact, we still have all of Checkerboard available because we haven't changed any of it). Before printing out the matrix defined by Value, let's change a few of the elements just to see how it is done. The code in lines 24 to 26 simply change three of the

variables to illustrate that you can operate on all of the matrix in loops, or on any part of the matrix in simple assignment statements. Notice especially line 26, in which "Value[3,6]" (which was just set to the value of 3), is used as a subscript. This is perfectly legal since it is defined as a simple integer variable and is within the range of 1 to 8, which is the requirement for a subscript of the variable Value. The last part of the program simply prints out the 64 values of the variable Value in the same manner as above. Notice that when you run the program, the three values are in fact changed as expected.

ARRAYS ARE FLEXIBLE

A few more words about arrays before we go on. The arrays in the last program were both defined to be square, namely 8 by 8, but that choice was purely arbitrary. The subscripts were chosen to go from 1 to 8 but they could have been chosen to go from 101 to 108 or any other range needed to clearly define the problem at hand. And, as you may have guessed, you are not limited to a doubly subscripted matrix but you can define a variable with as many subscripts as you need to achieve your desired end. There is a practical limit to the number of subscripts because you can very

quickly use up all of your available memory with one large subscripted variable.

THE TYPE DEFINITION

Now that you understand arrays, lets look at a more convenient way to define them by examining the Pascal file TYPES. You will notice a new section at the beginning of the listing with the heading type. The word "type" is another reserved word which is used at the beginning of a section used to define "user-defined types". Beginning with the simple predefined types we studied earlier, we can build up as many new types as we need and they can be as complex as we desire. The six names (from Array_Def to Boat) in the type section are not variables, but are defined to be types and can be used in the same manner as can integer, byte, real, etc.

This is a very difficult concept, but a very important one. The Pascal compiler is very picky about the variable types you use in the program, doing lots of checking to insure that you do not use the wrong type anywhere in the program. Because it is picky, you could do very little without the ability to define new types when needed, and that is the reason Pascal gives you the ability to define new types to solve a particular problem.

Some of these types are used in the var declaration part of the program. Notice that since Airplane is an array of Dog_Food and Dog_Food is in turn an array of boolean, then Airplane defines a doubly subscripted array, each element being a boolean variable. This does not define any

variables, only a user defined type, which can be used in a var to define a matrix of boolean variables. This is in fact done in the definition of Puppies, which is an array composed of 72 (6 times 12) boolean variables. In the same manner, Stuff is composed of an array of 14 variables, each being an integer variable. The elements of the array are, Stuff[12], Stuff[13], .. Stuff[25]. Notice also that Stuff2 is also defined in exactly the same manner and is also composed of 14 variables.

Careful inspection will reveal that Kitties is a variable which has the same definition as Puppies. It would probably be poor programming practice to define them in different manners unless they were in fact totally disassociated. In this example program, it serves to illustrate some of the ways user-defined types can be defined.

Be sure to compile and run this program.

IS THE CONCEPT OF "TYPES" IMPORTANT?

If you spend the time to carefully select the types for the variables used in the program, the Pascal compiler will do some debugging for you since it is picky about the use of variables with different types. Any aid you can use to help find and remove errors from your program is useful and you should learn to take advantage of type checking. The type checking in Pascal is relatively weak compared to some other languages such as Modula-2 or Ada, but still very useful.

In a tiny program like this example, the value of the type declaration part cannot be appreciated, but in a large program with many variables, the type declaration can be used to great advantage. This will be illustrated later.

THE CONSTANT DECLARATION

Examining the Pascal example program CONSTANT will give us an example of a constant definition. The reserved word "const" is the beginning of the section that is used to define constants that can be used anyplace in the program as long as they are consistent with the required data typing limitations. In this example, Max_Size is defined as a constant with the value of 12. This is not a variable and cannot be changed in the program, but is still a very

valuable number. For the moment ignore the next two constant definitions. As we inspect the type declarations, we see two user-defined types, both of which are arrays of size 1 to 12 since Max_Size is defined as 12. Then when we get to the var declaration part, we find five different variables, all defined as arrays from 1 to 12 (some are type integer and some are type char). When we come to the program we find that it is one big loop which we go through 12 times because the loop is executed Max_Size times.

In the above definition, there seems to be no advantage to using the constant, and there is none, until you find that for some reason you wish to increase the range of all arrays from 12 to 18. In order to do so, you only need to redefine the value of the constant, recompile, and the whole job is done. Without the constant definition, you would have had to change all type declarations and the upper limit of the loop in the program. Of course that would not be too bad in the small example program, but could be a real mess in a 2000 line program, especially if you missed changing one of the 12's to an 18. That would be a good example of data in and garbage out. This program should give you a good idea of what the constant can be used for, and as you develop good programming techniques, you will use the constant declaration to your advantage.

THE TURBO PASCAL TYPED CONSTANT

We skipped over the second and third constant declaration for a very good reason, they are not constant declarations. TURBO Pascal has defined, as an extension, the "typed constant". Using the syntax shown, Index_Start is defined as an integer type variable and is initialized to

the value of 49. This is a true variable and can be used as such in the program. The same effect can be achieved by simply defining Index_Start as an integer type variable in the var declaration part and setting it to the value of 49 in the program itself. Since it does not really fit the definition of a constant, it's use is discouraged until you gain experience as a Pascal programmer. Until then it will probably only be confusing to you. In like manner, Check_It_Out is a boolean type variable initialized to the value TRUE. It is not a constant.

The typed constants defined in the last paragraph have one additional characteristic, they are initialized only once, when the program is loaded. Even when used in a procedure or function, they are only initialized when the program is loaded, not upon each call to the procedure or function. Don't worry too much about this at this point,

when you gain experience with Pascal, you will be able to use this information very effectively.

THE LABEL DECLARATION

Finally, the example program LABELS will illustrate the use of labels. In the Pascal definition, a label is a number from 0 to 9999 that is used to define a point in the program to which you wish to jump. All labels must be defined in the label definition part of the program before they can be used. Then a new reserved word "goto" is used to jump to that point in the program. The best way to see how the goto is used with labels is to examine the program before you.

TURBO Pascal has an extension for labels. Any valid identifier, such as used for variables, can be used as a label in addition to the values from 0 to 9999. These are illustrated in the example program.

When you compile and run this program, the output will look a little better than the program did.

THE PACKED ARRAY

When Pascal was first defined in 1971, many of the computers in use at that time used very large words, 60 bits being a typical word size. Memory was very expensive, so large memories were not too common. A Pascal program that used arrays was inefficient because only one variable was stored in each word. Most of the bits in each word were totally wasted, so the packed array was defined in which several variables were stored in each word. This saved storage space but took extra time to unpack each word to use the data. The programmer was given a choice of using a fast scheme that wasted memory, the array, or a slower scheme that used memory more efficiently, the packed array.

The modern microcomputer has the best of both schemes, a short word, usually 16 bits, and a large memory. The packed array is therefore not even implemented in many compilers and will be ignored during compilation. The packed array is specifically ignored by either TURBO Pascal compiler.

ONE MORE TURBO PASCAL EXTENSION

Standard Pascal, as defined by Nicklaus Wirth, requires that the various fields in the definition part of the program come in a specific order and each must appear only once. The specific order is, label, const, type, var, and finally the procedures and functions. Of course, if any are

not needed, they are simply omitted. This is a rather rigid requirement but it was required by the pure Pascal definition probably to teach good programming techniques to beginning students.

All versions of TURBO Pascal are not nearly as rigid as the standard Pascal requirement. You are permitted to use the fields in any order and as often as you wish provided that you define everything before you use it, which is the unbroken rule of Pascal. It sometimes makes sense to define a few variables immediately after their types are defined to keep them near their type definitions, then define a few more types with the variables that are associated with them also. TURBO Pascal gives you this extra flexibility that can be used to your advantage.

PROGRAMMING EXERCISES

1. Write a program to store the integers 201 to 212 in an array then display them on the monitor.

2. Write a program to store a 10 by 10 array containing the products of the indices, therefore a multiplication table. Display the matrix on the video monitor.

3. Modify the program in 2 above to include a constant so that by simply changing the constant, the size of the matrix and the range of the table will be changed.

CHAPTER 7 - Strings and string procedures

PASCAL STRINGS

According to the Pascal definition, a string is simply an array of 2 of more characters of type char, and is contained in an array defined in a var declaration as a fixed length. Look at the example program STRARRAY. Notice that the strings are defined in the type declaration even though they could have been defined in the var part of the declaration. This is to begin getting you used to seeing the type declaration. The strings defined here are nothing more than arrays with char type variables.

A STRING IS A ARRAY OF CHAR

The interesting part of this file is the executable program. Notice that when the variable First_Name is assigned a value, the value assigned to it must contain exactly 10 characters or the compiler will generate an error. Try editing out a blank and you will get an invalid type error. Pascal is neat in allowing you to write out the values in the string array without specifically writing each character in a loop as can be seen in the Writeln statement. To combine the data, called concatenation, requires the use of the rather extensive looping and subscripting seen in the last part of the program. It would be even messier if we were to consider variable length fields which is nearly always the case in a real program.

Two things should be observed in this program. First, notice the fact that the string operations are truly array operations and will follow all of the characteristics discussed in the last chapter. Secondly, it is very obvious

that Pascal is rather weak when it comes to its handling of text type data. Pascal will handle text data, even though it may be difficult. This concerns the standard description of Pascal, we will see next that TURBO Pascal really shines here.

Compile and run STRARRAY and observe the output.

THE TURBO PASCAL STRING TYPE

Look at the example program STRINGS. You will see a much more concise program that actually does more. TURBO Pascal has, as an extension to standard Pascal, the string type of variable. It is used as shown, and the number in the square brackets in the var declaration is the maximum length of the string. In actual use in the program, the variable can be used as any length from zero characters up to the maximum given in the declaration. The variable First_Name, for example, actually has 11 locations of

storage for its data. The current length is stored in First_Name[0] and the data is stored in First_Name[1] through First_Name[10]. All data are stored as byte variables, including the size, so the length is therefore limited to a maximum of 255 characters.

STRINGS HAVE VARIABLE LENGTHS

Now look at the program itself. Even though the variable First_Name is defined as 10 characters long, it is perfectly legal to assign it a 4 character constant, with First_Name[0] automatically set to four by the system and the last six characters undefined and unneeded. When the

program is run the three variables are printed out all squeezed together indicating that the variables are indeed shorter than their full size as defined in the var declaration.

Using the string type is even easier when you desire to combine several fields into one as can be seen in the assignment to Full_Name. Notice that there are even two blanks, in the form of constant fields, inserted between the component parts of the full name. When it is written out, the full name is formatted neatly and is easy to read.

Compile and run STRINGS and observe the output.

WHAT IS IN A STRING TYPE VARIABLE?

The next example program named WHATSTRG, is intended to show you exactly what is in a string variable. This program is identical to the last program except for some added statements at the end. Notice the assignment to Total. The function Length is available in TURBO Pascal to return the current length of any string type variable. It returns a byte type variable with the value contained in the [0] position of the variable. We print out the number of characters in the string at this point, and then print out each character on a line by itself to illustrate that the TURBO Pascal string type variable is simply an array variable.

The TURBO Pascal reference manual has a full description of several more procedures and functions available in TURBO Pascal only. Refer to your TURBO Pascal version 3.0 reference manual in chapter 9, beginning on page 67, or if you are using TURBO Pascal version 4.0, you will find the string functions throughout chapter 27. The use of these should be clear after you grasp the material covered here.

CHAPTER 7 - Strings and string procedures

PROGRAMMING EXERCISES

1. Write a program in which you store your first, middle, and last names as variables, then display them one to a line. Concatenate the names with blanks between them and display your full name as a single variable.

CHAPTER 8 - Scalars, subranges, and sets.

PASCAL SCALARS

A scalar, also called an enumerated type, is a list of values which a variable of that type may assume. Look at the Pascal program ENTYPES for an example of some scalars. The first type declaration defines Days as being a type which can take on any one of seven values. Since, within the var declaration, Day is assigned the type of Days, then Day is a variable which can assume any one of seven different values. Moreover Day can be assigned the value Mon, or Tue, etc., which is considerably clearer than using 0 to represent Monday, 1 for Tuesday, etc. This makes the program easier to follow and understand.

Internally, Pascal does not actually assign the value Mon to the variable Day, but it uses an integer representation for each of the names. This is important to understand because you need to realize that you cannot print out Mon, Tue, etc., but can only use them for indexing control statements.

The second line of the type definition defines Time_Of_Day as another scalar which can have any of four different values, namely those listed. The variable Time can only be assigned one of four values since it is defined as the type Time_Of_Day. It should be clear that even though it can be assigned Morning, it cannot be assigned Morning_time or any other variant spelling of Morning, since it is simply another identifier which must have an exact spelling to be understood by the compiler.

Several real variables are defined to allow us to demonstrate the use of the scalar variables. After writing a header in lines 16 through 20, the real variables are initialized to some values that are probably not real life values, but will serve to illustrate the scalar variable.

A BIG SCALAR VARIABLE LOOP

The remainder of the program is one large loop being controlled by the variable Day as it goes through all of its values, one at a time. Note that the loop could have gone from Tue to Sat or whatever portion of the range desired, it does not have to go through all of the values of Day. Using Day as the case variable, the name of one of the days of the week is written out each time we go through the loop. Another loop controlled by Time is executed four times, once for each value of Time. The two case statements within the inner loop are used to calculate the total pay rate for each time period and each day. The data is formatted carefully

to make a nice looking table of pay rates as a function of Time and Day.

Take careful notice of the fact that the scalar variables never entered into the calculations, and they were not printed out. They were only used to control the flow of logic. It was much neater than trying to remember that Mon is represented by a 0, Tue is represented by a 1, etc. In fact, those numbers are used for the internal representation of the scalars but we can relax and let Pascal worry about the internal representation of our scalars.

Compile and run this program and observe the output.

LETS LOOK AT SOME SUBRANGES

Examine the program SUBRANGE for an example of subranges and some additional instruction on scalar variables. It may be expedient to define some variables that only cover a part of the full range as defined in a scalar type. Notice that Days is declared a scalar type as in the last program, and Work is declared a type with an even more restricted range. In the var declaration, Day is once again defined as the days of the week and can be assigned any of the days by the program. The variable Workday, however, is assigned the type Work, and can only be assigned the days Mon through Fri. If an attempt is made to assign Workday the value Sat, a run-time error will be generated. A carefully written program will never attempt that, and it would be an indication that something is wrong with either the program or the data. This is one of the advantages of Pascal over older languages and is a reason for the relatively strong type checking built into the language.

Further examination will reveal that Index is assigned the range of integers from 1 through 12. During execution of the program, if an attempt is made to assign Index any value outside of that range, a run time error will be generated. Suppose the variable Index was intended to refer to your employees, and you have only 12. If an attempt was made to refer to employee number 27, or employee number -8, there is clearly an error somewhere in the data and you would want to stop running the payroll to fix the problem. Pascal would have saved you a lot of grief.

SOME STATEMENTS WITH ERRORS IN THEM.

In order to have a program that would compile without errors, and yet show some errors, the section of the program in lines 16 through 27 is not really a part of the program

since it is within a comment area. This is a trick to remember when you are debugging a program, a troublesome part can be commented out until you are ready to include it in the rest. The errors are self explanatory and it would pay for you to spend enough time to understand each of the errors.

There are seven assignment statements as examples of subrange variable use in lines 29 through 35. Notice that the variable Day can always be assigned the value of either Workday or Weekend, but the reverse is not true because Day can assume values that would be illegal to assign to the others.

THREE VERY USEFUL FUNCTIONS

Lines 37 through 42 of the example program demonstrate the use of three very important functions when using scalars. The first is the Succ function that returns the value of the successor to the scalar used as an argument, the next value. If the argument is the last value, a run time error is generated. The next function is the Pred function that returns the predecessor to the argument of the function. Finally the Ord function which returns the ordinal value of the scalar.

All scalars have an internal representation starting at 0 and increasing by one until the end is reached. In our example program, Ord(Day) is 5 if Day has been assigned Sat, but Ord(Weekend) is 0 if Weekend has been assigned Sat. As you gain experience in programming with scalars and subranges, you will realize the value of these three new functions.

A few more thoughts about subranges are in order before we go on to another topic. A subrange is always defined by two predefined constants, and is always defined in an ascending order. A variable defined as a subrange type is actually a variable defined with a restricted range. Good

programming practice would dictate that subranges should be used as often as possible in order to prevent garbage data. There are actually very few variables ever used that cannot be restricted by some amount. The limits may give a hint at what the program is doing and can help in understanding the program operation. Subrange types can only be constructed using the simple types, integer, char, byte, or scalar.

Compile and run this program even though it has no output. Add some output statements to see what values some of the variables assume.

CHAPTER 8 - Scalars, subranges, and sets.

SETS

Now for a new topic, sets. Examining the example Pascal program SETS will reveal some sets. A scalar variable is defined first, in this case the scalar type named Goodies. A set is then defined with the reserved words "set of" followed by a predefined scalar type. Several variables are defined as sets of Treat, after which they can individually be assigned portions of the entire set.

Consider the variable Ice_Cream_Cone which has been defined as a set of type Treat. This variable is composed of as many elements of Goodies as we care to assign to it. In the program, we define it as being composed of Ice_Cream, and Cone. The set Ice_Cream_Cone is therefore composed of two elements, and it has no numerical or alphabetic value as most other variables have.

In lines 21 through 26, you will see 4 more delicious

deserts defined as sets of their components. Notice that the banana split is first defined as a range of terms, then another term is added to the group illustrating how you can add to a set. All five are combined in the set named Mixed, then Mixed is subtracted from the entire set of values to form the set of ingredients that are not used in any of the deserts. Each ingredient is then checked to see if it is "in" the set of unused ingredients, and printed out if it is. Note that "in" is another reserved word in Pascal. Running the program will reveal a list of unused elements.

In this example, better programming practice would have dictated defining a new variable, possibly called Remaining for the ingredients unused in line 32. It was desirable to illustrate that Mixed could be assigned a value based on subtracting itself from the entire set, so the poor variable name was used.

When you compile and run this program you will see that this example results in some nonsense results but hopefully it led your thinking toward the fact that sets can be used for inventory control, possibly a parts allocation scheme, or some other useful system.

SEARCHING WITH SETS

The Pascal program FINDCHRS is more useful than the last one. In it we start with a short sentence and search it for all lower case alphabetic letters and write a list of those used. Since we are using a portion of the complete range of char, we do not need to define a scalar before

defining the set, we can define the set using the range

'a'..'z'. The set Data_Set is assigned the value of no elements in the first statement of the program, and the print string, named Print_Group, is set to blank in the next. The variable Storage is assigned the sentence to search, and the search loop is begun. Each time through the loop, one of the characters is checked. It is either declared as a non-lower-case character, as a repeat of one already found, or as a new character to be added to the list.

You are left to decipher the details of the program, which should be no problem since there is nothing new here. Run the program and observe how the list grows with new letters as the sentence is scanned.

PROGRAMMING EXERCISE

1. Modify FINDCHRS to search for upper-case letters.

CHAPTER 9 - Records

We come to the grandaddy of all data structures in Pascal, the record. A record is composed of a number of variables any of which can be of any predefined data type, including other records. Rather than spend time trying to define a record in detail, lets go right to the first example program, SMALLREC. This is a program using nonsense data that will illustrate the use of a record.

A VERY SIMPLE RECORD

There is only one entry in the type declaration part of the program, namely the record identified by the name Description. The record is composed of three fields, the Year, Model, and Engine variables. Notice that the three fields are each of a different type, indicating that the record can be of mixed types. You have a complete example of the way a record is defined before you. It is composed of the identifier Description, the reserved word "record", the list of elements, and followed by end. This is one of the places in Pascal where an end is used without a corresponding begin. Notice that this only defines a type, it does not define any variables. That is done in the var declaration where the variable Cars is defined to have 10 complete records of the type Description. The variable Cars[1] has three components, Year, Model, and Engine, and any or all of these components can be used to store data pertaining to Cars[1].

When assigning data to the variable Cars[1], for example, there are actually three parts to the variable, so we use three assignment statements, one for each of the sub-fields. In order to assign values to the various sub-fields, the variable name is followed by the sub-field name with a separating period. The "var.sub_field" combination is a variable name.

Keep in mind that Cars[1] is a complete record containing three variables, and to assign or use one of the variables, you must designate which sub-field you are interested in. See the program where the three fields are assigned meaningless data for illustration. The Year field is assigned an integer number varying with the subscript, all Model fields are assigned the name Duesenburg, and all Engine variables are assigned the value V8. In order to further illustrate that there are actually 30 variables in use here, a few are changed at random in lines 20 through 24, being very careful to maintain the required types as defined in the type declaration part of the program. Finally, all ten composite variables, consisting of 30 actual variables in a logical grouping are printed out using the same "var.sub-field" notation described above.

CHAPTER 9 - Records

If the preceding description of a record is not clear in your mind, review it very carefully. It's a very important concept in Pascal, and you won't have a hope of a chance of understanding the next example until this one is clear.

Be sure to compile and run SMALLREC so you can study the output.

A SUPER RECORD

Examine the Pascal example file BIGREC for a very interesting record. First we have a constant defined. Ignore it for the moment, we will come back to it later. Within the type declaration we have three records defined, and upon close examination, you will notice that the first

two records are included as part of the definition of the third record. The record identified as Person, actually contains 9 variable definitions, three within the Full_Name record, three of its own, and three within the Date record. This is a type declaration and does not actually define any variables, that is done in the var part of the program.

The var part of the program defines some variables beginning with the array of Friend containing 50 (because of the constant definition in the const part) records of the user defined type, Person. Since the type Person defines 9 fields, we have now defined 9 times 50 = 450 separate and distinct variables, each with its own defined type. Remember that Pascal is picky about assigning data by the correct type. Each of the 450 separate variables has its own type associated with it, and the compiler will generate an error if you try to assign any of those variables the wrong type of data. Since Person is a type definition, it can be used to define more than one variable, and in fact it is used again to define three more records, Self, Mother, and Father. These three records are each composed of 9 variables, so we have 27 more variables which we can manipulate within the program. Finally we have the variable Index defined as a simple byte type variable.

HOW TO MANIPULATE ALL OF THAT DATA

In the program we begin by assigning data to all of the fields of Self in lines 31 through 43. Examining the first three statements of the main program, we see the construction we learned in the last example program being used, namely the period between descriptor fields. The main record is named Self, and we are interested in the first part of it, specifically the Name part of the Person record.

Since the Name part of the Person record is itself composed of three parts, we must designate which component of it we are interested in. Self.Name.First_Name is the complete description of the first name of Self and is used in the assignment statement in line 31 where it is assigned the name of "Charley". The next two fields are handled in the same way and are self explanatory.

WHAT IS THE WITH STATEMENT?

Continuing on to the fourth field, the City, there are only two levels required because City is not another record definition. The fourth field is therefore completely defined by Self.City. Notice the "with Self do" statement. This is a shorthand notation used with record definitions to simplify coding. From the begin in line 34 to the matching end in line 43, any variables within the Self record are used as though they had a "Self." in front of them. It greatly simplifies coding to be able to omit the leading identifier within the with section of code. You will see that City, State, and Zipcode are easily assigned values without further reference to the Self variable. When we get to the Day part of the birthday, we are back to three levels and the complete definition is Self.Birthday.Day but once again, the "Self." part is taken care of automatically because we are still within the "with Self do" area.

To illustrate the with statement further, another is introduced in line 39, "with Birthday do", and an area is defined by the begin end pair which extends from line 39 through line 42. Within this area both leading identifiers are handled automatically to simplify coding, and Month is equivalent to writing Self.Birthday.Month if both with statements were removed.

HOW FAR DOWN CAN YOU NEST THE WITH STATEMENT?

You may be wondering how many levels of nesting are allowed in record definitions. There doesn't appear to be a limit according to the Pascal definition, but we do get a hint at how far it is possible to go. In TURBO Pascal, you are allowed to have with statements nested to nine levels, and it would be worthless to nest with statements deeper than the level of records. Any program requiring more levels than nine is probably far beyond the scope of your

programming ability, and mine, for a long time. Pascal implementations other than TURBO Pascal probably have their own with statement nesting limitation. Check your reference manual.

CHAPTER 9 - Records

After assigning a value to Year, the entire record of Self is defined, all nine variables. It should be pointed out that even though Self is composed of nine separate variables, it is proper to call Self a variable itself because it is a record variable.

SUPER-ASSIGNMENT STATEMENTS

The statement in line 45, "Mother := Self;" is very interesting. Since both of these are records, both are the same type of record, and both therefore contain 9 variables, Pascal is smart enough to recognize that, and assign all nine values contained in Self to the corresponding variables of Mother. So after one statement, the record variable Mother is completely defined. The statement in line 46 assigns the same values to the nine respective variables of Father, and the next two lines assign all 50 Friend variables the same data. By this point in the program, we have therefore generated 450 + 27 = 477 separate pieces of data so far in this program. We could print it all out, but since it is nonsense data, it would only waste time and paper. Lines 49 through 52 write out three sample pieces of the data for your inspection.

WHAT GOOD IS ALL OF THIS

It should be obvious to you that what this program does, even though the data is nonsense, appears to be the beginning of a database management system, which indeed it is. Instead of assigning nonsense data, a list could be read in and stored for manipulation. It is a crude beginning, and has a long way to go to be useful, but you should see a seed for a useful program.

Now to go back to the const in line 4 as promised. The number of friends was defined as 50 and used for the size of the array and in the assignment loop in line 47. You can now edit this number and see how big this database can become on your computer. If you are using TURBO Pascal, you will be limited to slightly more than 1000 because of the 64K limitation of an executable program, and the fact that all of this data is stored within that 64K boundary. It should be noted that TURBO Pascal 4.0 allows a program larger than 64K but still places a limitation of 64K on each compilation unit. See how big you can make the number of friends before you get the memory overflow message. Keep the number in mind because when we get to the chapter on Pointers and Dynamic Allocation, you will see a marked increase in allowable size, especially if you have a large amount of RAM installed in your computer.

CHAPTER 9 - Records

A VARIANT RECORD

If any part of this chapter is still unclear, it would be good for you to go back and review it at this time. The next example will really tax your mind to completely understand it, especially if the prior material is not clear.

Examine the Pascal program VARREC for an example of a program with a variant record definition. In this example, we first define a scalar type, namely Kind_Of_Vehicle for use within the record. Then we have a record defining Vehicle, intended to define several different vehicles, each with different kinds of data. It would be possible to define all variables for all types of vehicles, but it would be a waste of storage space to define the number of tires for a boat, or the number of propeller blades used on a car or truck. The variant record lets us define the data precisely for each vehicle without wasting data storage space.

WHAT IS A TAG-FIELD?

In the record definition we have the usual record header followed by three variables defined in the same manner as the records in the last two example programs. Then we come to the case statement. Following this statement, the record is different for each of the four types defined in the associated scalar definition. The variable What_Kind is called the tag-field and must be defined as a scalar type prior to the record definition. The tag-field is used to select the variant, when the program uses one of the variables of this record type. The tag-field is followed by a colon and its type definition, then the reserved word "of". A list of the variants is then given, with each of the variants having the variables for its particular case defined. The list of variables for one variant is called the field list.

A few rules are in order at this point. The variants do not have to have the same number of variables in each field list, and in fact, one or more of the variants may have no variables at all in its variant part. If a variant has no variables, it must still be defined with a pair of empty parentheses followed by a semi-colon. All variables in the entire variant part must have unique names. The three variables, Wheels, Tires, and Tyres, all mean the same thing to the user, but they must be different for the compiler. You may use the same identifiers again in other records and for simple variables anywhere else in the program. The Pascal compiler can tell which variable you

mean by its context. Using the same variable name should be discouraged as bad programming practice because it may confuse you or another person trying to understand your program at a later date.

The final rule is that the variant part of the record must be the last part of it, and in fact, the last part of any or all variants can itself have a variant part to it. That is getting pretty advanced for our level of use of Pascal at this time however.

USING THE VARIANT RECORD

We properly define four variables with the record type Vehicle in line 22 and go on to examine the program itself.

We begin by defining one of our variables of type Vehicle, namely the variable named Ford. The seven lines assigning values to Ford are similar to the prior examples with the exception of line 28. In that line the tag-field which selects the particular variant used is set equal to the value Truck, which is a scalar definition, not a variable. This means that the variables named Motor, Tires, and Payload are available for use with the record Ford, but the variables named Wheels, Engine, Tyres, etc. are not available in the record named Ford.

Next, we will define the record Sunfish as a Boat, and define all of its variables in lines 33 through 41. All of Sunfish's variables are defined but in a rather random order to illustrate that they need not be defined in a particular order. You should remember the with statement from the last example program.

To go even further in randomly assigning the variables

to a record, we redefine Ford as having an Engine which it can only have if it is a car. This is one of the fine points of the Pascal record. If you assign any of the variant variables, the record is changed to that variant, but it is the programmers responsibility to assign the correct tag-field to the record, not Pascal's. Good programming practice would be to assign the tag-field before assigning any of the variant variables. The remainder of the Ford variables are assigned to complete that record, the non-variant part remaining from the last assignment.

The variable Mac is now set equal to the variable Sunfish in line 48. All variables within the record are copied to Mac including the tag-field, making Mac a Boat.

NOW TO SEE WHAT WE HAVE IN THE RECORDS

We have assigned Ford to be a car, and two boats exist, namely Sunfish and Mac. Since Schwinn was never defined, it has no data in it, and is at this point useless. The Ford tag-field has been defined as a car, so it should be true in the if statement, and the message in line 51 should print. The Sunfish is not a bicycle, so it will not print. The Mac has been defined as a boat in the single assignment statement, so it will print a message with an indication that all of the data in the record was transferred to its variables.

Even though we can make assignment statements with records, they cannot be used in any mathematical operations such as addition, or multiplication. They are simply used

for data storage. It is true however, that the individual elements in a record can be used in any mathematical statements legal for their respective types.

One other point should be mentioned. The tag-field can be completely eliminated resulting in a "free union" variant record. This is possible because Pascal, as you may remember from above, will automatically assign the variant required when you assign data to one of the variables within a variant. This is the reason that all variables within any of the variants must have unique names. The free union record should be avoided in your early programming efforts because you cannot test a record to see what variant it has been assigned to it. It is definitely an advanced technique in Pascal.

Be sure you compile and run VARREC and study the output until you understand it completely.

PROGRAMMING EXERCISE

1. Write a simple program with a record to store the names of five of your friends and display the names.

CHAPTER 10 - Standard Input/Output

During the course of this tutorial we have been using the Write and Writeln procedures to display data, and it is now time to discuss them fully. Actually there is little to

be said that has not already been said, but in order to get all of the data in one place they will be redefined here.

As mentioned earlier, Write and Writeln are not actually reserved words but are procedure calls. They are therefore merely identifiers that could be changed but there should never be a reason to do so. Lets get on to our first example program WRITELNX which has lots of output.

MANY OUTPUT STATEMENTS

Pascal has two output statements that are only slightly different in the way they work. The Writeln statement outputs all of the data specified within it, then returns the cursor to the beginning of the next line. The Write statement outputs all of the data specified within it, then leaves the cursor at the next character where additional data can be output. The Write statement can therefore be used to output a line in bits and pieces if desired for programming convenience. The first example program for this chapter, WRITELNX, has many output statements for your observation. All outputs are repeated so you can observe where the present field ends and the next starts.

Observe the integer output statements beginning in line 13. The first simply directs the system to output Index twice, and it outputs the value with no separating blanks. The second statement says to output Index twice also, but it instructs the system to put each output in a field 15 characters wide with the data right justified in the field. This makes the output look much better. This illustrates that you have complete control over the appearance of your output data.

The real output statements are similar to the integer except that when the data is put into a field 15 characters wide, it is still displayed in scientific format. Adding a second field descriptor tells the system how many digits you want displayed after the decimal point. Lines 21 through 23 illustrate the second field and its use.

The boolean, char, and string examples should be self explanatory. Notice that when the string is output, even though the string has been defined as a maximum of 10 characters, it has been assigned a string of only 8 characters, so only 8 characters are output.

Compile and run this program and observe the results.

If you are using TURBO Pascal version 4.0, the added data types described in chapter 3 of this tutorial are output in the same manner as those illustrated in the program WRITELNX.

NOW FOR SOME INPUT FROM THE KEYBOARD

The example file READINT will illustrate reading some integer data from the keyboard. A message is output in line 8 with an interesting fact that should be pointed out. Anyplace where Pascal uses a string variable or constant, it uses the apostrophe for a delimiter. Therefore, anyplace where an apostrophe is used in a string, it will end the string. Two apostrophes in a row will be construed as a single apostrophe within the string and will not terminate the string. The term 'Read' within the string will therefore be displayed as shown earlier in this sentence.

The variable Index is used to loop five times through a sequence of statements with one Read statement in it. The three integer values are read in and stored in their respective variables with the one statement. If less than three are entered at the keyboard, only as many as are read in will be defined, the rest will be unchanged. Following completion of the first loop, there is a second loop in lines 19 through 25 that will be executed 5 times with only one minor change, the Read statement is replaced by the Readln statement. At this point it would be best run this program trying several variations with input data.

When you run READINT, it will request three integers. Reply with three small integers of your choice with as many blank spaces between each as you desire, followed by a carriage return. The system will echo your three numbers back out, and request three more. Respond with only one number this time, different from each of the first three, and a carriage return. You will get your new number followed by your previous second and third number indicating that you did not re-enter the last two integer variables. Enter three more numbers, this time including a negative number and observe the echo once again.

Continue entering numbers until the system outputs the message indicating that it will now be using the Readln for reading data. At this point enter the same numbers that you did in the previous section and notice the difference, which is only very slight. Each time you hit the enter key to cause the computer to process the data you have just given it, it will echo the carriage return to the display, and the

"Thank you" message will be on a new line. When entering data from the keyboard, the only difference in Read and Readln is whether or not the carriage return is echoed to the display following the data read operation.

It should not be a surprise to you that after you enter the data, the data is stored within the program and can be used anywhere that integer data is legal for use. Thus, you could read in an integer, and use the integer to control the number of times through a loop, as a case selector, etc.

TIME TO CRASH THE COMPUTER

Crashing the computer will not hurt a thing. Rerun the

above program and instead of entering integer data, enter some real data with decimal points, or even some character data. The computer should display some kind of message indicating that you have caused an I/O error (Input/Output), and TURBO Pascal will abort operation (that simply means to stop the program and return control to the operating system). No harm has been done, simply start it again to enter more numbers or errors.

READING REAL NUMBERS

The example program READREAL will illustrate how to read real numbers into the computer. It will read an integer and three real numbers each time through the loop. It is perfectly fine to give the system a number without a decimal point for a real number. The computer will simply read it as a decimal number with zeros after the decimal point and consider it as a real number internally. As you found out in the last example program, however, it is not permissible to include a decimal point in the data if the computer is looking for an integer variable. Include some character data for a real number and crash the system in this program too.

READING CHARACTER DATA

The next example program, READCHAR, will read in one character each time through the loop and display it for you. Try entering more than one character and you will see that the extra characters will simply be ignored. It is not possible to crash this program because any character you enter will be valid.

The next example, READARRY, will read in a string of characters and display them for you if you are using TURBO Pascal 3.0. TURBO Pascal 4.0 does not allow reading into an array but does allow reading into the individual elements of

the array one element at a time. This program does not work with TURBO Pascal 4.0 so you should go directly to the next program, READSTRG, if you are using that version.

Continuing our discussion of READARRY, up to 10 characters will be read, and if less than 10 are read, the rest will be blank filled. Try entering 10 characters, then 4, to see that the residual 6 characters are blanked out before storing and printing. Since the array is fixed at ten characters, ten characters are always printed out, including trailing blanks.

Finally READSTRG will also read up to 10 characters, but since a string is a dynamic length variable, it will only print out the characters you input each time, up to the maximum of 10 as defined in the var declaration. It will display trailing blanks if you type them in because blanks are valid characters.

BULLET PROOF PROGRAMMING

It can be frustrating to be running a program and have it declare an I/O error and terminate operation simply because you have entered an incorrect character. The integer and real data inputs defined earlier in this chapter are fine for quick little programs to do specific calculations, but if you are writing a large applications program it is better to use another technique. Since the character and string inputs cannot abort operation of the program, it is best to use them to input the variable data and check the data internally under your own program control. An error message can then be given to the operator and another opportunity granted to input the correct data. All well written large application programs use this technique.

HOW DO I PRINT SOMETHING ON THE PRINTER

With all of the Pascal knowledge you now have, it is the simplest thing in the world to get data to the printer. The example file PRINTOUT will show you graphically how to do it. Every Write or Writeln statement is required to have

a device identifier prior to the first output field. If there is none, it is automatically defaulted to the standard output device, the display monitor. The example program has a few outputs to the monitor in lines 9 and 10 with the device identifier included, namely "Output". This is only done to show you the general form of the Write statements. There are also many statements in this program with the display identifier "Lst", which is the standard name for the list device or the printer.

Compile and run this program with your printer turned on for some printer output. If you are using TURBO Pascal 3.0, you will have to comment out line 4 since it will not be understood by your compiler. It is required with version 4.0 to tell the system where to find the output device name Lst.

Just to supply you with a bit more information, every Read and Readln statement is also required to have a device identifier prior to the first input field. As you may suspect, it is also defaulted to Input if none is specified, and the standard input device is the keyboard.

PROGRAMMING EXERCISE

1. Write a program containing a loop to read in a character string up to 60 characters long, then print the string on your printer. When you run the program, you will have the simplest word processing program in the world. Be sure to include a test to end the loop, such as when "END" is typed in.

CHAPTER 11 - Files

One of the most common operations when using a computer is to either read from, or write to a file. You are already somewhat experienced in file handling from the last chapter, because in computer terminology, the keyboard, terminal, and printer are all classified as files. A file is any serial input or output device that the computer has access to. Since it is serial, only one piece of information is available to the computer at any instant of time. This is in contrast to an array, for example, in which all elements of the array are stored internally and are all available at any

time.

A SHORT HISTORY LESSON

Several years ago computers were all large cumbersome machines with large peripheral devices such as magnetic tape drives, punch card readers, paper tape readers or punches, etc. It was a simple task to assign the paper tape reader a symbol and use that symbol whenever it was necessary to read a paper tape. There was never more than one file on the paper tape being read, so it was simply read sequentially, and hopefully the data was the desired data. With the advent of floppy disks, and hard disks, it became practical to put several files of data on one disk, none of which necessarily had anything to do with any of the other files on that disk. This led to the problem of reading the proper file from the disk, not just reading the disk.

Pascal was originally released in 1971, before the introduction of the compact floppy disk. The original release of Pascal had no provision for selecting a certain file from among the many included on the disk. Each compiler writer had to overcome this deficiency and he did so by defining an extension to the standard Pascal system. Unfortunately, all of the extensions were not the same, and there are now several ways to accomplish this operation. There are primarily two ways, one using the Assign statement, and the other using the Open statement. They are similar to each other and they accomplish the same end result.

BACK TO THE PRESENT TIME

All of the above was described to let you know that we will have a problem in this chapter, namely, how do we cover all of the possible implementations of Pascal available? The answer is, we can't. Most of what is covered in this chapter will apply to all compilers, and all that is covered will apply to the TURBO Pascal compilers, versions 3.0 and 4.0. If you are not using TURBO Pascal and your compiler complains about some of the statements, it will be up to you

to dig out the details of how to do the intended operations. If any one or more of these operations cannot be accomplished with your compiler, you should seriously consider purchasing a better compiler because all of these operations are needed in a useful Pascal environment.

READING AND DISPLAYING A FILE

Examine the file READFILE for an example of a program that can read a text file from the disk. In fact it will read itself from the disk and display it on the video monitor. The first statement in the program is the Assign statement. This is TURBO Pascal's way of selecting which file on the disk will be either read from or written to. In this case we will read from the disk. The first argument in the Assign statement is the device specifier similar to Lst used in the last chapter for the printer. We have chosen to use the name Turkey for the device identifier, but could have used any valid identifier. This identifier must be defined in a var declaration as a TEXT type variable. The next argument is the filename desired. The filename can be defined as a string constant, as it is here, or as a string variable.

The "TEXT" type is a predefined type and is used to define a file identifier. It is predefined as a "file of char", so it can only be used for a text file. We will see later that there is another type of file, a binary file.

Now that we have a file identified, it is necessary to prepare it for reading by executing a reset statement in line 9. The reset statement positions the read pointer at the beginning of the file ready to read the first piece of information in the file. Once we have done that, data is read from the file in the same manner as it was when reading from the keyboard. In this program, the input is controlled by the while loop which is executed until we exhaust the data in the file.

WHAT ARE THE "EOF" AND "EOLN" FUNCTIONS?

The Eof function is new and must be defined. When we read data from the file, we move closer and closer to the end, until finally we reach the end and there is no more data to read. This is called "end of file" and is abbreviated Eof. Pascal has this function available as a part of the standard library which returns FALSE until we reach the last line of the file. When there is no more data to read left in the file, the function Eof returns TRUE. To use the function, we merely give it our file identifier as

an argument. It should be clear that we will loop until we read all of the data available in the input file.

The Eoln function is not used in this program but is a very useful function. If the input pointer is anywhere in the text file except at the end of a line, the Eoln returns FALSE, but at the end of a line, it returns a value of TRUE. This function can therefore be used to find the end of a line of text for variable length text input.

To actually read the data, we use the Readln procedure, giving it our identifier Turkey and the name of the variable we want the data read into. In this case, we read up to 80 characters into the string and if more are available, ignore them. You should remember when we did this in the last chapter from the keyboard input. We are using the same technique here except we are reading from a file this time. Since we would like to do something with the data, we output the line to the default device, the video monitor. It should be clear to you by now that the program will read the entire file and display it on the monitor.

Finally, we Close the file Turkey. It is not really necessary to close the file because the system will close it for you automatically at program termination, but it is a good habit to get into. It must be carefully pointed out here, that you did not do anything to the input file, you only read it and left it intact. You could Reset it and reread it again in this same program.

Compile and run this program to see if it does what you expect it to do.

A PROGRAM TO READ ANY FILE

Examine the next program READDISP for an improved file reading program. This is very similar except that it asks you for the name of the file that you desire to display, and enters the name into a 12 character string named Name_Of_File_To_Input. This is then used in the Assign statement to select the file to be read, and the file is reset as before. Lines 15 through 18 display a header, and from that point on, the program is identical to the last one with a few small additions. In order to demonstrate the use of a function within the Writeln specification, the program calls for the length of the input string in line 23 and displays it before each line. The lines are counted as they are read and displayed, and the line count is displayed at the end of the listing.

You should be able to see clearly how each of these operations is accomplished. Compile and run this program,

entering any filename we have used so far (be sure to include the .PAS extension). After a successful run, enter a nonexistent filename and see the I/O error.

HOW TO COPY A FILE (SORT OF)

Examine the file READSTOR for an example of both reading from a file and writing to another one. In this program we request an operator input for the filename to read, after which we Assign the name to the file and Reset it. When we reset the file however, we go to a bit of extra trouble to assure that the file actually exists.

Suppose we input a filename, and the file did not exist because the file was actually missing, or because we entered the filename wrong. Without the extra effort, the TURBO Pascal runtime system would indicate a run-time error, and terminate the program returning us to the operating system. In order to make a program easier to use, it would be nice to tell the operator that the file didn't exist and give him the opportunity to try again with another file name. The method given in lines 16 through 20 of this program will allow you to do just that.

USING A COMPILER DIRECTIVE

First you must disable the built in TURBO Pascal I/O checking by inserting the compiler directive in line 16. This tells the system to ignore any I/O errors from this point on and if the file doesn't exist, the system will not abort when you attempt to reset it in line 17. Another compiler directive is given in line 18 to enable I/O checking again for the remainder of the program.

WE DO OUR OWN FILE CHECKING

If the file didn't exist and could not therefore be reset, we have a problem because the program thinks the file is available for use but it actually isn't. Fortunately, TURBO Pascal has a built in variable, named "IOResult", that informs us of the result of each I/O operation. Following any I/O operation, if this variable contains the value of zero, the I/O operation was correct, and if it contains any other value, the operation had some sort of error. In our case, we simply compare it to zero to generate a boolean value, then based on the boolean value we either give an error message and stop, or perform the desired file

operations.

It would be good programming practice to check all file openings in this manner to allow the operator to recover from a simple oversight or spelling error.

If the file was opened properly, then in line 21 through 24 we request a different filename to write to, which is assigned to a different identifier. Note that the output file is not checked for a valid opening as it should be. The statement in line 24 is new to us, the Rewrite statement. This name apparently comes from the words REset for WRITEing because that is exactly what it does. It clears the entire file of any prior data and prepares to write into the very beginning of the file. Each time you write into it, the file grows by the amount of the new data written.

Once the identifier has been defined, and the Rewrite has been executed, writing to the file is identical to writing to the display with the addition of the identifier being specified prior to the first output field. With that in mind, you should have no trouble comprehending the operation of the program. This program is very similar to the last, except that it numbers the lines as the file is copied. After running the program, look in your default directory for the new filename which you input when it asked for the output filename. Examine that file to see if it is truly a copy of the input file with line numbers added.

One word of caution. If you used an existing filename for the output file, the file was overwritten, and the

original destroyed. In that case, it was good that you followed instructions at the beginning of this tutorial and made a working copy of the distribution disk. You did do that, didn't you?

Compile and run this program two different ways, once with a valid input filename that should run properly, and the second time with an input filename that doesn't exist to prove to yourself that the test actually does work correctly.

HOW TO READ INTEGER DATA FROM A FILE

It is well and good to be able to read text from a file, but now we would like to read other forms of data from a file. First we will look at an example program to read data from a text file, then later we will see an example program that reads from a binary file.

Examine the program READINTS for an example of reading data from a text file. A text file is an ASCII file that

can be read by a text editor, printed, displayed, or in some cases, compiled and executed. It is simply a file made up of a long string of char type data, and usually includes linefeeds, carriage returns, and blanks for neat formatting. Nearly every file on the Tutorial disk you received with this package is a text file. One notable exception is the file named LIST.COM, which is an executable program file.

The example program has nothing new, you have seen everything in it before. We have an assignment, followed by a reset of our file, followed by four read and write loops.

Each of the loops has a subtle difference to illustrate the Read and Readln statements. Notice that the same file is used for reading four times with a Reset prior to each, illustrating the nondestructive read mentioned a few paragraphs ago.

The file we will be using is named INTDATA.TXT and is on your disk. You could display it at this time using the program READDISP we covered recently. Notice that it is simply composed of the integer values from 101 to 148 arranged four to a line with a couple of spaces between each for separation and a neat appearance. The important thing to remember is that there are four data points per line.

READ AND READLN ARE SLIGHTLY DIFFERENT

As variables are read in with either procedure, the input file is scanned for the variables using blanks as delimiters. If there are not enough data points on one line to satisfy the arguments in the input list, the next line is searched also, and the next, etc. Finally when all of the arguments in the input list are satisfied, the Read is complete, but the Readln is not. If it is a Read procedure, the input pointer is left at that point in the file, but if it is a Readln procedure, the input pointer is advanced to the beginning of the next line. The next paragraph should clear that up for you.

The input data file INTDATA.TXT has four data points per line but the first loop in the program READINTS.PAS requests only three each time through the loop. The first time through, it reads the values 101, 102, and 103, and displays those values, leaving the input pointer just prior to the 104, because it is a Read procedure. The next time through, it reads the value 104, advances to the next line and reads the values 105, and 106, leaving the pointer just prior to the 107. This continues until the 5 passes through the loop are completed.

The loop in lines 19 through 22 contains a Readln procedure and also reads the values 101, 102, and 103, but when the input parameter list is satisfied, it moves the pointer to the beginning of the next line, leaving it just before the 105. The values are printed out and the next time we come to the Readln, we read the 105, 106, and 107, and the pointer is moved to the beginning of the next line. It would be good to run the program now to see the difference in output data for the two loops. Remember that the only difference is that the first loop uses the Read procedure, and the second uses the Readln procedure.

When you come back to the program again, observe the last two loops, which operate much like the first two except that there are now five requested integer variables, and the input file still only has four per line. This is no problem. Both input procedures will simply read the first four in the first line, advance to the second line for its required fifth input, and each will do its own operation next. The Read procedure will leave the input pointer just before the second data point of the second line, and the Readln will advance the input pointer to the beginning of the third line. Run this program and observe the four output fields to see an illustration of these principles.

NOW TO READ SOME REAL VARIABLES FROM A FILE

By whatever method you desire, take a look at the file named REALDATA.TXT supplied on your Pascal Tutorial disk. You will see 8 lines of what appears to be scrambled data, but it is good data that Pascal can read. Notice especially line 4 which has some data missing, and line 6 which has some extra data.

Examine the program file READDATA which will be used to illustrate the method of reading real type data. Everything should be familiar to you, since there is nothing new here. The Readln statement is requesting one integer variable, and three real variables, which is what most of the input file

contained. When we come to the fourth line, there are not enough data points available, so the first two data points of the next line are read to complete the fourth pass through the loop. Since the file pointer is advanced to the beginning of the next line, we are automatically synchronized with the data again. When we come to the sixth line, the last two data points are simply ignored. Run the program to see if the results are as you would predict.

If a Read were substituted for the Readln in line 14 of the program, the file pointer would not be advanced to the beginning of line 6, after the fourth pass through the loop.

The next attempt to read would result in trying to read the value 0.0006 as an integer, and a run time error would result. Modify the program, substituting a Read for the Readln in line 14, and see if this is not true.

It should be pointed out that TURBO Pascal 4.0 requires a digit both before and after the decimal point in all data that is to be read in as real type data or it will be flagged as a run-time error and the program will be halted. The digits can be zero as they are in several places in the example file but they must be there. If you are using TURBO Pascal 3.0, the leading and trailing digits are not required.

That is all there is to reading and writing text files. If you learn the necessities, you will not be stumbling around in the area of input/output which is very intimidating to many people. Remember to Assign, then Reset before reading, Rewrite before writing, and Close before quitting. It is of the utmost importance to close a file you have been writing to before quitting to write the last

few buffers to the file, but it is not as important to close read files unless you are using a lot of them, as there is an implementation dependent limit of how many files can be open at once. It is possible to read from a file, close it, reopen it, and write to it in one program. You can reuse a file as often as you desire in a program, but you cannot read from and write into a file at the same time.

NOW FOR BINARY INPUT AND OUTPUT

Examine the file BINOUT for an example of writing data to a file in binary form. First there is a record defined in the type declaration part composed of three different variable types. In the var part, Output_File is defined as a "file of Dat_Rec", the record defined earlier. The variable Dog_Food is then defined as an array of the record, and a simple variable is defined.

Any file assigned a type of TEXT, which is a "file of char", is a text file. A text file can be read and modified with a text editor, printed out, displayed on the monitor, etc. If a file is defined with any other definition, it will be a binary file and will be in an internal format as defined by the Pascal compiler. Attempting to display such a file will result in very strange looking gibberish on the monitor.

When we get to the program, the output file is assigned a name in line 15, and a Rewrite is performed on it to reset the input pointer to the beginning of the file, empty the

file, and prepare for writing data into it. The loop in lines 18 through 22 simply assigns nonsense data to all of

the variables in the 20 records so we have something to work with.

We write a message to the display that we are ready to start outputting data, and we output the data one record at a time with the standard Write statement. A few cautions are in order here. The output file can be defined as any simple variable type, integer, byte, real, or a record, but the types cannot be mixed. The record itself however, can be any combination of data including other records if desired, but any file can only have one type of record written to it.

A Writeln statement is illegal when writing to a binary file because a binary file is not line oriented. A Write statement is limited to one output field per statement. It is a simple matter to put one Write statement in the program for each variable you wish to write out to the file. It is important to Close the file when you are finished writing to it.

WHY USE A BINARY FILE

A binary file written by a Pascal program cannot be read by a word processor, a text editor or any other application program such as a database or spreadsheet, and it may not even be readable by a Pascal program compiled by a different companies compiler because the actual data structure is implementation dependent. It can't even be read by a Pascal program using the same compiler unless the data structure is identical to the one used to write the file. With all these rules, it seems like a silly way to output data, but there are advantages to using a binary output.

A binary file uses less file space than a corresponding text file because the data is stored in a packed mode. Since all significant digits of real data are stored, it is more precise unless you are careful to output all significant data to the corresponding TEXT file. Finally, since the binary data does not require formatting into ASCII characters, it will be considerably faster than outputting it in TEXT format. When you run this example program, it will create the file KIBBLES.BIT, and put 20 records in it. Return to DOS and look for this file and verify its existence. If you try to TYPE it, using the DOS TYPE command, you will have a real mess, but that might be a good

exercise.

CHAPTER 11 - Files

READING A BINARY FILE

BININ is another example program that will read in the file we just created. Notice that the variables are named differently, but the types are all identical to those used to write the file. An additional line is found in the program, the if statement. We must check for the "end of file" marker to stop reading when we find it or Pascal will list an error and terminate operation. Three pieces of information are written out to verify that we actually did read the data file in.

Once again, a few rules are in order. A Readln is illegal since there are no lines in a binary file, and only one variable or record can be read in with each Read statement.

WHAT ABOUT FILE POINTERS, GET, AND PUT STATEMENTS?

File pointers and the Get and Put procedures are a part of standard Pascal, but since they are redundant, they are not a part of TURBO Pascal. The standard Read and Write procedures are more flexible, more efficient, and easier to use. The use of Get and Put will not be illustrated or defined here. If you ever have any need for them, they should be covered in detail in your Pascal reference manual for the particular implementation you are using.

Pointers will be covered in detail in the next chapter of this tutorial.

PROGRAMMING EXERCISES

1. Write a program to read in any text file, and display it on the monitor with line numbers and the number of characters in each line. Finally display the number of lines found in the file, and the total number of characters in the entire file. Compare this number with the filesize given by the DOS command DIR.

2. Write a silly program that will read two text files and display them both on the monitor on alternating lines. This is the same as "shuffling" the two files together. Take care to allow them to end at different times, inserting blank lines for the file that terminates earlier.

CHAPTER 12 - Pointers and Dynamic Allocation

For certain types of programs, pointers and dynamic allocation can be a tremendous advantage, but many programs do not need such a high degree of data structure. For that reason, it would probably be to your advantage to lightly skim over these topics and come back to them later when you have a substantial base of Pascal programming experience. It would be good to at least skim over this material rather than completely neglecting it, so you will have an idea of how pointers and dynamic allocation work and that they are available for your use when needed.

A complete understanding of this material will require deep concentration as it is complex and not at all intuitive. Nevertheless, if you pay close attention, you will have a good grasp of pointers and dynamic allocation in a short time.

WHAT ARE POINTERS, AND WHAT GOOD ARE THEY?

Examine the program named POINT for your first example of a program using pointers. In the var declaration you will see two variables named Where and Who that have the symbol ^ in front of their types. This defines them, not as variables, but as pointers to integer type variables. In line 12 of the program, the variable Index is assigned the value of 17 for purposes of illustration. The pointer named Where is then assigned the address of the variable Index which means that it does not contain the value of 17, it contains the address of the storage location where the variable Index is stored. In like manner, we assign the address of Index to the pointer named Who. It should be obvious to you that Addr is a TURBO Pascal function that returns the address of its argument.

HOW DO WE USE THE POINTERS?

It should be clear to you that we now have a single variable named Index with two pointers pointing at it. If the pointers are useful, we should be able to do something with them now, so we simply print out the same variable three different ways in line 15. When we write "Where^", we are telling the system that we are not interested in the pointer itself but instead we are interested in the data to which the pointer points. This is referred to as dereferencing the pointer. Careful study of the output fields in line 15 will reveal that we first display the value of Index, then the value to which the pointer Where points, and finally the value to which the pointer Who points. Since both pointers point to the variable Index, we are essentially displaying the value of Index three times. You will confirm this when you compile and run this program.

In line 17, we tell the system to assign the value of 23 to the variable to which the pointer Where points as an illustration. If you understood the discussion in the previous paragraph, you will understand that we are actually assigning the variable named Index the value of 23 because that is where the pointer named Where is pointing. In line 18, we once again display the value of the variable Index 3 times just as we did in line 15. It would be to your advantage to compile and run this program to see that the value of 17 is output three times, then the value of 23 is output three times.

In a program as simple as this, the value of pointers is not at all clear but a simple program is required in order to make the technique clear. Display the program named POINT on your monitor again because we are not yet finished with it.

A FEW MORE POINTERS

In line 4, we define a new type named Int_Point which is a pointer to an integer type variable. We use this new type in line 9 to define three more pointers and in line 20, we assign one of them the address of the variable named Index. Since the pointers are of identical types, in line 21 we can assign Pt2 the value of Pt1, which is actually the address of the variable named Index. Likewise, the pointer Pt3 is assigned the value of Pt2, and we have all three pointers pointing to the variable named Index. If you are using TURBO Pascal version 4.0, you are allowed to assign pointers like this only if they have the same type, which these three do. However, since the pointers named Where and Who are assigned individually, they are not of the same type according to the rules of Pascal and if line 14 were changed to read "Who := Where;", a compilation error would occur with TURBO Pascal version 4.0. This error would not occur with TURBO Pascal 3.0 since it is a little less stringent in its type checking.

Finally, we assign the only variable in this program which is named Index the value of 151 in line 23 and display the value 151 three times as we did above. Compile and run this program again to see that it does indeed display the value 151 three times.

THIS IS JUST FOR TURBO PASCAL VERSION 4.0

If you are using TURBO Pascal version 4.0, you should display the program named POINT4 on your monitor for an example of another new extension to the Pascal programming

language by Borland. This program is identical to the last except in lines 13, 14 and 20, where the symbol @ is used to denote the address of the variable index rather than the function "Addr". This is only available in TURBO Pascal version 4.0 as a convenience to you. In ANSI standard Pascal the @ symbol is used as a synonym for the ^ symbol but Borland chose to use it for a completely different purpose. If you are using TURBO Pascal 3.0, you will not be able to compile and run this program, but nothing is lost because it is identical to the previous one.

OUR FIRST LOOK AT DYNAMIC ALLOCATION

If you examine the file named POINTERS, you will see a very trivial example of pointers and how they are used with dynamically allocated variables. In the var declaration, you will see that the two variables have a ^ in front of their respective types once again, defining two pointers. They will be used to point to dynamically allocated variables that have not yet been defined.

The pointer My_Name is a pointer to a 20 character string. The pointer actually points to an address somewhere within the computer memory, but we don't know where yet. Actually, there is nothing for it to point at because we have not defined a variable. After we assign it something to point to, we can use the pointer to access the data stored at that address.

Your computer has some amount of memory installed in it. If it is an IBM-PC or compatible, it can have up to 640K of RAM which is addressable by various programs. The operating system requires about 60K of the total, and if you are using TURBO Pascal version 3.0, it requires about 35K, but if you are using version 4.0, it requires about 110K. The TURBO Pascal program can use up to 64K. Adding those three numbers together results in about 159K or 234K. Any memory you have installed in excess of that is available for the stack and the heap. The stack is a standard area defined and controlled by DOS that can grow and shrink as needed. Many books are available to define the stack and its use if you are interested in more information on it.

WHAT IS THE HEAP?

The heap is a Pascal defined entity that utilizes otherwise unused memory to store data. It begins immediately following the program and grows as necessary upward toward the stack which is growing downward. As long as they never meet, there is no problem. If they meet, a run-time error is generated. The heap is therefore outside

of the 64K limitation of TURBO Pascal and many other Pascal compilers.

TURBO Pascal version 4.0 does not limit us to 64K, but there are other reasons for using the heap in addition to the 64K limitation. These should be evident as we learn how the heap works.

If you did not understand the last few paragraphs, don't worry. Simply remember that dynamically allocated

variables are stored on the heap and do not count in the 64K limitation placed upon you by some compilers.

Back to our example program, POINTERS. When we actually begin executing the program, we still have not defined the variables we wish to use to store data in. The first executable statement in line 10 generates a variable for us with no name and stores it on the heap. Since it has no name, we cannot do anything with it, except for the fact that we do have a pointer My_Name that is pointing to it. By using the pointer, we can store up to 20 characters in it, because that is its type, and later go back and retrieve it.

WHAT IS DYNAMIC ALLOCATION?

The variable we have just described is a dynamically allocated variable because it was not defined in a var declaration, but with a "New" procedure. The "New" procedure creates a variable of the type defined by the pointer, puts it on the heap, and finally assigns the address of the variable to the pointer itself. Thus My_Name contains the address of the variable generated. The variable itself is referenced by using the pointer to it followed by a ^, just like in the last program, and is read, "the variable to which the pointer points".

The statement in line 11 assigns a place on the heap to an integer type variable and puts its address in My_Age. Following the "New" statements we have two assignment statements in which the two variables pointed at are assigned values compatible with their respective types, and they are both written out to the video display in much the same manner as we did in the program named POINT.

GETTING RID OF DYNAMICALLY ALLOCATED DATA

The two statements in lines 19 and 20 are illustrations of the way the dynamically allocated variables are removed from use. When they are no longer needed, they are disposed

of with the Dispose procedure which frees up their space on the heap so it can be reused.

In such a simple program, pointers cannot be appreciated, but it is necessary for a simple illustration. In a large, very active program, it is possible to define many variables, dispose of some of them, define more, and dispose of more, etc. Each time some variables are disposed of, their space is then made available for additional variables defined with the "New" procedure.

The heap can be made up of any assortment of variables, they do not have to all be the same. One point must be kept in mind. Anytime a variable is defined, it will have a pointer pointing to it. The pointer is the only means by which the variable can be accessed. If the pointer to the variable is lost or changed, the data itself is lost for all practical purposes.

Compile and run this program and examine the output.

DYNAMICALLY STORING RECORDS;

The next example program, DYNREC, is a repeat of one we studied in an earlier chapter. For your own edification, review the example program BIGREC before going ahead in this chapter. Assuming that you are back in DYNREC, you will notice that this program looks very similar to the earlier one, and in fact they do exactly the same thing. The only difference in the type declaration is the addition of a pointer Person_Id, and in the var declaration, the first four variables are defined as pointers here, and were defined as record variables in the last program.

A point should be made here. Pointers are not generally used in very small programs. This program is a good bit larger than the last and should be a clue to you as to why such a trivial program was used to introduce pointers

in this tutorial. A very small, concise program can illustrate a topic much better that an large complex program, but we must go on to more useful constructs of any new topic.

WE JUST BROKE THE GREAT RULE OF PASCAL

Notice in the type declaration that we used the identifier Person in line 18 before we defined it in line 19, which is illegal to do in Pascal. Foreseeing the need to define a pointer prior to the record, the designers of Pascal allow us to break the rule in this one place. The pointer could have been defined after the record in this

particular case, but it was more convenient to put it before, and in the next example program, it will be required to put it before the record. We will get there soon.

Since Friend is really 50 pointers, we have now defined 53 different pointers to records, but so far have defined no variables other than Temp and Index. We immediately use the New procedure to dynamically allocate a record with Self pointing to it, and use the pointer so defined to fill the dynamically allocated record. Compare this to the program named BIGREC and you will see that it is identical except for the addition of the "New" and adding the ^ to each use of the pointer to designate the data pointed to.

THIS IS A TRICK, BE CAREFUL

Now go down to line 48 where Mother is allocated a record and is then pointing to the record. It seems an easy thing to do then to simply assign all of the values of self

to all the values of mother as shown in the next statement, but it doesn't work. All the statement does, is make the pointer Mother point to the same place where Self is pointing because we did a pointer assignment. The data that was allocated to the pointer Mother is now somewhere on the heap, but we don't know where, and we cannot find it, use it, or deallocate it. This is an example of losing data on the heap. The proper way is given in the next two statements where all fields of Father are defined by all fields of Mother which is pointing at the original Self record. Note that since Mother and Self are both pointing at the same record, changing the data with either pointer results in the data appearing to be changed in both because there is, in fact, only one field where the data is stored.

In order to Write from or Read into a dynamically assigned record it is necessary to use a temporary record since dynamically assigned records are not allowed to be used in I/O statements. This is illustrated in lines 57 through 63 of the program where some data is written to the monitor.

Finally, the dynamically allocated variables are disposed of prior to ending the program. For a simple program such as this, it is not necessary to dispose of them because all dynamic variables are disposed of automatically when the program is terminated and we return to DOS or the TURBO Pascal integrated environment. Notice that if the "Dispose(Mother);" statement was included in the program, the data could not be found due to the lost pointer, and the program would be unpredictable, probably leading to a system crash.

SO WHAT GOOD IS THIS ANYWAY?

Remember when you were initially studying BIGREC? I suggested that you see how big you could make the constant Number_Of_Friends before you ran out of memory. At that time we found that it could be made slightly greater than 1000 before we got the memory overflow message at compilation. Try the same thing with DYNREC to see how many records it can handle, remembering that the records are created dynamically, so you will have to run the program to actually run out of memory. The final result will depend on how much memory you have installed, and how many memory resident programs you are using such as "Sidekick". If you have a full memory of 640K, I would suggest you start somewhere above 8000 records of Friend.

Now you should have a good idea of why Dynamic Allocation can be used to greatly increase the usefulness of your programs. There is, however, one more important topic we must cover on dynamic allocation. That is the linked list.

WHAT IS A LINKED LIST?

Understanding and using a linked list is by far the most baffling topic you will confront in Pascal. Many people simply throw up their hands and never try to use a linked list. I will try to help you understand it by use of an example and lots of explanation. Examine the program named LINKLIST for an example of a linked list. I tried to keep it short so you could see the entire operation and yet do something meaningful.

To begin with, notice that there are two types defined in lines 4 and 6, a pointer to the record and the record itself. The record, however, has one thing about it that is new to us, the last entry, Next is a pointer to another record of this type. This record then, has the ability to point to itself, which would be trivial and meaningless, or to another record of the same type which would be extremely useful in some cases. In fact, this is the way a linked list is used. I must point out, that the pointer to another record, in this case called Next, does not have to be last in the list, it can be anywhere it is convenient for you.

A couple of pages ago, we discussed the fact that we

had to break the great rule of Pascal and use an identifier before it was defined. This is the reason the exception to the rule was allowed. Since the pointer points to the record, and the record contains a reference to the pointer,

one has to be defined after being used, and by rules of Pascal, the pointer can be defined first, provided that the record is defined immediately following it. That is a mouthful but if you just use the syntax shown in the example, you will not get into trouble with it.

STILL NO VARIABLES?

It may seem strange, but we still will have no variables defined, except for our old friend Index. In fact for this example, we will only define 3 pointers. In the last example we defined 54 pointers, and had lots of storage room. Before we are finished, we will have at least a dozen pointers but they will be stored in our records, so they too will be dynamically allocated.

Lets look at the program itself now. In line 20, we create a dynamically allocated record and define it by the pointer Place_In_List. It is composed of the three data fields, and another pointer. We define Start_Of_List to point to the first record created, and we will leave it unchanged throughout the program. The pointer Start_Of_List will always point to the first record in the linked list which we are building up.

WHAT IS "nil" AND WHAT IS IT USED FOR?

We define the three variables in the record to be any

name we desire for illustrative purposes, and set the pointer in the record to "nil". The word nil is another reserved word that doesn't give the pointer an address but defines it as empty. A pointer that is currently nil cannot be used to manipulate data because it has no value, but it can be tested in a logical statement to see if it is nil. It is therefore a dummy assignment. With all of that, the first record is completely defined.

DEFINING THE SECOND RECORD

When you were young you may have played a searching game in which you were given a clue telling you where to find the next clue. The next clue had a clue to the location of the third clue. You kept going from clue to clue until you found the prize. You simply exercised a linked list. We will now build up the same kind of a list in which each record will tell us where the next record is at.

In lines 27 through 33 we will define the second record. Our goal will be to store a pointer to the second record in the pointer field of the first record. In order

to keep track of the last record, the one in which we need to update the pointer, we will keep a pointer to it in Temp_Place. Now we can dynamically allocate another New record and use Place_In_List to point to it. Since Temp_Place is now pointing at the first record, we can use it to store the value of the pointer which points to the new record which we do in line 29. The 3 data fields of the new record are assigned nonsense data for our illustration, and the pointer field of the new record is assigned nil.

Lets review our progress to this point. We now have the first record with a person's name and a pointer to the second record, and a second record with a different person's name and a pointer assigned nil. We also have three pointers, one pointing to the first record, one pointing to the last record, and one we used just to get here since it is only a temporary pointer. If you understand what is happening so far, lets go on to add some additional records to the list. If you are confused, go back over this material again.

TEN MORE RECORDS

The next section of code is contained within a for loop so the statements are simply repeated ten times. If you observe carefully, you will notice that the statements are identical to the second group of statements in the program (except of course for the name assigned). They operate in exactly the same manner, and we end up with ten more names added to the list. You will now see why the temporary pointer was necessary, but pointers are cheap, so feel free to use them at will. A pointer only uses 4 bytes of memory.

FINALLY, A COMPLETE LINKED LIST

We now have generated a linked list of twelve entries. We have a pointer pointing at the first entry, and another pointer pointing at the last. The only data stored within the program itself are three pointers, and one integer, all of the data is on the heap. This is one advantage to a linked list, it uses very little local memory, but it is costly in terms of programming. (Keep in mind that all of the data must be stored somewhere in memory, and in the case of the linked list, it is stored on the heap.) You should never use a linked list simply to save memory, but only because a certain program lends itself well to it. Some sorting routines are extremely fast because of using a linked list, and it could be advantageous to use in a database.

Following is a graphical representation of what the linked list looks like.

```
Start_Of_List---->John
                  Q
                  Doe
                  Next---->Mary
                           R
                           Johnson
                           Next---->William
                                    S
                                    Jones
                                    Next---->
                                       .
                                       .
                                       .
                                    ---->William
                                         S
                                         Jones
                                         Next---->nil
```

HOW DO WE GET TO THE DATA NOW?

Since the data is in a list, how can we get a copy of the fourth entry for example? The only way is to start at the beginning of the list and successively examine pointers until you reach the desired one. Suppose you are at the fourth and then wish to examine the third. You cannot back up, because you didn't define the list that way, you can only start at the beginning and count to the third. You could have defined the record with two pointers, one pointing forward, and one pointing backward. This would be a doubly-linked list and you could then go directly from entry four to entry three.

Now that the list is defined, we will read the data from the list and display it on the video monitor. We begin by defining the pointer, Place_In_List, as the start of the

list. Now you see why it was important to keep a copy of where the list started. In the same manner as filling the list, we go from record to record until we find the record with nil as a pointer.

There are entire books on how to use linked lists, and most Pascal programmers will seldom, if ever, use them. For this reason, additional detail is considered unnecessary, but to be a fully informed Pascal programmer, some insight is necessary.

CHAPTER 12 - Pointers and Dynamic Allocation

PROGRAMMING EXERCISE

1. Write a program to store a few names dynamically, then display the stored names on the monitor. As your first exercise in dynamic allocation, keep it very simple.

CHAPTER 13 - Units in TURBO Pascal 4.0

THIS IS FOR TURBO PASCAL 4.0 USERS ONLY

If you are using TURBO Pascal version 3.0 or earlier, you will find that none of the programs in this chapter can be compiled or run with your system, but it would be to your advantage to read this material anyway.

When Niclaus Wirth originally defined Pascal, it was intended to be a very small language to be used primarily for teaching programming concepts to computer neophytes. A program would be contained in a single file and compiled in its entirety each time it was compiled. There was no provision for splitting a program up into smaller parts, compiling each part separately, and linking all of the parts together into a final completed package.

Since human beings make mistakes, and because the entire program must be recompiled each time any mistake is discovered, pure Pascal is unsuitable for very large programs. Seeing this problem, many compiler writers have defined some method by which a large program could be broken down into smaller parts and separately compiled.

This chapter will define and illustrate the way Borland has chosen to do so.

PART OF A PROGRAM

Load the program named AREAS and display it on your monitor. This is the first example of a TURBO Pascal "unit" and although it is similar to a program in many ways, it has a few differences which must be pointed out. We will start by pointing out the major sections, then get into the details of each section.

You will first notice that this program begins with the reserved word "unit" instead of our unusal "program", followed by the unit name, Areas. In line 10, the reserved word "interface" is used and all of the statements following it down to the next reserved word "implementation", are part of the interface with any program outside of this unit. The next reserved word used is "implementation" and gives the definitions and executable parts of the private portion of the unit.

Finally, in lines 48 through 50, we find what appears to be a program block just like we have been using all through this tutorial, but actually is not. We will see in a few paragraphs that this is the initialization section and does a very specific job for us even though somewhat different than what we have become used to.

THE INTERFACE PART

Following the unit name we have a section of code in lines 10 through 15 that define the interface of this module to the outside world. Anything defined here is available to the outside world and can be used by any other program provided it has a "uses Areas;" statement in it. Constants, types, and variables could also be defined here, and if they were, they too would be available to any user program, but in this case, only the four functions are made available. It should be fairly obvious that the functions calculate the areas of four different geometric shapes.

THE IMPLEMENTATION PART

From line 16 through line 47 we have the implementation part as delineated by the reserved word "implementation" and the beginning of the initialization block. The implementation part is the actual workhorse of the unit since it contains all of the executable code for the four functions defined above.

Lines 26 through 31 contain the code needed to generate the area of a circle, and this code is no different than the code that would be used if this function were placed in the declaration part of any Pascal program. There is a difference in the function header since the formal parameters are not repeated here. TURBO Pascal allows you to either drop the formal parameters here or include them if you think the code would be more readable. If you include them, they must be exactly as shown in the interface part or you will get a compile error.

A LOCAL PROCEDURE

In lines 20 through 24, we have a procedure that is used within one of the four functions, namely the first. It is really a stupid procedure since it really wastes time setting up linkage for the procedure call and does nothing that couldn't be done just as easy with a simple multiply, but it does illustrate that you can use another procedure within the unit body. The procedure Mult_Two_Numbers cannot be used outside of this unit because it is not included in the interface part of the unit. It is, in effect, invisible to the outside world.

The variable My_Pi would be more correctly represented as a constant but it is defined as a variable to illustrate use of the body of the unit later. Since My_Pi is not defined in the interface part of the unit, it also is

invisible to the outside world and in fact protected from accidental corruption by a misplaced statement in another program. The procedure and the variable for all practical purposes have an impenetrable barrier around them protecting them from unauthorized use by the outside world, but the functions internal to this unit have free access to them just as in any other program.

WHAT IS THE BODY USED FOR?

Lines 48 through 50 constitute the body of the unit and although they appear to consist of another executable program that can be called and used, they actually perform another very specific and useful purpose. It is actually an initialization section and all of the statements in this part of the unit are executed once and only once, and they

are executed when the main program is loaded. This is done automatically for you by the system. There is no way provided for you to call the statements in the body after the program has begun execution. This is why the variable My_Pi was defined as a variable, so we could use this section to initialize it to a useful value.

The body can actually have function and procedure calls that are executed when the program is loaded, as well as loops or conditional statements.

If you would like to execute some statements during initialization and again during the execution of the program one or more times, you can write a procedure or function to accomplish your desires and call it at the appropriate times in the main program.

SELECTIVE NAMING OF FUNCTIONS AND PROCEDURES

If you will study the interface part of this unit you will find that everything you need to use this unit is contained within it, provided that you know enough about plane geometry to understand the functions. You should strive for this understanding in all of your interfaces so that the implementation doesn't even require consultation. Keep in mind, that if you need to, you can include comments to further define the functions in the interface part of the unit.

At this time, you should compile this unit. You will have to compile it to disk rather than only to memory so it will be available for use later in this chapter. You do this by using the menus to change the Compile/Destination to the Disk option. Note that it will not generate an .EXE

file but instead a .TPU file. This is Borland's filename extension for a unit.

ANOTHER UNIT

Load the file named PERIMS for another example of a unit. This is similar to the last except that it does not contain an internal procedure, and it is composed of three procedures that calculate the perimeters of geometric shapes, all of which are visible to the outside world because they are included in the interface part of the unit. Once again, we have a private variable named My_Pi and a block of code (actually a single statement) to initialize the value of My_Pi when the unit is loaded.

Be sure you compile this unit to disk in the same manner as the last and they will be ready for use.

Now that we have several functions and procedures that can be used to calculate the areas or perimiters of several different shapes, we need a program to illustrate their use, so if you load and display the program named GARDEN you will have an example of their use.

HOW DO WE USE OUR DEFINED UNITS?

GARDEN is a very simple program that uses one of the functions and one of the procedures. The only thing you must do is add the names of the units prior to using the external functions or procedures. Lines 16 and 17 each use one of our newly defined routines. As you can see, there is nothing magic about the new routines, and once you include the unit names in a uses statement, the new routines are in a sense, an extension to the Pascal language.

Compile and run this program and see that it really does what you expect it to do.

ONE MORE EXAMPLE OF UNIT USE

Load and display the program named SHAPES4 for another example of using a predefined unit. In line 3, this program includes our new unit named Areas so all four of the area functions are available, and in fact, all four are used

within the body of the program. This program should not be difficult for you to understand and you will be left to study it on your own. You should observe that this program is repeated in chapter 14 in a different form for users of TURBO Pascal 3.0.

CHAPTER 13 - Units in TURBO Pascal 4.0

MULTIPLE USES OF AN IDENTIFIER

Suppose we wanted to move the variable named My_Pi to the interface section in both of the units we defined earlier. Then in the program named GARDEN when we included both of the units in the uses statement, both variables named My_Pi would be available for use so we would have a bit of a problem defining which one we really meant to use. TURBO Pascal has a way to tell the system which one you wish to use by using a qualifier in much the same way that you use a field of a record. The variable name Areas.My_Pi would refer to that variable from the unit named Areas, and the name Perims.My_Pi would refer to the variable from the unit named Perims.

You could even define a new variable of the same name in your main program and refer to it by the qualified name Garden.My_Pi if you chose to. This is not recommended as it would get very confusing to you. The compiler would be very happy to compile and run such a program, because it would not get confused.

WHY USE UNITS?

There are basically three reasons to use units in your

programming. First, some programs are so large that they should be broken up into smaller chunks for ease of handling and reasonable compilation size. In fact some are so large that they cannot be compiled all at one time since TURBO Pascal has an upper limit of 64K of code which can be compiled at once. Most other compilers have a similar limit also.

Secondly, once you complete the code to perform a certain job, you may wish to use the same code in another program to do the same job. If you put the code in a unit, it is ready to simply call and use again. This is becoming a rather important topic in software engineering usually referred to as "Reusable Software".

THIS IS INFORMATION HIDING

Finally, it is sometimes important to hide a portion of code from the rest of the program to assure that it cannot be unduly modified by an error somewhere else in the program. This too is becoming an important area of software engineering and is usually referred to as information hiding.

CHAPTER 13 - Units in TURBO Pascal 4.0

PROGRAMMING EXERCISE

1. Move My_Pi to the interface in both units and change one of the values slightly to see if you can read in the right one at the right time. Define another variable of the same name in your main program and see

if you can differentiate between all three values.

CHAPTER 14 - Complete sample programs

Prior to this point, this tutorial has given you many example programs illustrating a point of some kind, but these have all been "nonsense" programs as far as being useful. It would be a disservice to you to simply quit with only tiny programs to study so the following programs are offered to you as examples of good Pascal programming practice. They are useful programs, but they are still short enough to easily grasp their meaning. We will discuss them one at a time.

AMORTIZATION TABLE GENERATOR

This is not one program, but five. Each one is an improvement on the previous one, and the series is intended to give you an idea of program development.

AMORT1 - This is the bare outline of the amortization program. Although it is an operating program, it doesn't do very much. After some thought and planning, the main program was written to allow for an initialization, then an annual repeating loop. The annual loop would require a header, a monthly calculation, and an annual balance. Finally, a procedure was outlined for each of these functions with a minimum of calculations in each procedure. This program can be compiled and run to see that it does do something for each month and for each year. It has a major problem because it does not stop when the loan is payed off but keeps going to the end of that year. The primary structure is complete.

AMORT2 - This is an improvement over AMORT1. The monthly calculations are correct but the final payment is still incorrectly done. Notice that for ease of testing, the loan variables are simply defined as constants in the initialize procedure. To make the procedures easier to find, comments with asterisks were added. This program is nearly usable. Compile and run it.

AMORT3 - Now we calculate the final payment correctly and we have a correct annual header with column headings. We have introduced a new variable to be used for an

annual interest accumulation. This is neat to have at income tax time. This program can also be compiled and run.

AMORT4 - This program does nearly everything we would like it to do. All of the information needed to build the table for any loan is now read in from the keyboard, greatly adding to the flexibility. After the

information is available, the monthly payment is calculated in the newly added procedure Calculate_Payment. The annual header has a new line added to include the original loan amount and the interest rate in the information. Compile and run this program to see its operation.

AMORT5 - The only additional feature in this program is the addition of a printout of the results. Examining the program, you will notice that many of the output statements are duplicated with the "Lst" included for the device selection. Compile and run this program, but be sure to turn your printer on to get a printout of the amortization table you ask for. If you are using TURBO Pascal version 3.0, you will need to either comment out line 3 or remove it altogether.

TOP DOWN PROGRAMMING

The preceding example is an example of a top-down approach to programming. This is where the overall task is outlined, and the details are added in whatever fashion makes sense to the designer. The opposite is a bottom-up

programming effort, in which the heart of the problem is defined and the rest of the program is built up around it. In this case, the monthly payment schedule would probably be a starting point and the remainder of the program slowly built up around it. Use whichever method works best for you.

The final program AMORT5 is by no means a program which can never be improved upon. Many improvements can be thought of. These will be exercises for you if you so desire.

1. In the data input section, ask if a printout is desired, and only print if it was requested. This would involve defining a new variable and "if" statements controlling all write statements with "Lst" as a device selector.

2. Format the printout with a formfeed every three years to cause a neater printout. The program presently prints data right across the paper folds with no regard to the top of page.

3. Modify the program to include semimonthly payments. Payments twice a month are becoming popular, but this program cannot handle them.

4. Instead of listing the months as numbers, put in a case statement to cause the months to be printed out as three letter names. You could also include the day of the month when the payment is due.

5. Any other modification you can think up. The more you modify this and other programs, the more experience and confidence you will gain.

LIST, to list your Pascal programs

Since the differences between TURBO Pascal 3.0 and 4.0 are significant, two files are included here. If you are using TURBO Pascal 3.0, rename LIST3.PAS to LIST.PAS, and if you are using TURBO Pascal 4.0, rename LIST4.PAS to LIST.PAS before continuing on to the next section.

LIST is a very useful program that you can use to list your Pascal programs on the printer. It can only be compiled with TURBO Pascal because it uses TURBO extensions. The two extensions it uses are the string type variable and (in the case of TURBO Pascal version 3.0), the absolute type variable. The absolute type variable in line 13 and the coding in the Initialize procedure is an example of how you can read in the parameters given on the command line.

If you are using TURBO Pascal 4.0 a completely different method is used in the Initialize procedure which should be no problem for you to understand at this point.

To use this program to print out the last program, for example, you would enter the following at the DOS prompt LIST AMORT5.PAS. This program reads in the AMORT5.PAS from the command line and uses it to define the input file. It should be pointed out that this program cannot be run from a "compiled in memory" compilation with the TURBO Pascal compiler. It must be compiled to a Disk file, and you must quit TURBO Pascal in order to run it from the DOS command level.

The parameter, AMORT5.PAS, is stored at computer memory location 80(hexadecimal) referred to the present code segment. If you didn't understand that, don't worry, you can still find the input parameter in any program using the method given in the initialize procedure for your version of TURBO Pascal.

If you do not have TURBO Pascal, but you are using MS-DOS or PC-DOS, you can still use this program because it is on your disk already compiled as LIST.COM, and can be run like any other .COM or .EXE program.

CHAPTER 14 - Complete sample programs

TIMEDATE, to get today's time and date

This is a very useful program as an example of using some of the extensions of TURBO Pascal if you are using TURBO Pascal 3.0. It interrogates the inner workings of DOS and gets the present time and date for you, provided you entered them correctly when you turned your computer on. The procedure Time_And_Date can be included in any Pascal program you write to give you the time and date for your listings. As an exercise in programming, add the time and date to the program LIST to improve on its usefulness.

The program named TIMEDAT4.PAS does the same thing as the last, but it works with TURBO Pascal 4.0 using the means of defining a DOS call as it has been revised for the newer version. It turns out to be an almost trivial program but is still a good illustration of how to use some of the newer Borland extensions to Pascal.

SETTIME, a useful utility program

This program is very interesting in that it changes the date and time stamp on any file in the current directory. It is the program used to set the time and date on all of the files on the distribution disks included with this tutorial. It sets the time to 12:00:00 and the date to Jan 15, 1988 but you can use it to set any desired time.

You could ask the operator for the desired time and date or use the procedure to get the present date and set the time to noon or whatever time you desire. Its usefulness is limited only by your imagination.

SHAPES3, an example of menus

This program is not very useful, but it illustrates one way to handle menus in a Pascal program, but only if you are using version 3.0 of TURBO Pascal. Chapter 13 included the identical program done slightly differently for use with the TURBO Pascal 4.0 compiler. You can study the structure and imagine many ways a menu can be used to improve the usefulness of your own programs.

OT, The OAKTREE directory program

This program should be very useful to you, especially if you have a hard disk. It will list the entire contents of your hard disk (or floppy) in a very easy to read and easy to use form. The program is documented in OT.DOC, and is precompiled for you in OT.COM in case you are not using

CHAPTER 14 - Complete sample programs

TURBO Pascal. It uses many of the TURBO Pascal extensions and will probably not compile with any other Pascal compiler without extensive modifications.

You will find two versions of the source code for this program, one named OT3.PAS for use with TURBO Pascal version 3.0, and another named OT4.PAS for use with version 4.0 of the TURBO Pascal compiler. You should rename one of them OT.PAS for use with your particular compiler.

The two versions are different in a number of ways. The first version was written for TURBO Pascal version 3.0 over a year ago and was only slightly modified for this new version of the tutorial. The newer version, OT4.PAS, was

modified extensively to use some of the procedures provided by Borland such as GetDate, GetTime, FindFirst, and FindNext. The program for version 4.0 is somewhat smaller since the predefined procedures use fewer characters to perform a given job, and the executable version shows an even greater reduction in size. Apparently Borland has done a very good job in code size reduction with the introduction of version 4.0.

It would benefit you greatly to study the two versions of OT.PAS side by side and compare the benefits of using the predefined procedures.

You will find either program to be a good example of linked lists because it includes a sort routine using a dynamically allocated B-TREE and another sorting routine that uses a dynamically allocated linked list with a bubble_sort. These methods are completely defined in Niklaus Wirth's book, "Algorithms + Data Structures = Programs", a highly recommended book if you are interested in advanced programming techniques.

It might also be pointed out that both OT3.PAS and OT4.PAS also makes use of recursive methods for both sorting and handling subdirectories. It is definitely an example of advanced programming methods, and it would be a good vehicle for your personal study.

Most Important - Your own programs

Having completed this tutorial on Pascal, you are well on your way to becoming a proficient Pascal programmer. The best way you can improve your skills now is to actually write Pascal programs. Another way to aid in your building of skill and confidence is to study other Pascal programs. Many programming examples can be found in computing magazines and books. One of the best books available is

"Programming in Pascal" by Peter Grogono, and another is "Oh! Pascal!" by Doug Cooper and Michael Clancy.

You already own one of the best books available for reference if you are using TURBO Pascal. Although the TURBO Pascal reference manual is worth very little as a learning tool, it is excellent as a language reference manual. Now that you have completed all 14 chapters of this tutorial, you have a good grasp of the terminology of Pascal and should have little trouble reading and understanding your reference manual. Your only limitation at this point is your own perseverance and imagination.

Happy programming.