Turbo Tutor

A Turbo Pascal Tutorial

BORLAND
INTERNATIONAL

The TURBO Pascal Tutor

A Self-Study Guide to TURBO Pascal

PREFACE

Congratulations! Since you have this book, you have undoubtedly joined the ranks of the 200,000+ owners and users of TURBO Pascal. And you've probably bought this book to help you better learn how to use TURBO Pascal. That's what this book is for, and we don't think that you'll be disappointed.

If you're a novice computer user, or novice programmer, then you'll probably want to begin with Part I, *TURBO Pascal for the Absolute Novice*. This will help you to get your feet wet, showing you just how to get started with TURBO Pascal. If you're a more experienced programmer, you may want to skim through these chapters, especially Chapter 6, *Getting Started with TURBO Pascal.*

If you know your way around a computer pretty well, then you're probably ready to dive into Part II, *A Programmer's Guide to TURBO Pascal.* This starts right at rock bottom and builds swiftly, taking you step by step through all the different aspects of Pascal in general, and TURBO Pascal in particular. By the time you get to the end, you should have a solid foundation in Pascal and be able to glean any additional needed information from the **TURBO Pascal Reference Manual** and other books.

Once you feel comfortable with TURBO Pascal, you might take a look at Part III, *Advanced Topics in TURBO Pascal.* This section contains listings of working programs, showing you how to do things like read the directory off a disk or communicate through a serial port. The listings also demonstrate different programming techniques; they're worth studying for that, if for no other reason.

Of course, this book is more than just a book: it comes with a disk as well. The disk is filled with running programs and tutorial information, giving you a ready-made library of routines to copy into your own programs. This is both time-saving and educational, especially as you adapt these routines to suit your needs.

This book, of course, can't replace the **TURBO Pascal Reference Manual**. Rather, this book's goal is to help you grasp the basic principles underlying Pascal and its various aspects; the **Reference Manual** can then help point out the exact definitions of the TURBO Pascal implementation.

Hope you enjoy your exploration. Good luck and have fun!

— Borland International Inc.

Acknowledgements

The following products mentioned in this manual are manufactured and/or sold by the following companies:

The IBM Personal Computer (PC) is a product of IBM. IBM® is a registered trademark of International Business Machines Corporation.

MicroPro® and WordStar® are registered trademarks of MicroPro International Corporation.

CP/M® is a registered trademark and CP/M-80™ and CP/M-86™ are trademarks of Digital Research, Inc.

Microsoft® is a registered trademark and MS™ is a trademark of Microsoft Corporation.

Tupperware® is a registered trademark of Dart Industries Inc.

TURBO™ Pascal and SideKick™ are trademarks of Borland International, Inc.

Third Edition, January, 1985
Printed in the United States of America
9 8 7 6 5 4 3 2 1

ABOUT THE AUTHOR

Frank Borland is more mystique than mystic, as elusive as the Trinity Alps big foot, as shy as the famous Loch Ness monster. Even at Borland International, his namesake, few people have ever seen him. The old-timers recognize him for his remarkable algorithms, still the fastest in the west. Borland lives deep in the Santa Cruz mountains with his transportable computer, his burro, and his dogs. In the early days, he didn't have a permanent homestead, but lived in a couple of camps deep in the redwood groves. Now, Frank has settled down a little, bought a cabin, and is raising a family, thanks to the success of his programming.

These days he is seen even less around town, but still can occasionally be reached by modem.

If you are a Compuserve user, you are closer to Frank Borland than you realize. He is writing either a gothic novel or an epic poem—he hasn't decided which—entirely in bulletin board messages left on different SIGS (Special Interest Groups). But he never uses his real name, and he switches names often, so his writing is hard to follow. Look for messages in cadence, or rhymes. (You can find information on Borland products and a Borland SIG by typing GO BOR from any Compuserve prompt.)

Frank is a warm-hearted person. He wrote Sidekick (one of his latest programming efforts) for humanitarian reasons. Carrying notepads, calculator, and calendar from camp to camp was beginning to stunt the growth of his burro, Lotus, so he wrote Sidekick to make all that unnecessary. He left a note in our mailbox, saying he'd saved Lotus' development.

He rarely talks about his background, or why he chose to abandon normal life and take to the mountains. Some say it had do to with changing the whole motherboard on a PC, just to replace a single chip. Others blame the high price of micro-computer software. We don't really know. Do you?

TABLE OF CONTENTS

PART I

TURBO PASCAL FOR THE ABSOLUTE NOVICE

1. INTRODUCTION

I'd like to begin by thanking you for buying *TURBO Pascal*. Like me, you must be fed up with the high price of computer software, and are ready to get good value for (very little of) your money. Don't be fooled—my software is as good as, if not better than, the high-priced programs on the market, many of which cost 10 times more than mine. As you can imagine, I am quite proud of my programs, especially TURBO Pascal.

But before I forget my manners, let me introduce myself. My name is Frank Borland, and the company which puts out TURBO Pascal was named after me.

I wrote this book so that you could learn to program in TURBO Pascal. You and I will look at the *Pascal* language (that is to say, *standard* Pascal), and then I will teach you how to write good programs in TURBO Pascal.

I live up in the mountains with my family, my dogs, and my burro, Lotus. It is peaceful here. Just the kind of place where a person can sit around and think and put things into proper perspective. Once in a while, when I really want to think through a problem, I take Lotus and hike a few miles from home. There I will camp out for a few days until the quiet and isolation have allowed me to solve my problem. Then I return home to my computer and easily bang out the code that was such a problem only a few days before.

Some people think I am an elusive person, with an aura of mistique surrounding me. Actually, I am just a simple man with simple needs. I keep to myself for the most part, not because I am a loner, but because I got tired of city life, and besides, this is the only way I can write enough good programs to keep the folks at my company happy.

It was while camping under the stars that I got the idea to write TURBO Pascal. The thought occurred to me that people like you wanted to write computer programs with a good programming language, but probably didn't want to pay the high prices companies are charging for their languages. I couldn't think of a single reason that you could not have a complete, high-quality, honest-to-goodness programming language for the price of a pair of shoes or a decent meal. So, I set out to write TURBO Pascal. I think it turned out very well, and I use it myself for most of my daily programming chores.

I don't know how much you know about computers, so I'll just assume that you are like I was when I got my first computer. I couldn't make it do *anything*. (Of course, computers have become easier to use over the years.) I had to rely on terrible manuals with missing information, a bunch of individual pieces that were not designed to work together, and my own patience. Fortunately, I had a few friends who had already been through the initial frustrations, and who were willing to help get me going. It wasn't long before I could do many things with my computer, and was actually looking for more things to do.

Now that I have been through all of that, I am ready to help other people like you to get going with such things as TURBO Pascal. Sometimes all it takes is a certain bit of information presented in the right way to make you see the light. Other times it takes pages of explanation and examples. This book is an attempt to bridge the gap between your knowledge of computers and whatever you need to know to write good programs.

This book starts out slowly, and then builds upon what you have learned as you progress. Before I teach you how to program in Pascal, I am first going to give you a little background on computers and computer programming. I'll also teach you some habits which I have found keep things simple and unconfused.

Next we will look at each of the parts of Pascal and why they are there. We will look at different kinds of numbers and what they do. We will even learn about a logical algebra called *Boolean*.

From time to time we will be looking at the **TURBO Pascal Reference Manual** which came with your TURBO Pascal. While it may be a little confusing for you at this time, it will prove to be a real treasure chest of information when you know more about Pascal. It will also serve you well, because you will outgrow this tutorial pretty quickly.

Besides the **Reference Manual**, there are a few other things you will need. First, is your computer—of course!

Since TURBO Pascal is available for so many different types of computers, I will have to keep this tutorial as general as possible. In other words, I will not be able to give you a lot of specific pointers about your particular computer.

For that reason, you should take a little time to get familiar with how your computer works, if you haven't already. In particular, you should know how to start a program going (we call this *running* a program), save a file, and make backup disks. You should also know where your **CONTROL**, **ESCAPE**, and *Programmable Function* keys are (if you have any).

It will also help you in your learning to have a *quiet* and *comfortable* spot to place your computer. There should be room around it for you to place this book and your **Reference Manual**, and enough light for you to read them easily, as well. Learning *anything* new is hard enough; you should stack as many things in your favor as possible.

Something else you will find *very* handy is a printer. It is almost impossible to program effectively without one. The reason is this:

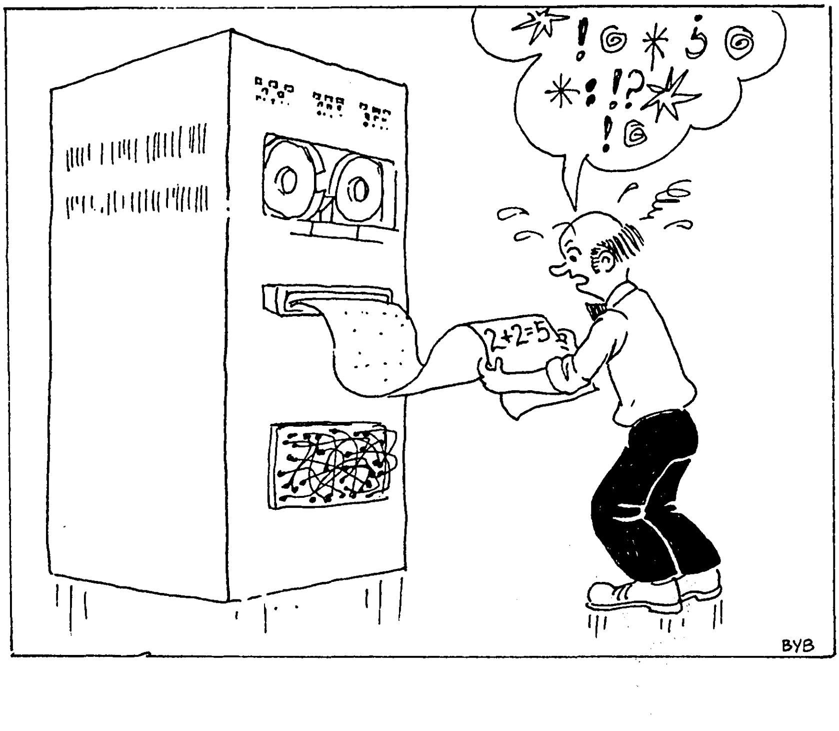
Although TURBO Pascal will find most of your syntactical errors for you—mistakes where you leave out a punctuation mark, or call a variable by a different name—it won't find another kind of error. Every now and then, you will find yourself stumped with a logical *bug*—something in your program which won't let it work the way you think it should. Well, a printed copy of what you have just done can be compared to a printed copy of your earlier work, or traced by hand to see just what you have told the computer to do. Sometimes your mistake will just jump out at you, if you can see your program on a printed page.

Another handy thing about printers and Pascal has to do with the very structure of Pascal. You will find that *big* programs written in Pascal are made up of *little* programs written in Pascal. We call these little programs *procedures*, and we'll discuss them a little later on in this book. The result is that these little Pascal programs (*procedures*) can be stored on your floppy disk for later use, and they can be printed out and stored as printed programs which you could then enter when needed.

Then, when you are working on the solution to a problem that your computer can help you with, you won't have to reinvent the wheel. It very often is the case that you have already written something for another program that will be of use in the program you are working on now. A quick look through your printout file will be more reliable than an anxious scramble trying to remember where you last saw the code that you need.So that's about it for introductions. Let's get on to something I have always found to be important to my understanding of a new matter—a bit of *history*.

2. A SHORT HISTORY OF PASCAL

I can still remember back in the early days of computers (it wasn't *that* long ago) when programs were hand-entered by flipping switches to toggle the state of the computer's electronics. In fact, before my time, the first computers were *mechanical*. To program by flipping switches, you had to know your particular computer's *machine language* and you had to convert everything to *binary* (base 2 numbering system) or some other number representation. And, if that wasn't bad enough, you had to check your completed program by stepping through it and looking at a series of lights.



—An Early Programming Attempt—

You see, computers (that is, *digital* computers) are devices which only understand different patters of two different states. One state is the presence of voltage (often called a "1"); the other state is the absence of voltage (often called a "0"). Telling the computer to do one simple thing like "add 2 + 2" involved entering a series of patterns of ones and zeros by flipping a whole bunch of switches. If the answer came up "5," the poor programmer would have to look at his mess and try to decide where the mistake was. The situation tended to make some early programmers crazy. Boy, those were the days. I am glad they're behind us.

Later, paper tape came along to ease the frustration. The switches remained on the computer, but paper tape readers were added as *input* devices and paper tape punches were added as *output* devices. Little holes were punched in the tape which represented the ones and zeros (ons and offs) previously entered by switches. Punched cards, which were originally designed to automate the weaving industry, and later used to compile the United States census, also were used in a similar manner to paper tape.

Entering machine-oriented digits to represent computer instructions was tedious, time consuming, and error prone. As programmers become more frustrated, they began using a shorthand, English-like method of representing these instructions. They then converted them to machine "code" by associating computer instruction values with their shorthand notation.

Finally, someone had the idea to write a program which would make the computer do the "dirty work" of converting the programmer's shorthand into machine codes. The result was the first *assembler*. This was a huge improvement, and assemblers are still in wide use today; however, the problem is that both the assembler and its assembly-language programs must be specific to a particular computer architecture. You could spend months writing a program for a particular computer, but to use it on another type of computer you would have to learn another assembler language and then spend a lot of time converting your original program to a new one for the other machine. This problem was the reason *programming languages* were invented.

A programming language is nothing more than a program which converts a (supposedly) standard series of instructions into machine-specific code. Therefore, in theory, if you had two computers and had the same language for each (the languages *are* machine specific), you could write a program in the language and it would run on both computers. This is called portability, and is a very important consideration these days.

The United States government Department of Defense (DOD) decided that it needed a portable programming language. This language would be common to all the people (and their computers) writing programs for DOD.

The language they chose was the "COmmon Business Oriented Language", or *COBOL* for short. They chose COBOL because someone demonstrated to them that the same program could be run on two different computers without modifications. Because the government backed it, COBOL has been, and is still, popular in certain circles.

Notice, however, I hinted that the ability to run the same program on two computers was not completely accurate. Most languages have a *standard* set of instructions which will run on any computer. In addition, they have *extensions* or *enhancements* which let you easily take advantage of special features of a particular computer. For example, you may have a computer that has color, graphics, and sound capabilities and another computer that has none of these. Usually, you must stay away from these enhancements if you want to write a portable program.

When scientists learned that computers could be helpful in their calculations, they invented a language which would translate formulas. They called it "FORmula TRANslator", or *FORTRAN*, for short. It is still in wide use today.

This language works pretty well when you have a scientific problem to solve; one which usually involves a small amount of data, but lots of calculations. FORTRAN became such a big deal because International Business Machines (IBM), a manufacturer of computers with world-wide clout, adopted it as their "official" language for their big mainframe computers.

Both these languages, COBOL and FORTRAN, did what they were supposed to do pretty well. They were quite big, however. By this I mean that they needed a lot of room in a computer's memory —the part of the computer which remembers programs and data while the computer is on. There were other languages, of course, but these were the two most well known to Americans.

When *microcomputers* came along in the early 1970's, they had too small a memory capacity to allow these compilers to function. They were also quite involved to learn.

One of the first "simple" languages was the "Beginner's All-purpose Symbolic Instruction Code", or *BASIC* for short. This is a language that anyone can learn in a hurry, if they don't have a very complicated problem to solve. You may already know and use BASIC. Your version may be an *interpreter*-type language, which means that instructions can be executed immediately or interactively. Another attribute of interpreters is that they are small and efficient in their use of memory, and are easy to move from one computer to another. Because the interpreter is essentially a program which emulates a complier and a computer while the program is running, its programs tend to run much slower than equivalent complied code.

Also, a person has to be very good at remembering what is going on in all parts of his program. BASIC is a hard language with which to trace what is happening in a program. It is also hard to trace exactly where it is happening. You can imagine the implications of this, if you put yourself in the place of a person having to fix another's program.

To sum it all up, BASIC is a simple, easy-to-use language that provides a simple solution to simple problems, but which provides complex solutions to complex problems, due to its lack of structure.

The other languages we have discussed are *compiler*-type languages, which means you can only run your program after you have completed writing it and have compiled it into machine language code. TURBO Pascal is a compiler-type language. One of TURBO Pascal's unique characteristics is that it is very small, fast, and easy-to-use. Effectively, it has the speed advantages of an interpreter during program development as well as the execution efficiency of a compiled program.

Pascal was written by a computer genius named Niklaus Wirth in Zurich, Switzerland, in 1970-1971. He had already written several other computer languages such as *PL/1* and *ALGOL*. Dr. Wirth based some of Pascal's concepts upon the work he had done before.

The reason he wrote it in the first place was to teach his students how to program a computer effectively. You see, programming starts with a definition of a *problem*, then breaking the problem down into its *smallest parts*. The last step is to actually *write commands* which the computer will understand, and which will make the computer work until the problem is *solved*.

Professor Wirth was concerned because he knew that a student's first programming language teaches him *habits* in the same way that his first love teaches him about life. It colors his views on a lot of things. Wirth figured it would be better to start a student out with good programming habits.

Pascal is a *structured* language, which means that it is easier to write your progarm in modules by following certain, predefined steps. Certain parts of your program must be placed in certain locations within the program, and must follow certain conventions. I will teach you more about structured programming as we journey through the following chapters.

Structured languages are becoming more and more human oriented and problem oriented. They have the ability to have a very English-like content. At first it won't be apparent, but as you learn to read Pascal programs, you will find that the statements read very much like English sentences. The result is that the problem and its solution is very easy to see, and is machine independent.

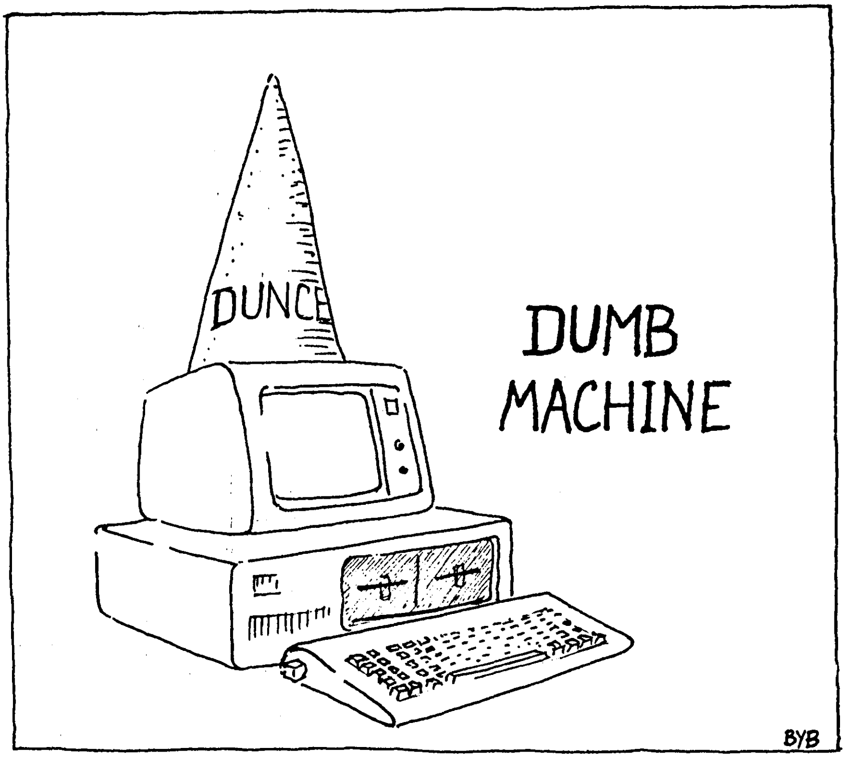
We'll talk more about the technique of programming in the chapter after next. For now, let's look a little more closely at Pascal, and especially TURBO Pascal.

3. WRITING A SIMPLE PROGRAM

Did you know that computers can show up a basic difference between people's learning habits? Many people who sit down at a computer which they have never see before will start right in pressing keys. On the other hand, some people will wait for instructions before touching anything. Even with instructions, it will take these people quite a while before they are comfortable with a computer, while the first type of people were never really uncomfortable in the first place.

Now, the reason I am bringing this matter up here is not to scare you, or make you feel as though "my goodness, this is too difficult." It is because I want to encourage you to sit down and press computer keys, especially if you are the type who is over cautious about such things. This is the way you will learn about your computer most quickly. By the way, today's computers are designed so that with appropriate precautions (such as making backup copies of all your diskettes), it is highly unlikely that you will cause any permanent damage by typing on the keyboard.

You see, a computer, unlike your mind, does not *think* in any real sense. At its most fundamental level it thinks in terms of "on/off", or "yes/no", but **never** "semi-on" or "maybe". It will only follow its *instructions* (whether they make sense, or not) until its electrical power goes off.



Also you should remember that as a human being you have ultimate *control* over your computer: you tell it what to do and when to do it. You do this by pressing keys and giving it immediate instructions, or by pressing keys for instructions which the machine will remember and then execute later on. (By the way, the second example is what we mean when we talk about writing a computer program).

When we work with computers we have to get into two habits. One: we *always* (not sometimes) must be consistent in how we tell the computer to do something. (In other words, the computer is very inflexible.) Two: If we give the computer the wrong information to work with, it won't know the difference, and will try to proceed as usual, causing erroneous and sometimes interesting results. This is what we will call "Garbage in, Garbage out."

In other words, with computers, as with no other part of life, it will always benefit you to take the time to first: "Think It Through" —to pretty well know what you are going to do before you start, and second: "Do It Right The First Time"—so you won't have to do it again.

Well, what do you say we just try a program? Our first program doesn't do much, and you may not understand everything about it right away —but then even driving a car is not second nature at first. If you need to, read the chapter on "Getting Started" in your **Reference Manual** to show you how to write and run a TURBO Pascal program.

By the way, I have written all the special words TURBO Pascal recognizes as instructions in **bold letters**. This makes it particularly easy to read a program and make sense out of it. By the way, these special words are called *reserved words*.

To get started right away, do the following steps:

1. Start your TURBO Pascal program by following the instructions in your **Reference Manual**. Press **Y** when you are asked whether you would like messages included.

2. Press **W** and answer **MYNAME** when asked for the name of your workfile. (Don't forget to press **RETURN** after you have typed in the program's name.)

3. Press **E** to get your TURBO Pascal editor going.

4. Type in the following program, just as it is written here. Don't forget to put in everything, including all punctuation marks and spaces. If you make a mistake, you can use your <Del> key (or whatever key you have defined during TURBO Pascal terminal installation) to backspace and erase the characters you've typed.

```
program            MyName;

const
  TotalTimes = 20;

var
  Name              :  String[25];
  NumberOfTimes     :  Integer;

begin
  Write('What is your name, please: ');
  Readln(Name);
  ClrScr;
  for NumberOfTimes := 1 to TotalTimes do
    begin
      Writeln('Your name is ',Name);
    end;
end.
```

5. When you have finished typing in the program, press your **CONTROL** key and hold it down while you press **K**, then let both keys up and press *D*.

6. Press **S** to save the program you just typed in. This way you will have it later, if you want it.

7. Press **R** to compile the program (don't worry what that means for now) and to run it. You will see a message at the bottom of your screen which you can also ignore for now.

In a moment, the program will run itself.So, just what did this program do?

If you typed it in correctly, and then typed your name correctly when it asked you for it, the program told you your name 20 times. If you did not type it in correctly, or if you did not type your name correctly, all the program gave you was garbage (if anything at all). Remember what I said about "Garbage In/Garbage Out?" (If you did get garbage, you may want to go back and compare what you typed into your computer with the program here in the book, and then try again. If you made a mistake and haven't yet learned the editing commands, it might be easier to retype the entire program at this time, using a new file name.)

Now let's look at the different parts of the program and talk for a moment about each. We will go into more detail about each in a few pages. For now, just try to get a sense of what is going on.

The program, called "MyName", has three main parts:

First: the place where you tell the compiler the name of the program.

```
programMyName;
```

The compiler uses this information to know where all the information it needs begins. It is the same as when you want to give a friend some instructions and you say, "This is where you start." We call this the *Program Header*.

Second: the definition statements:

```
const
  TotalTimes = 20;

var
  Name                  : string[25];
      NumberOfTimes     : Integer;
```

The compiler uses the definition statements as tools when it begins to do what you want it to.

const is short for *constant*, the name of a number which will not change while the program is running. Since the number stays constant, we call it a constant—get it?

var is short for *variable.* Like its name, a variable can contain just about anything, for instance a name, a number, or something you make up.

Third: the program statements themselves:

```
begin
  Write('What is your name, please: ');
  Readln(Name);
  ClrScr;
  for NumberOfTimes := 1 to TotalTimes do
    begin
      Writeln('Your name is ',Name);
    end;
end.
```

The program statements tell the compiler specific steps the computer must take to accomplish its assigned tasks.

An English narrative of this program would go something like this:

1. Start here

2. Ask "What is your name, please?"

3. Read the answer.

4. Erase the computer's screen.

5. Start a process where you write "Your name is" and the answer you read before (in step 3) twenty times, then stop the process.

6. Finish, end, quit, stop, fini.

By the way did you notice that every specific *idea*, or *statement*, ends with a semicolon (;)?This is similar to the way we speak with each other, in sentences, I mean. We express our ideas, each in turn, in sentences; Pascal does the same in "statements". We end our sentences with periods; we end our Pascal statements with semicolons.

And yes, I see that **begin**, and **const**, and **var** don't have semicolons. This is an idiosyncrasy of how the compiler operates.

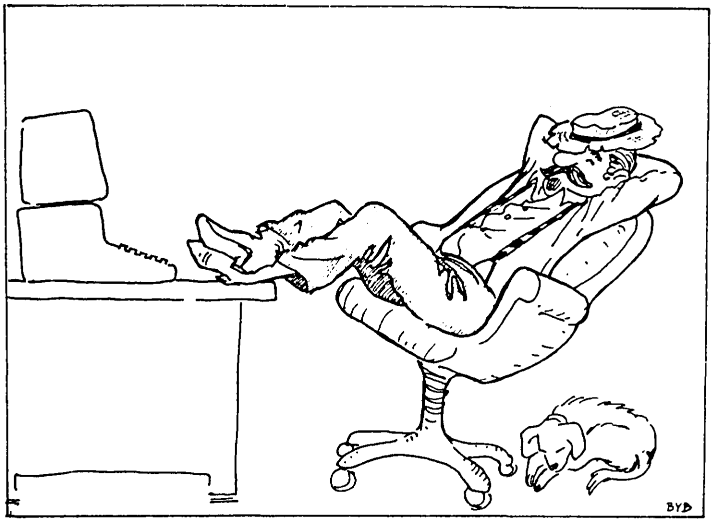
Now let's look at the process of programming, itself, a little more closely in the next chapter.

4. PROGRAMMING IS MORE THAN WRITING A PROGRAM

The process of *programming* a computer is not really about writing a series of instructions for a machine to follow. Rather, it is about *solving a problem*, making life easier, or doing something you want to do. You accomplish these things through the use of a tool called a *computer*—and another tool called your *common sense*.

No tutorial would be complete without my giving you some basic (as in *fundamental*—not as in BASIC) pointers on how to go about programming. That's what I will start to do in this chapter.

By the way, when you read this section and then continue to other sections, I want you to get a sense of what is going on. *Don't try to memorize facts and figures.* Once you get a feel for the basic concepts, you will be able to handle any programming problem.



—Just Sitting Back Thinking—

Computers are pretty consistent machines. If they do what you want them to, they are consistently good. If they don't, they are consistently bad. The problem arises when we don't realize that they do exactly what we tell them to—*whether or not what we tell them is what we really want them to do.*

Now is that confusing enough for you? Don't worry, it will all be clear in a little while. What we need to do now is to get comfortable with some concepts which have to do with computers and programming.

4.1 DATA

Sometimes it seems that no matter where we go, we are bombarded with something called *data*. Data this and data that, but never anything really specific. People seem to forget that what we need to get something done, or to make a decision, is to reduce the data into more manageable pieces. We are going to define this reduced, meaningful data as *processed information.*

When we get right down to it, *data* is the basis from which we derive *information*. In other words, we take raw data, and then manipulate it until we can use it to inform us. Finally, we use this processed information to make a decision.

If we have done the right things—obtained the appropriate data and then interpreted it correctly—our decision will work out for the better. By the way, interpreting data correctly means doing it with a good deal of horse sense; not with blind trust in the data.

Let's say you want to buy a car and then sell it. *Raw data*, in this case, would be the price you could buy it for, and the price you could get for it. *Processed information*, however, is the result of comparing the two. You base your final decision to buy and/or sell on that information.

Now, the color of the car and its make and model may also be items of data which need to be considered. The thing is that they may not really be important (necessary to your decision) in this case, unless you are interested in such things.

If it is a simple decision as to whether or not to have a car, your data can be restricted to simple price considerations. You may, however, also want to consider make, model, color, gas mileage, and so forth, depending upon your needs and desires, and the perceived needs and desires of the buyer.

The bottom line is this: use common sense to decide what data to bother to collect, what data to use, and what data to trust. Let's go back to our prices, for a moment. Let's also say that they are all you choose to consider.

Your two items of raw data (the purchase and selling price) are nothing more than *numbers* until you put a **$** in front of them. The **$** is what we would call a *data label*. It, paradoxically, turns the raw data into *processed information*—sort of.

A data label helps us use the data in a meaningful way. It does so, because it takes a number, or bunch of letters, without apparent intrinsic significance, and jogs our memory.

The data we choose to consider, and which can be manipulated by a computer, comes in two forms: numbers and letters. A computer scientist would call the two *numeric* and *character*.

Imagine, for a moment, that you see a big plant with green leaves and shade in front of you. You store this information in your mind as a picture of what you saw. Later, when someone asks you what it was that you saw, you would say "a tree." In this way, you would convey to them the idea of a big plant with green leaves and shade.

This is very similar to how a computer works, inside its "mind." The difference between you and a computer, is that while you would never confuse a tree with a cat (because each means something different to you), numbers and letters mean absolutely nothing, in themselves, to a computer. It doesn't matter in what combination the letters or numbers may be.

In other words, since the computer does not care what it takes in, we can put in almost *anything*. All we need to do is make the data going in mean something to us. This way it will also mean something to us when we get it back out.

As a practical matter, we divide the data to be stored in a computer under two rules:

1. Each character on your keyboard produces a representative code when pressed. If a character (numbers included) is simply to be stored as its representative code, we call it *character data*. This would include things like names, addresses, phone numbers, and so forth. (Since we often string a bunch of characters together into a word, we will refer to a bunch of characters with a particular meaning, when they are together, as being a *string*. If the character makes sense by itself, we will simply call it a *character*. Simple, isn't it?)

2. If the data consists of numbers which are to be manipulated, as in a calculation, and then returned to you, the computer (complier) coverts the representative codes for the numbers into their actual values, in whichever number base you select (decimal, hexadecimal, binary). We call data in this form *numeric data*. These are things like dollar amounts, inventory counts, and so forth. (A date could fall into either category, depending upon whether or not it is data which requires manipulation.)

One convenient thing about computers is that because the two kinds of data fall into clearly defined categories, they can not be substituted for one another.

Another convenient thing, at least in Pascal, is that sometimes we have to use data in more than strictly numeric or character form. For instance, our data could be more than simply a list of days (Mon, Tue, Wed, Thu, Fri, Sat), or the alphabet (A through Z). Pascal allows us to define our data as being almost anything we want. We will learn more about this later.

4.2 VARIABLES

In order to store data items in a computer, we need *places*—similar to pigeon holes—to put them. We call these places *variables*, and this is how they work:

Let's say we have to store a list of names. The first actual name (Joe Blow) would be stored in a variable—the pigeon hole—called NAME[1].The second name (Dudley Doright) could go into a variable named NAME[2], and so on. If we had no data to go into a variable, it would remain empty.

Variable	Contains Character Data
NAME[1]	Joe Blow
NAME[2]	Dudley Doright
NAME[3]	
NAME[4]	

You will notice that NAME[1] and NAME[2] aren't worth a thing, by themselves. *All they are good for is holding something else.* (NAME[1] is not Joe Blow, it only holds a string of characters which means "Joe Blow"). To make things easier for us, however, in Pascal we call our variables by some name which will give us an idea of what data they are likely to contain.

4.3 STATEMENTS

When we want to convey to one another one thought, or a discrete part of one thought, we use a *sentence*. The same kind of thing in Pascal is called a *statement*. English sentences end in periods; Pascal statements end in *semicolons*.

For now, it is only important that you understand that when I refer to a Pascal program statement, I am speaking about one clearly stated instruction (thought) to the computer.

I will give you more definitions later, as we go along in this tutorial. No need to burden you with more than is necessary, right now.

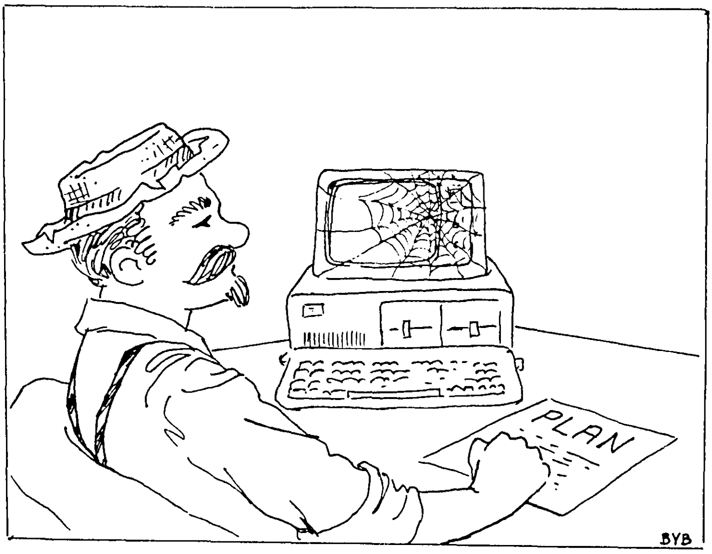
Next we will take a look at the first step in developing a program. So, if you are ready...

5. DEVELOPING A PROGRAM

By this time, I hope you are getting the impression that a computer is no more than a tool. It is a singularly impressive and useful tool, but it is nonetheless a tool in the same way a hammer or saw is a tool. The only reason to use a tool—or a computer—is to *solve a problem*; to make our life easier.

A hammer and saw solve problems for carpenters. After all, it is easier to saw a board into pieces than it is to rip it apart with your hands and teeth.

The analogy can be carried further:A carpenter doesn't simply start in sawing and banging. The first thing he does is to plan and measure. He knows what he is going to do before he does it. *He also knows which tools are appropriate to his plan.*



—Planning, the First Step—

The best way to go about developing a program is to do so *systematically*. Leave the computer alone for awhile, and get out a pencil and a pad of paper. You need to make a list of things to do, and then follow that list as you do the actual program entry, like a pilot following a checklist, or someone walking up a flight of stairs—one step at a time. It is important to write your plan down to make it easy to keep track of.

And, perhaps most importantly, you will want to keep in mind *common sense*. In other words, if you have to go back down a few stairs to pick up something you missed: do it, before you get too far away.

Here are the steps that folks who program computers recommend following:

First Step: Identify the Problem to be Solved

In order to solve a problem, you must first have a very good idea about what the problem is. If your car has a flat tire, the problem is obvious. So is the solution. But then, a computer isn't much help in fixing a flat.

On the other hand, if you suddenly are asked to keep track of all the names and addresses of the families attending your church, a computer would be ideal. The problem is also reasonably obvious.

You will have to store *data* regarding the church members. This data will involve such things as each member's name, address, phone number, and maybe birthday.

Part of your *identification* of the problem should include a description of *why* you have to solve it. It should also include an indication of what *results* you expect to achieve from your solution.

You should also *take your problem apart*, breaking it into its smallest parts. Then you can attack each part of the problem in its turn. When each small part of the problem is solved, and you put all of them together, the large problem will be solved. In computer jargon this is called *Top-Down Design and Stepwise Refinement*.

By the way, this method of program design is ideally suited to TURBO Pascal programming. You will see how when we get into *procedures*, a little farther in this book.

One general note: While you are thinking about the problem to be solved, try to think of a similar problem you have solved before. This may give you insight into your current problem and suggest some possible solutions.

Second Step: Decide What You Need to Get Out of Your System

The only reason to store *data* in your computer is to get it back out again in the form of *information*. The only way to know what *data* to store is to know what *information* you will need later.

The best way to go about it is to *fully describe* each piece of data. You will want to include the data's source, its form (numeric or character), and a name for a variable to hold the data. Finally, you should determine whether or not it makes sense to actually include each particular data item you have planned for.

Third Step: Decide What Data You Need to Put Into Your System

Sometimes people put deciding what data to put into their system in front of the second step. When they do, they soon find themselves "backing down the stairs" to get things right. This is because getting the information out is basic to knowing what is to be put in.

As you are deciding what data to put into your system, make sure that each item is, in fact, available. Then look at it with an idea as to what mistakes people could make as they enter this data.

For instance, a date entered by an American would normally have the form: month, day, and then year (MM/DD/YYYY). A European would more likely put the day before the month as in (DD/MM/YYYY). Part of good programming is to identify possible *errors* before they are made, and to somehow plan around them.

As another example, let's say that your program is going to ask the user to answer a question with either a "yes" or "no." Will you have the user enter "Y" or "N"? What if he instead enters "y" or "n" or "Yes" or "NO" or "no"? Will your program still work correctly. Or, what if the user enters "R"? Will you provide an error message and ask them to enter the answer again?

These are the sorts of things you must think about when designing a program. Fortunately, TURBO Pascal provides easy ways to handle these situations, which we will cover when the time comes.

Fourth Step: Miscellaneous Stuff

This is where you put down *reminders* to yourself about things which don't fall neatly into the other three steps. Things like how many times to do something, how the screen should look, or anything else. I view this step as making notes on a piece of scratch paper.

5.1 PSEUDOCODE

As you are writing your plan on paper, you should state each step in *complete sentences*. Then, as you write the solution to each step, each should also be a complete sentence.This collection of sentences which describe the solution to a problem (in easy-to-read steps) is called *Pseudocode*. The term is derived from jargon for writing a program, which most programmers call *coding a program*, or *writing code*.

This brings up two more computer terms with which you should be familiar:*SOURCE CODE* and *OBJECT CODE*. *Source code* refers to the words you write in the Pascal programming language using your TURBO Pascal editor. This is the information the *compiler* translates into *object code*. *Object code*, in turn, is the information your computer uses to solve your problem. You see, you can read *source code*, while your computer cannot. On the other hand, your computer can read *object code*, while you can't. It is your compiler (the "heart" of TURBO Pascal system) which does the translation from *source code* to *object code*.

Do you remember the program we ran back in Chapter 3? Let's take another quick look at it.

The problem we wanted to solve—one of the world's least significant—was to see our name a number of times on a computer screen. Using the method outlined above, we went about it in the following way:

Problem to be solved: an unreasoning desire to see my name in lights. Since computer letters are in "lights" and are cheaper than a marquee, they will have to do.

Information to get out of the computer: my name, a string of no more than 25 characters, in a variable called "name".

Data to go into the computer: my name, to go into a string variable called *Name*. I get it from my memory, and it is usually readily available.

Miscellaneous Stuff: I want my name to appear 20 times; I'll use a constant variable called *TotalTimes* to hold that number. This way, if I have to change the number of times sometime later, and my program grows bigger with *TotalTimes* appearing in more than one place, I will only have to change one thing to change them all.

I will need one more variable called *NumberOfTimes* to keep track of the question, "Has the computer written my name 20 times yet?"

I will also want to start with a clean screen, but TURBO Pascal has already taken care of that for me with a function (about which we'll talk more later) called *ClrScr*.

Pseudocode:

```
1. Start the program

2. Ask "What is your name, please?"

3. Read the answer into a variable called Name and don't let the
   answer be longer than 25 characters.
```

4. Erase the computer's screen.

5. a. Add 1 to whatever number the integer (which means a single whole number) variable *NumberOfTimes* holds.

   b. Compare the number now held by *NumberOfTimes* with the number held by the constant *TotalTimes*.

   c. If *NumberOfTimes* is less than or equal to *TotalTimes* then write "Your name is" and the string variable Name and go down to the next line on the screen.

   d. If *NumberOfTimes* is more than *TotalTimes*, then stop.

To save your having to turn back to where this program first appeared in this book, here it is again. Compare it with the pseudocode, and see how close they are:..

```
program     MyName;
const
  TotalTimes = 20;
var
  Name                : string[25];
  NumberOfTimes       : Integer;
begin
  Write('What is your name, please:  ');
  Readln(Name);
  ClrScr;
  for NumberOfTimes := 1 to TotalTimes do
    begin
      Writeln('Your name is ',Name);
    end;
end.
```

The only real difference between the two is that the program source code is written in a way that the compiler will be able to read it and then translate it. Other than for that accommodation to the compiler's requirements, it is very close to English.

When you get more familiar with TURBO Pascal, and with good programming habits, you will be thankful for Pascal's similarity to English. You will be able to take almost any Pascal program, written by almost anyone, and figure out what they are trying to do.

Well, that's enough about getting ready to write a program. I could have continued longer, but we have many more issues to discuss. Let's get into the next chapter now and get down to the nitty-gritty . . .

6. GETTING STARTED WITH TURBO PASCAL

Sometimes it seems that the hardest part of doing anything is simply getting started. Then sometimes it seems impossible to get something finished once you have gotten started.

Now we have come to the time when we are ready to really get into TURBO Pascal.

The first thing you want to do with TURBO Pascal is to make a backup copy of diskette you bought. (You did buy it, didn't you?) Then you will put your store bought, or *master*, diskette in a safe place away from kids, stray cats, magnets, and wires with electricity running through them. Keep in mind that, for now, we are talking about the TURBO Pascal diskette; *not* the diskette that came with this book.

I've made some assumptions here, and I hope that they are correct. First, I've assumed that you have learned how to do some of the basic functions with your computer, such as turn it on and off, bring up its operating system (DOS), use the keyboard, make copies of diskettes, and display a directory of files on a diskette. If my assumption is wrong, then get some help or do some reading and learn how to do these things before continuing.

When you ask your computer for a directory of programs on the TURBO Pascal disk, you will notice one named TINST. This is what we call a *general installation* program. It is a program which allows us to use TURBO Pascal on a number of different brands of computers, and yet have it run in the same way on each.

You can skip TINST and go right into writing a TURBO Pascal program, if you want to. But if you are not using an IBM-PC or compatible machine and want to use the built-in program text editor, you will have to run it anyway.

6.1 TURBO PASCAL

To start your adventure in good programming (did he really say that?) you will want to do the following:

1. Insert a backup copy of your TURBO Pascal program disk in drive A:. If you want to keep files of programs you write on another disk, put another blank, formatted diskette in Drive B:.

2. Type the following (note that the word <Enter> enclosed in "angle brackets" means press the **Enter** or **Return** key on your keyboard; do *not* type the word Enter or the brackets themselves):

   TURBO <Enter>

3. When the computer asks whether you want the messages included press:

   Y

   This has to do with mistakes you make when you write your program and then try to compile it.

By pressing **Y** you are telling the computer to tell you what the mistake was, in English. Pressing **N**, for "NO", tells the computer that you would rather look up your mistake in the **Reference Manual** from the mistake's number (we call this an *error code*). You can do this because the computer will tell you the error code number to look up. All in all, it is simpler, right now, to have the computer tell you.

If you have done the foregoing correctly, you will shortly see what we call the *Main Menu*.

6.2 THE MAIN MENU

When we say *menu* we usually mean a list of items from which we can pick one and do something with it. In a restaurant we eat what we pick. With a computer, we either do something with our choice, like run a part of a program, or give the computer some information. With the Main Menu we do the latter. Here is what it looks like:

```
Logged drive:

Work file:     Main file:

Edit      Compile    Run        Save
eXecute   Dir        Quit       compiler Options

Text:       0 bytes
Free:nnnnn bytes

>
```

Let's take each part of the menu and see what it does. We will do so in a more or less logical order, but first... do you notice that some letters are brighter than others? With computers, this usually means that pressing a key with this letter will have some effect different from pressing another key. (Notice that there are no two bright letters the same). That is the case here. When I mention any option from this menu, it will mean that you should press a key which corresponds to a bright letter to indicate that option.

Logged drive:

The logged drive is the disk drive (or hard disk directory) with which you wish to work. For instance, if you have your TURBO Pascal master disk in drive A:, and the programs you are writing stored on a disk in drive B:, then the logged drive should be drive B:. To make it so, do the following:

1. Press

   L

   to choose the Logged drive option on the menu.

2. Press

   B

   to indicate your choice of drive B:.

3. Press

   <Enter>

   to tell the computer to accept your instruction.

But how about with a hard disk, or even a floppy disk with subdirectories? Well, if you're using PC-DOS or MS-DOS, version 2.0 (or later), you would see the following:

```
Logged drive:

Active directory:
```

This shows the *path name* of the current default directory. To change this, you would:

1. Press

   A

   to indicate the Active Directory option.

2. Type in the name of your disk subdirectory. For instance, \PASCAL\PROGRAMS, if you have one directory for TURBO Pascal and another for the files of programs you write.

3. Press

   <Enter>

   to tell the computer to accept your instruction.

Work File:
Main file:

Since these two are so closely related, let's look at them together.

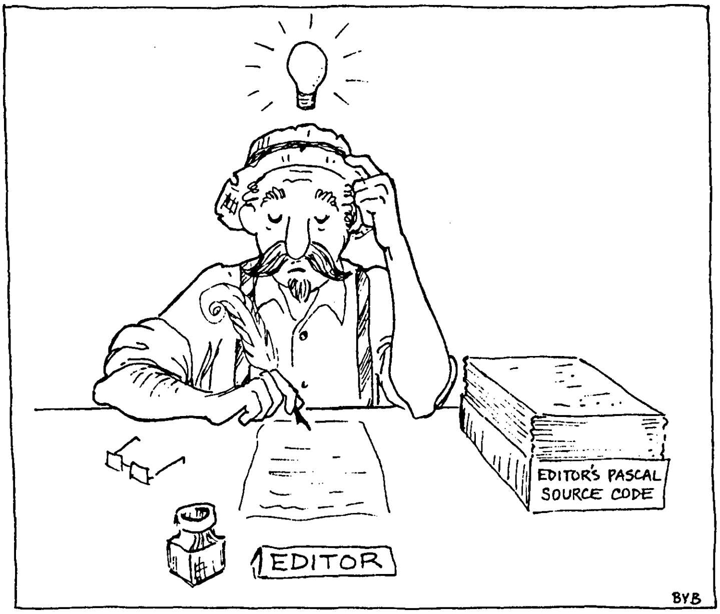
Work file: is the name of the file which is also the program you are *now writing*. If you just enter the name of a file, TURBO Pascal will add the file extension **.PAS** for you. If you have an extension in mind which you would rather have than .PAS, just put it in (TURBO Pascal won't mind).

Main file: has to do with a more advanced programming concept of *INCLUDED FILES*. This is farther along in programming than we will go in this tutorial. Suffice it to say, when you get better (more knowledgeable) at computer programming, the INCLUDE FILES facility will be available to you.

If you ever choose to edit, run, or compile a program without having first indicated a work file, you will be prompted for a work file name.

Edit

There has to be a way for you to actually write your program in a form the computer can use. More importantly, there must be a way for you fix the mistakes you make as you are writing the program as well as after you have written it and tried it out. A correct term for this process is "Editing". To accomplish this process we use an Editor, which is called for our use by pressing <E>.



—TURBO Pascal Editor—

Note

> TURBO Pascal is a *compiler*-type language in which you write your programs using an editor, then compile them so that the computer can understand them. In contrast, most BASIC programming languages are *interpreter*-type languages in which you write your program using the interpreter and the computer translates it as you go. If you have been using BASIC, try to forget everything you know about it, it really won't help you very much with TURBO Pascal.

Writing your program using the TURBO Pascal editor is a lot like writing a note with a word processing computer program called WordStar. If you use that program already, then you are familiar with the various commands which help you accomplish what you would like to do, and you are all set to go. On the other hand, if you don't—you're not; but you are still not out of luck.

In the latter case, you have two choices: either you get familiar with the commands as they are listed in the **TURBO Pascal Reference Manual**, or you "customize" the commands to suit yourself.

The latter is not a hard thing to do. Just follow the directions in the **Reference Manual** Section 1.6.3 *Installation of Editing Commands*.

By the way, I would make my notes in the **Reference Manual** in *pencil*. Believe me, you will want to keep track of what pressing a particular key, or key combination, does. Sooner or later you will change your mind about some command or other. A pencil will make changing your mind a lot less messy.

When you start to edit a program (by pressing **E**), the screen will go blank for a moment while TURBO Pascal is taking care of some housekeeping. Like anything with TURBO Pascal, the moment will be a short one because TURBO Pascal does things so quickly.

What is happening is this: If you have specified the name of a work file which is already on your disk, TURBO Pascal goes to the disk and retrieves your file. Then it turns on the editor and lists out your program. Very shortly, it is ready for you to work with your file. Then you may add stuff, change things around, or do whatever you need to do. (Note that if you had already edited another file, you would have been warned that you should save it before editing another.)

If you have specified the name of a file which is not on your disk, TURBO Pascal will take just a moment to check, and then will send you into the editor so you can start your programming. Once you are done editing, you can save your work to disk. (See **S**ave, below.)

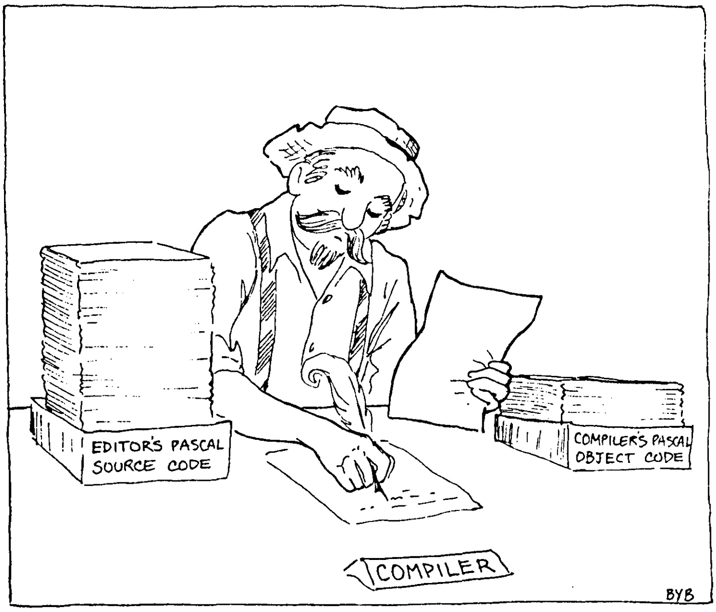
To write a program, once you are "in the editor", is simply a matter of typing away! Oh yes—to get out of the editor and back to the Main Menu, press

```
<Control> KD
```

This means, press and hold down the key marked "Control" or "Ctrl", then press "K", then "D". That's all there is to it.

Compile

The whole object of programming in TURBO Pascal is this:to write code and compile it. Compiling is the step where you find out how well you did. This is where you find out if you wrote the program correctly, and if your program's logic is correct.



—The TURBO Pascal Compiler—

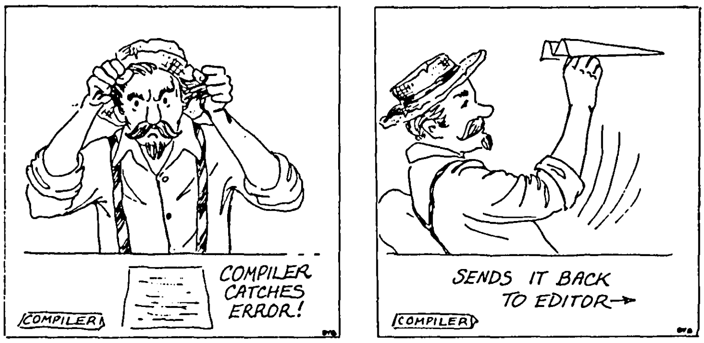
When you press **C**, TURBO Pascal takes your edited (source code) program and changes it into a form the computer can use. This form is called *object code* or *compiled code*. (You will remember we spoke about compiled programs in the section on programming in general).

To go over some jargon again: the compiler takes your *source code*, the program you wrote in TURBO Pascal using an editor, and changes it into *object code*, the form the computer can use.

The TURBO Pascal compiler compiles programs, and then puts them into one of three places: into memory, into a **.COM** file (or a **.CMD** file for CP/M-86), or into a **.CHN** file. Which of the these three to use is up to you and can be changed at any time. You can see how to implement your decision in the section on "compiler **O**ptions", below. Either way, you compile your program by pressing **C**.

Usually you will first do what we call *compile into memory*. You do this when you write a program using the editor and then want to try it out real quick. With the program in memory, you will be able to run it, find a mistake, and then go back to editing it, if you want.

If you have made a mistake which is bad enough that TURBO Pascal can't get around it, the compilation process will stop. Then TURBO Pascal will switch you back into the Editor, tell you what it thinks you did wrong (or what you forgot), and show you where it thinks the problem is.



—Error Handling—

The thing to do now is to fix your mistake and try again. Believe me—the first time you write a complete Pascal program, on your own and then run it will be a high point in your life indeed.

Remember, however, that the compiler will only find a syntax error. Logical errors should already have been eliminated in your planning (it says here). Believe me, it is better that they be found by you in planning, than by your friends when you show your program to them. This method also takes a shorter amount of time, in the long run.

You can compile your program directly into memory by pressing **C** as many times as you want (we'll talk about what that means in just a moment). One thing: if all you do is press **C**, then all you will be able to check is if the program compiles correctly all the way through. This is a good idea, as long as there is the possibility of syntax errors. (Syntax errors are errors such as where you have left out some necessary punctuation, or made a mistake with the spelling of a variable name.)

Run

Once your program has compiled all the way through -or if you are impatient like me, as soon as you are done editing—you will want to press **R** which means "Run the program". With this command, TURBO will see if your program has already been compiled, or even if you have edited it since you last compiled. If so, then it will first compile your source code into memory. Having done that, it will then immediately execute your program's instructions.

compiler **O**ption

Now seems like a good time to clarify the difference between compiling to memory and compiling to a **.COM** file.

Once you are satisfied with the program, you will want to save it in a form where you can use it without first having to call up TURBO Pascal. This is done by compiling your program into a **.COM** file; a program file you can run by simply typing its name.

You choose this option by pressing **O** at the Main Menu. With that, you are presented with another menu which looks like this:

```
compile -> Memory
           Com-file
           cHn-file

Find run-time error      Quit
```

Note that if you are using a CPM-86 version of TURBO Pascal, the **C** option on the screen will be to compile a **.CMD** file. Now, let's look at each of these options.

Memory

The normal, or default, value is the first one: Memory. In other words, without any change on your part, TURBO Pascal will always compile your program into your computer's memory. Once you have made your choice, pressing **Q** will send you back to the Main Menu.

Com-file (or **C**md-file)

If you are at this menu, and you press **C**, for **C**om-file (or **C**md-file), your program will be compiled and placed in a **.COM** file on your logged disk drive. If your work file is called "MYNAME", and you have chosen to compile the program into a **.COM** file, then MYNAME will be saved to disk under the name **MYNAME.COM**.

From then on, you will be able to treat **MYNAME.COM** (or **MYNAME.CMD**) like any other program file. You will be able to copy it to another disk, rename it, delete it, or run it, *all independently of TURBO Pascal.*

The reason you are able to do this is because your MYNAME program is not the only thing which went into the **MYNAME.COM** file. A part of the TURBO Pascal language, called the *library*, was also saved.

A *library* is a collection of programming routines which are used by any Pascal program to work with your computer. They are always there, even if you haven't written them, or ever knew they were there. The programming routines take care of things like putting letters on your screen, calling the Disk Operating System to save information, and so forth. The library will increase the size of your file by about 10 thousand bytes (10 Kbytes) or so.

When you press **C**, you will also be prompted for some additional information. The prompt will look like this:

```
minimum cOde segment size:        XXXX paragraphs (max. YYYY)
minimum Data segment size:        XXXX paragraphs (max. YYYY)
mInimum free dynamic memory:  XXXX paragraphs
mAximum free dynamic memory:  XXXX paragraphs
```

This is an introductory tutorial, and what is going on here is a bit beyond our current efforts. This information is discussed in the **Reference Manual**, if you are interested in finding out more. For now, you can ignore this stuff.

c**H**n-file

The Chain File option is selected by pressing **H** at this menu. A chain file is similar to a **.COM** (or **.CMD**) file, except that it does not contain any of the TURBO Pascal library routines. You would use a chain file when you want to write a Pascal program which will be called to be run from within another Pascal program.

Can you see how the concept of "chaining" came about? One file pulls another, just as one person could pull another by using a real chain.

In order for a **.CHN** file to work, there must have been first a regular **.COM** (OR **.CMD**) file loaded in memory.

The idea that you will have either a **.COM** file or a **.CMD** file (depending upon your operating system) should be obvious by now, so I am not going to keep mentioning both. From now on, I will simply use the term **.COM** file and assume that you know what I mean.

Find run-time error

Do you remember about how TURBO Pascal will find an error in a program in memory when you try to compile it and then run it? Do you remember how it will send you right back into the editor, and show you where it thinks your mistake occurred?

Run Time Error

Well, one of the library routines included in any TURBO Pascal-generated **.COM** file is similar to this. When you run a **.COM** file and a mistake occurs, the program's execution will stop, and you will be presented with a message like this:

```
Run-time error 01, PC=1B56
Program aborted
```

This is telling you that a *Run-time error* occurred, and that the computer has stopped executing your program. A run-time error is one which happens while a program is being executed—or *running*. The number (01, or whatever) refers to what TURBO Pascal thinks the mistake was. You can look up the error, by its number, in the back of your **Reference Manual** (if it is of interest to you).

PC has to do with what is called the *Program Counter*; a thing in the computer which keeps track of just where the computer is in the computer program at any time. Its function is similar to your following the street names on a map with your thumb, as you go down an avenue. It is written in *hexadecimal* (base 16) math. (You could translate it into decimal (base 10) math by using **SideKick**, another one of my programs available from Borland International.)

When a *PC* number appears, you should make a quick note of it. Later, when you have TURBO Pascal going, and the program in question loaded, you can find the line where the error occurred. You do so by getting into the compiler Options menu, pressing **F** (for **F**ind run-time error), and then answering with the *PC* number you made a note of when you are prompted for it. The compiler will point out where in the source program the error occurred.

Quit (the compiler Options menu)

Anytime you start something, you will eventually want to quit. It's the same with this menu. The only difference is: when you press **Q** to leave this menu, you will only go back to TURBO Pascal's Main Menu.

Save

One of the smarter things you will do in programming, is to **SAVE ALL YOU DO!!!!** Is that clear?

SAVE early and often. The reason for this is that sometime, somewhere, when you least expect it, the power to your computer will be interrupted. When that happens, all the work you have done since your last *save* will go into never-never land. Like the Good Book says, "A word to the wise, ..."

Dir

Every now and then, you will want to be reminded of the name of the file you were working on the last time. If it is on the current logged drive, pressing **D** will allow you to find out. Or you may simply want to have a list of the files you have stored on disk. Pressing **D** will show you those files as well.

What happens when you press **D** is that you are prompted for a *mask*. If you wanted to see only the files ending in **.PAS**, for instance, type in an asterisk (*) and a "dot-pas" like this:

***.pas** <Enter>

All the files on the currently logged drive ending in **.PAS** will be listed for you. If you want all the files to be listed, simply press:

<Return>

Remember, only the files on the logged drive (and the active directory) will be listed. You will have to change the logged drive (see pressing **L**, above) before pressing **D** to get a listing of the directory, if it is different from the one you are currently working with.

A hint: I usually press **D** to get a listing of the **.PAS** files on my drive. Then, while they are still on the screen before me, I press **W** for work file and enter the name of the file I want while I can see its name before me. (If you know what wildcards are, you can use * to match any group of characters or **?** to match any single character in the mask.)

e**X**ecute

On some implementations of TURBO Pascal—that is "on some kinds of computers running TURBO Pascal"—you are able to run another (non-TURBO Pascal) program from within TURBO Pascal. You do so by pressing **X** and then entering the name of the program you want to run. When it is over (finished running) you are automatically brought back into TURBO Pascal.

Quit

All good things have to come to an end, every now and then. To leave TURBO Pascal to get back into your computer's disk operating system (DOS), press **Q**. If you haven't saved your work file, you'll be asked:

```
Workfile MYNAME.PAS not saved. Save (Y/N)?
```

to which you should almost always reply **Y**.

Memory Space

The message at the bottom of the menu:

```
Text:      0 bytes
Free: nnnnn bytes
```

These are two more pieces of information available to you, and which are best discussed together.

You are allowed from about 32 thousand characters (letters, numbers, and symbols) less than your computer's memory capacity in your program files. That means that if you have a 64K computer, your program can be up to 32 thousand characters long. If you have more memory, your programs can be substantially larger. If that makes you nervous, this chapter has about 20,000 characters in it, so far. If your program gets so big that it won't fit in the allowed space, you can break it into smaller, more manageable, pieces which can then be c**H**ained together. This part of the menu, then, lets you know how long your source file is, and how much room you have left to work with.

6.3 CONCLUSION

Well, that ends *TURBO Pascal for the Absolute Novice*. You're now ready to start learning how to program in Pascal, which, of course, is the point of the next part of the book, *A Programmer's Guide to TURBO Pascal*. I'm happy to have been your guide; I hope that you've enjoyed it as much as I have. You've done fine up until now, and I'm sure you'll zip right through the rest of the book. Me? Well, the dogs are chasing the burro (or is it the other way around?), the chickens are out of their yard, and I've got a few programming projects of my own that need to be finished, so I'll see you around . . .

PART II

A PROGRAMMER'S GUIDE TO TURBO PASCAL

7. THE BASICS OF PASCAL

"Things are always at their best in their beginning."

—Blaise Pascal, Lettres Provinciales, no. 4

Okay. Either you skipped Part I, or you're a fast reader. Whichever it is, you're about to get a quick introduction to Pascal. First, you're going to learn how to write a simple Pascal program by converting a simple BASIC program. After that, you'll learn some Pascal terms which you'll be using throughout the rest of the book. You'll undoubtedly have lots of questions as you go along; rest assured that most (if not all) will have been answered in great detail by the time you finish Part II.

7.1 A QUICK EXAMPLE

If you've gotten this far (and you obviously have, or else you wouldn't be reading this), then you should know the fundamentals of how to use TURBO Pascal. (That is, you should know how to use the parts of the Main Menu; I don't yet expect that you can write a program.) If my assumption is not correct, go back through Part I again. Of course, you won't really need that for the following example, since it's just a penciland-paper exercise anyway, but you will need it before you go too much farther into Part II.

Having gotten that out of the way, let's take a simple BASIC program and convert it into a simple Pascal program. Suppose our BASIC program looks like this:

```
100  REM SIMPLE BASIC PROGRAM
110  INPUT A
120  INPUT B
130  C = A + B
140  PRINT C
150  STOP
160  REM END OF SIMPLE BASIC PROGRAM
```

This program allows you to enter two numbers, which are stored in variables *A* and *B*. These two values are then added together and stored in *C*. The contents of *C* are then printed out so that you can see the sum of the two numbers that you entered. So far, so good. Now comes the fun part: converting this innocuous little piece of code into a real, live Pascal program. You're going to do it a step at a time, so that you can understand what you're doing and why. As usual, the explanation is more complicated than the actual process.

The first step is to remove all the line numbers, since you don't need them. Even if you *wanted* them, you would still remove them, since Pascal uses other means of getting around than jumping to a given line. Your program now looks like this:

```
REM SIMPLE BASIC PROGRAM
INPUT A
INPUT B
C = A + B
PRINT C
STOP
REM END OF SIMPLE BASIC PROGRAM
```

Next, you must change the names of the input/output (I/O) commands. Instead of *INPUT*, you will use the Pascal **procedure** *ReadLn*, and you will replace *PRINT* with the **procedure** *WriteLn*. Now you have:

```
REM SIMPLE BASIC PROGRAM
ReadLn(A)
ReadLn(B)
C = A + B
WriteLn(C)
STOP
REM END OF SIMPLE BASIC PROGRAM
```

Next, three more changes. First, you will replace the equals sign (=) in the fourth line with a colon followed by an equal sign (:=). Pascal uses := for assigning values and reserves the plain = for comparing values.

Your second change is to put a semicolon (;) between all *executable statements*. Since a Pascal statement can (and often does) occupy several lines, the semicolon marks where one statement stops and the next one begins. For neatness' sake, you'll place the semicolon at the end of the preceding statement.

Lastly, you will eliminate the command *STOP*, which is not needed or used in Pascal (and isn't really needed in your BASIC version here).

Your program has changed a little more:

```
REM SIMPLE BASIC PROGRAM
ReadLn(A);
ReadLn(B);
C := A + B;
WriteLn(C)
REM END OF SIMPLE BASIC PROGRAM
```

Now for a major change. In BASIC, you can start execution on any line, selectively execute groups of lines currently in memory, and so on. In contrast, a PASCAL program is a well-defined group of instructions, with what is known as the *main body*. Execution always begins at the start of the main body and proceeds through to the end. So you need to tell Pascal that your program is a **program** and then place your instructions in the main body. After you do this, it will look like this:

```
program Simple;
begin
  REM SIMPLE BASIC PROGRAM
  Readln(A);
  ReadLn(B);
  C := A + B;
  WriteLn(C)
  REM end OF SIMPLE BASIC PROGRAM
end.
```

On to the next item. First some background: Pascal was designed as a teaching language and, like most teachers, will pick nits (but only for your own good, mind you). Case in point: BASIC allows you to create variables as you go along, while Pascal demands that all (and I do mean **ALL**) variables be declared *before* they can be used. So let's declare our variables:

```
program simple;
var
  A,B,C       : Integer;
begin
  REM SIMPLE BASIC PROGRAM
  ReadLn(A);
  ReadLn(B);
  C := A + B;
  WriteLn(C)
  REM end OF SIMPLE BASIC PROGRAM
end.
```

Only one thing remains to make your simple BASIC program into a simple Pascal program: change the comment statements. In BASIC, each comment line starts with *REM* (for *REMINDER*). In Pascal, a comment starts with the character { and ends with the character }, and you can have as many lines of comments between those two characters as you want. If your terminal doesn't have { and }, or if you just don't like curly braces, then you can use the sequences **(*** and ***)** instead. You will now add a few comments to ensure that your program is well documented. And here's your final program:

```
program Simple;
{
        A simple Pascal program converted from BASIC.
         DATE:       17 June 1985
         AUTHOR:     Put your name here
}
var
  A,B   :  Integer;{   input variables   }
  C     :  Integer;{   output variable   }
begin   {   main body of program Simple   }
  ReadLn(A);
  ReadLn(B);
  C := A + B;
  WriteLn(C)
end.  {   of program Simple   }
```

Here it is, an honest-to-goodness Pascal program, guaranteed to compile and run under TURBO Pascal! It really doesn't look all that different from your original BASIC program. And you thought Pascal was hard!

Here's a review of the steps you used to translate your BASIC program into Pascal:

1. Delete all the line numbers
2. Replace I/O statements
3. Replace assignment = with **:=**
4. Add semicolons at the end of statements
5. Drop *STOP* at the end of the program
6. Add **program** Name / **begin...end.**
7. Declare all variables in the **var** section
8. Convert *REM* statements

It would be convenient if this set of rules was enough for all such translations. Unfortunately, this isfar from true. In fact, these rules will only translate a small subset of all BASIC programs. Moreover, most BASIC programs over a certain size cannot be translated at all, only rewritten. However, your purpose here is not to show you how to convert BASIC programs into Pascal; it is to show you that you can learn how to write simple Pascal programs in just a few minutes.

7.2 SOME PASCAL TERMS

You're going to learn about some terms that will be used throughout the book. Some of them (such as *identifier*) can be fully explained before you leave this chapter. Others (such as *statement*) will have their definitions expanded as you go through the book, so don't get upset if our first discussion doesn't seem to be all-encompassing.

7.2.1 Characters

Let's start at bedrock. Your program will be composed of letters, digits, spaces, and other printing *characters*. These are known as *ASCII characters* (ASCII stands for American Standard Code for Information Interchange, if you care. It is a computer-industry standard way of representing all of the possible characters on your keyboard).

There are 128 standard ASCII characters (numbered 0 though 127), and the table at the end of this chapter shows you all of them. Notice that the first 32 have strange 2-and 3-letter abbreviations. These are known as *control characters*, because they're used to control certain computer operations. For example, if your program tried to write ASCII character 7 (BEL) on the screen, you would hear a beep instead (on an old teletypewriter, you would hear an actual bell ring, hence the name BEL, get it?).

Other commonly used control characters are BS (backspace), CR (carriage return), and ESC (escape). You can type most of these characters from your keyboard by typing the corresponding character in the third column of the table (starting with character 64, "@" and going through character " _ ") while holding down the **Ctrl** (or **Control**) key. What the computer (or, more accurately, the program currently running in the computer) will do with them is another question entirely.

Control characters are usually written with **Ctrl-** or with an up-arrow front. For example, character 7 (BEL) would be written:

Ctrl-G or

^G

By the way, there's another non-printing character in the ASCII set, that is, one with a value greater than 31. See if you can find it. (Hint: it's not number 32—that's really a space; that is, the character you get if you hit the space bar, which is actually a printing character to a computer.)

Now on to the printing characters. As you can see from the table, the ASCII character set contains all the letters of the alphabet, in both upper and lower case. The upper case letters start at code 65 and go to 90, while the lower case ones start at 97 and go to 122. A little math shows that the ASCII value of any lower case letter is exactly 32 greater than its upper case equivalent. Remember that—it can come in handy. The ASCII set also has all the decimal digits (0 through 9), starting at code 48. Letters and digits together are collectively known as *alphanumeric characters*.

Of course, there are plenty of visible characters besides just the alphanumerics. Starting at 32 (which, as I said, is a space:" "), you have several punctuation and other special characters in addition to the letters and digits. These are usually called *special characters*. The alphanumerics and the special characters together are known as the *printing characters*, and they plus the control characters form the entire ASCII character set.

To summarize: characters 0 through 31 are the control characters; along with character 127 (did you find it?), they form the non-printing characters. Characters 65 through 90 ('A'...'Z') and 97 through 122 ('a'...'z') are the letters, while characters 48 through 57 ('0'...'9') are digits. Letters and digits together are known as alphanumeric characters. All other characters not already mentioned make up the special characters.The alphanumeric and the special characters together make up the printing characters. And the printing and non-printing characters form the entire ASCII character set.

Well, almost the entire set. Many computers recognize characters 128 through 255 as special printing or non-printing characters. For example, the IBM PC uses those codes to represent foreign characters, graphics characters, and special symbols. However, since those are **not** standard for all makes of computers, they really can't be listed here. Just be aware that you can have characters with values greater that 127.

7.2.2 Identifiers

Okay, now that you completely understand ASCII characters, you can learn about *identifiers*. In Pascal, many things have names: programs, constants, data types, variables, procedures, functions, you name it (if you'll pardon the expression). According to Standard Pascal, an *identifier* must start with a letter and is then followed by zero or more letters and/or digits. TURBO Pascal also allows you to use the underline character "_" (ASCII 95) anywhere you can use a letter, which means anywhere.

The case (capitalization) of letters is irrelevant, so that the identifiers *HOTSTUFF*, *HotStuff*, and *hotstuff* are all the same. Note, though that the underline character *does* make a difference; *hotstuff* and *hot_stuff* are two *different* identifiers. As for length, well, you can also make identifiers as long as you like (just about). The actual limit is 127 characters (letters, numbers, and underscores), which should be long enough to please anyone.

7.2.3 Reserved Words

Pascal uses some special identifiers for putting programs together. You used four of them in your quick example above: **program, var, begin**, and **end.**These identifiers (and the others listed below) are called *reserved words*. That means you can't use them to name programs, constants, data types, and so forth. For example, you couldn't declare a variable named *Program* or call a program *Begin*. You can only use them in the way Pascal (and Niklaus Wirth) decrees. Here's a list of all the reserved words (sometimes called **keywords**) in Standard Pascal:

and	**array**	**begin**	**case**	**const**	**div**
do	**downto**	**else**	**end**	**file**	**for**
function	**goto**	**if**	**in**	**label**	**mod**
nil	**not**	**of**	**or**	**packed**	**procedure**
program	**record**	**repeat**	**set**	**then**	**to**
type	**until**	**var**	**while**	**with**	

And here are some reserved words that TURBO Pascal uses as well:

absolute **external** **inline** **shl** **shr** **string**
xor

If they all look strange now, don't worry; you'll know them well by the time you finish this book.

7.2.4 Symbols

Many special characters are used by Pascal for various purposes. You've already seen how the semicolon (;) separates statements, the sequence colon-equals (:=) assigns values to variables, curly braces ({ and }) delimit comments, and the period (.) signals the end of the program. Other commonly used symbols are

* / + -	arithmetic and set operations
= > < >= <= <>	comparison operations
[ ]	array and set delimiters
( )	function and procedure parameter lists
^	pointer and file operations
.	record field delimiter

Certain reserved words also function as symbols:

not and or xor shl shr	logical operations
div mod	integer arithmetic operations
in	set inclusion

7.2.5 Constants

You often need to use a specific, fixed value of some sort when writing a program. For example, if you were solving some geometrical problems, you might want to use the value of Pi (3.1415926...). Such a value is called a *constant*, since it can only have one value. (The opposite of a constant is a *variable*, which is an identifier that can have more than one value. More about those later.) A constant can be one of (at least) seven types:

TYPE	**EXAMPLES**		
Integer	3	0	-17382
Byte	3	0	255
Real	2.71828	0.00	-6.67E-22
Char	'A'	'$'	'0'
Boolean	true	false	
set	[0..9]	[]	['a','r','u']
string	'STARS.TXT'	''	'Amt in $'

Also, the reserved word NIL is the constant value of an *unassigned pointer*. This probably means nothing to you now, and you can safely forget it until Chapter 17, which talks all about such things.

Incidentally, the first five types shown are the standard *predefined data types* of TURBO Pascal (more on those in Chapter 9). *Sets* are a type of data structure; we'll cover these in Chapter 16. *Strings* will be discussed in Chapter 13.

7.2.6 Variables

Sometimes you need to work with values that are unknown or that might change while the program is executing. To do this, you make up a name (*identifier*) and declare it to be of some *data type*. For example, in the sample program above, you declared the identifiers *A, B,* and *C* to be of *type* Integer. These identifiers are known as *variables* (since their values can vary—get it?). Here are some more examples of variables:

```
program SampleVariables; {
        purpose         show different types
                        of variables
}
var
  Value1,Value2,Sum          :  Integer;
  Radius,Circumference       :  Real;
  Selected,Done              :  Boolean;
  Answer,Initial             :  Char;
begin
  Value1 := 63; Value2 := 228;
  Sum := Value1 + Value2;
  Radius := 40.25;
  Circumference:=2.0 * Radius * Pi;  {Pi is predefined as 3.1415926536}
  Selected := True;Done         := not Selected;
  Answer := 'Y';Initial         := 'd'
end. {  of program SampleVariables  }
```

You can, of course, type in, compile, and run this program. Be warned though: since there's no input or output (I/O, for short), you won't see anything happen. A safe program, but dull.

7.2.7 Expressions

Just like constants and variables, *expressions* of different types can be created. An expression of a given type contains constants and/or variables of that *same or compatible types*, along with appropriate *symbols* (if needed). When the program encounters an expression, it *evaluates* it, that is, it combines all the constants and/or variables and comes up with a single value of the appropriate type.

In your first sample program, you have the integer expression *A* + *B*. When the program evaluates this expression, it gets the value of *A* and adds to it the value of *B*. The result is some integer value, which is then stored in *C*. Likewise, in your second sample program, you had the integer expression *Value1* × *Value2*, and the real expression *2.0* * *Radius* * *Pi*.

See, it's not as bad as you thought. Really, you've probably seen many expressions before; in your science and math classes, they were called *formulas* (or, if you want to be classical, formulae). Things like the circumference of a circle or the velocity of a falling object with respect to time are just expressions (of the type Real, usually). In fact, FORTRAN (one of the earliest programming languages) stands for FORmula TRANslator. So, if it helps you, just think *formula* every time you use the term *expression*.

7.3 REVIEW

Each of these topics deserves (and will get) more discussion. I wanted to define enough basic terms to let you start talking about Pascal itself... priming the pump, so to speak. You should now have some grasp of what ASCII characters, identifiers, symbols, constants, variables, and expressions are. If you do, you're ready to go on to Chapter 8 and learn about program structure and statements. If you don't, reread this chapter and go on to Chapter 8 anyway. It'll come with time.

One more item before we leave Chapter 7 is the ASCII table I promised you several pages back.

ASCII Character Set Table

0	NUL	32	(space)	64	@	96	`
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	HT	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	<	92	\	124	|
29	GS	61	=	93	]	125	}
30	RS	62	>	94	^	126	~
31	US	63	?	95	_	127	DEL

8. PROGRAM STRUCTURE

Pascal programs are composed of *statements*. That sounds simple enough, and it is, but it's a little like saying the English language is composed of the 26 letters of the alphabet. This statement is also true, but it's also a trifle simplistic. **Statements** describe the actions to be performed by the computer. They can be as simple—or as complex—as *your* knowledge, *your* needs, and *your* creativity. You have to start somewhere, and it's best to start simple and with the two *sections* essential to **all** Pascal programs. They are:

1 The declaration section
2 The program body

To show the basic structure of Pascal, you need to set up a simple program:

```
program Simple;
var
  A,B,C        :  Integer;
begin
  Write('Enter two numbers:    ');
  ReadLn(A); ReadLn(B);
  C := A + B;
  WriteLn('The sum is ',C)
end. { of program Simple }
```

This program writes the prompt:

```
Enter two numbers:
```

on the screen. The program waits for two numbers to be entered, then computes and prints to the screen

```
The sum is <sum>.
```

This sequence of events will occur only once for each execution of the program.

8.1 THE DECLARATION SECTION

The declaration section consists of a series of statements.

8.1.1 The PROGRAM Statement

The first line of your program is the **program** statement with the identifier *Simple*. TURBO Pascal programs can (but do not need to) begin with a **program** statement. This statement contains the reserved word **program** followed by the name of your program. (A *reserved word* is one that only Pascal can use; that is, you couldn't name some variable **program**).

Standard Pascal requires a list of file parameters (*input, output*), but TURBO Pascal just ignores such a list (as, indeed, it ignores the whole statement), so we'll also ignore it from now on. A semicolon separates the **program** statement from the rest of the program. Here are more examples:

```
program BudgetAnalysis;

program K;

program WithAVeryLongNameIndeed;
```

8.1.2 Declaration Statements

In your sample program, you used three variables: *A*, *B*, and *C*. Variables are named locations in memory containing a value that can be changed during program execution. You'll learn more about them in Chapter 9, but for now just think of them as shoeboxes or pigeonholes that hold numbers or other information.

Pascal demands that all (and that means *all*) variables must be *declared* before they can be used. *Declared* means that each variable must have its name (A,B,C, for example) and type (Integer, for example) defined before it is used. In fact, this is such a key element of Pascal that you need to call special attention to it. So *memorize the following rule*:

THE GREAT UNDERLYING RULE OF PASCAL:

All identifiers must be declared before they are used.

This rule (which I'll call **The GURP**, for short), applies to *all* identifiers: constants, data types, variables, subprograms, and any other use of an identifier. Right now, though, you're only concerned with variables.

Variable declaration is done between the **program** statement (**program** Simple;) and the main body statement (begin...end.). To signal the start of the variable declaration statements, you use the reserved word **var**, then name your variables and give their type(s). Each statement in this section contains a list of one or more variable names, followed by a colon, then the variable type, and ending with a semicolon. Given these requirements, Pascal allows you flexibility in formatting these sections. It would be equally correct to declare your variables this way:

```
var
   A,B,C   :  Integer;
```

or

```
var
   A       :  Integer;
   B       :  Integer;
   C       :  Integer;
```

As mentioned back in Chapter 7, there are five predefined data types in Pascal:Integer, Byte, Real, Char, and Boolean. The first four are just what they sound like:*integer* numbers (0, 5, -18232); *byte* values (0 through 255); *real* or floating point numbers (0.0, 3.14159, -35.232e-5); and *ASCII* characters ('A','-','3'). *Boolean* variables have either of two values—*True* or *False*—and are used to store the truth value of some proposition so that it can be tested later .Here are some variable declarations for these different data types:

```
var
  Alive,Breathing,Conscious   :  Boolean;
  Age,Height,Weight           :  Integer;
  Score,Tries                 :  Byte;
  Ratio,Percentage            :  Real;
  First,Middle,Last           :  Char;
```

Note that a *declaration* statement (or any other kind of statement) doesn't have to fit on just one line. If you wanted to be expansive, you could rewrite this as:

```
var
  Alive,
  Breathing,
  Conscious
  :  Boolean;
  Age,
  Height,
  Weight
  :  Integer;
  Score,
  Tries
  :  Byte;
  Ratio,
  Percentage
  :  Real;
  First,
  Middle,
  Last
  :  Char;
```

You would have lots of room to put a comment after each variable name. You would also have very long programs. Here's yet another variation:

```
var  Alive  :  Boolean; Breathing  :  Boolean; Conscious  :  Boolean;
  Age  :  Integer; Height  :  Integer; Weight  :  Integer;
  Ratio  :  Real; Percentage  :  Real; First  :  Char;
  Middle  :  Char; Last  :  Char;
```

8.2 THE PROGRAM BODY

The main body of a Pascal program consists of the statement **begin...end**, with any number of executable statements between these words. Execution always starts with the first statement after

begin and proceeds to the last statement before **end** (or, at least, tries to). Notice that there may be several such **begin...end** pairs, and that only the final **end** is followed by a period (.). This is the *only* place where a period should follow **end**.

Between the reserved words **begin** and **end** in our sample program are five executable statements:

```
Write('Enter two numbers:   ');
ReadLn(A);
ReadLn(B);
C := A + B;
WriteLn('The sum is  ',C)
```

Note that a semicolon appears at the end of the first four statements. Pascal doesn't use lines to distinguish between statements—just semicolons. More precisely, Pascal uses semicolons to *separate* statements.You could have written your program this way:

```
program Simple ; var A,B,C : Integer
; begin Write('Enter two numbers:   ') ; ReadLn(A)
; ReadLn(B) ; C:=A+B ; WriteLn('The sum is   ',C) end.
```

or this way:

```
program SIMPLE
;
var
A,B,C : Integer
;
begin
Write('Enter two numbers:   ')
;
ReadLn(A)
;
ReadLn(B)
;
C := A + B
;
WriteLn('The sum is   ',C)
end.
```

As you can see, semicolons are only used when it's necessary to show where one statement ends and the next begins. You can also see that Pascal also doesn't care about line indentation, upper and lower case, and (sometimes) spaces. You have the freedom to format your programs in a variety of ways. The best policy is to pick a consistent, readable, and easy-to-use format, then stick with it. The two alternate styles shown are neither readable nor desirable to use, so we'll stick with the regular TURBO format.

Now let's look at the five executable statements, one at a time.

```
Write('Enter two numbers:  ');
```

The command Write is really a *predefined procedure* telling your program to write whatever is enclosed in the parentheses out to the screen. It's one of a set of *standard procedures* and *functions* that Pascal defines for you. Two tables at the end of this chapter list the *predefined procedures* and *functions* that TURBO Pascal provides. You'll meet most of these as you tackle the appropriate topics in this book.

As mentioned, *Write* will print its contents on the screen. (It will do much more that that, as you will see in Chapter 18.)In this example, you have text -a *string constant*. Upon execution of this statement, the message:

```
Enter two numbers:
```

will appear on the screen. The program will then wait with the cursor two spaces away from the colon. It's waiting because the next statement is

```
ReadLn(A);
```

As you can guess, *ReadLn* is another *standard procedure* that causes the program to wait for input from the keyboard (and more; again, see Chapter 18). Here, since *A* is an *Integer* variable, it's waiting for you to enter an *Integer* number. When you type in a number, then press the **Enter** key, the program assigns that number to the variable *A* in the parentheses. After you do this, the program will wait again, because the next statement is

```
ReadLn(B);
```

It's waiting for another number, followed by **Enter**. Since the program is waiting for **Enter**, you can correct any mistakes you might have made in typing the number when you do so before pressing **Enter**. Your program won't accept (read) data until **Enter** is pressed.

Your program executes the next two statements on its own. With the statement

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

the program evaluates the expression *A* + *B*. In other words, it adds the contents (value) of *A* with that of *B*. It then gives that value to the variable *C*. This is known as an *assignment statement*, since it assigns some value to a variable. Let's suppose that, when prompted by the program, you had entered the values *21* (for *A*) and *35* (for *B*).

After execution of this statement, the variable *C* would contain the value **56**. Note that Pascal uses **:=** to assign the value on the right to the variable on the left. By contrast, Pascal uses = to compare two values (IF A = B THEN...). Be careful not to confuse these two types of statements. You'll learn more about comparisons in Chapter 10.

After the assignment statement, we come to our last standard procedure call:

```
WriteLn('The sum is ',C);
```

This prints on the screen the message

```
The sum is
```

followed by the value of *C*. For example, if you had indeed entered *21* and *35* for *A* and *B*, then the message would be

```
The sum is 56
```

Since you used *WriteLn*, the cursor will be moved down to the start of the next line after the message has been written.

The procedures *Write* and *WriteLn* will print out any number or combination of messages and values enclosed in the parentheses. Expressions (formulas) can be included as well. You could eliminate the assignment statement and get the same message by changing the last statement to be

```
    WriteLn('The sum is   ',A + B);
```

All separate messages and values within the parentheses must be separated by commas.

8.3 COMMENTS

At this stage, you need to add one more element to your program: *comment statements*, to ensure that your program is well documented. A comment statement starts with a left curly bracket ({) or the sequence **(***. It then continues for as many lines as you wish. It ends a right curly bracket (}) or the sequence ***)**, depending upon how you started it. Everything within the comment statement is ignored. (Well, almost everything. See the discussion of compiler options in the **TURBO Pascal Reference Manual**.) Here's our simple, yet well-documented program:

```
program Simple;
{
     purpose          adds two user-entered values and
                      prints out the sum
     last update      21 April 1985
}
var  { variable declaration statements }
  A,B               :  Integer;       { input variables }
  C                 :  Integer;       { output variable }
{  main body of program Simple }
begin
  Write('Enter two numbers:   ');
  ReadLn(A);
  ReadLn(B);
  C := A + B;
  WriteLn('The sum is   ',C);
end { of program Simple }
```

Notice that you didn't have to put a comment symbol on each line of the program description between the program statement and the variable declarations. Instead, you just put the *start comment* symbol ({), went on for as long as you wanted, then finished with the *end comment* symbol (}).

One thing you have to really watch for is *nested comments*. Let's say that you've written the following section of code:

```
for Indx := 1 to SysMax do
  with Starmap^[Indx] do begin
    Write('SYSTEM:    ',Name);             { write system name }
    Write(' Population:    ',Pop);         { show population   }
    if Indx = CurStat.Loc.System           { check location    }
      then Write(' <current location> ');
    WriteLn
  end;
```

Don't worry if you don't understand the code; just notice the comments to the right of three statements. They're a little redundant, but they'll help illustrate the problem. Now, suppose that you wanted to temporarily remove the Write statement showing the population as well as the **if...then** statement; that is, you didn't want to delete them from the program, but you didn't want them to execute, either. You could simply put comment symbols before and after the statements:

```
for Indx := 1 to SysMax do
  with Starmap^[Indx] do begin
    Write('SYSTEM:    ',Name);             { write system name }

{   temporarily cut from program

    Write(' Population:    ',Pop);         { show population   }
    if Indx = CurStat.Loc.System           { check location    }
      then Write(' <current location> ');

}

    WriteLn
end;
```

Your problem is solved, right? Wrong. The comment will start o.k., but it will end when it finds the } after the words *show* population. The **if...then** statement will be as "visible" as it was before, and the following }, trying to end a comment that's already done, becomes an illegal symbol. And, if you simply deleted the comment { show population }, the same problem would crop up with the comment { check location }.

There are two ways of handling this problem. The messy way is to do something about any comments inside the one you're creating (known as *nested* comments). You could delete them, but that means you would have to re-enter them when you "uncommented" those statements. That would be easy in this case, but what if you needed to "comment out" a large section of the program? Deleting and then re-entering all those comments (assuming, of course, there are any) could be tedious. Another, similar solution would be to change the *end comment* symbol (}) in each of those comments to something else. For example, you could replace them with ~ (tilde):

```
{   temporarily cut from program
Write(' Population:    ',Pop);                 { show population ~
if Indx = CurStat.Loc.System                    { check location ~
  then Write(' <current location> ');
}
```

This works fine, but you must be sure to change each and every one back when the code is "un-commented." Why? Because if you miss one, the comment will merrily continue throughout your program until it finds an }.

An error like this can cause real headaches. If you're lucky, it will mess up your program enough so that it won't compile, and you'll be forced to find the error immediately. If you're not, the program will compile just fine, and you won't see any problem until you execute it. If you're really unlucky, the program will appear to work just fine, and you won't discover that there's a problem until after the first 5000 copies have been shipped.

There is a cleaner way of handling this problem, however. Remember, you can use two different sets of delimiters for comments: {,} and **(*,*)**. What's more, TURBO Pascal allows you to nest one kind of comment within the other. *Allow* isn't quite the right word; it's a natural result of the comment definition. If you start a comment with {, it will ignore everything it comes across, including **(*** and ***)**, until it finds **}**. The reverse is true with **(***; that is, it will ignore { and }.

The solution, then, is to consistently use one type of comment —say, {,} — for all program comments and use the other type — in this case, **(*,*)** — for blocking out sections of code. Your example would now look like this:

```
for Indx := 1 to SysMax do
  with Starmap^[Indx] do begin
    Write('SYSTEM:   ',Name);              { write system name }
(*temporarily cut from program
    Write(' Population:   ',Pop);          { show population   }
    if Indx = CurStat.Loc.System           { check location    }
      then Write(' <current location> ');
*)
    WriteLn
  end;
```

A few more words about comments. Not only can they extend across many lines, they can be stuck in anywhere. As a demonstration of this, we present your example program with comments stuck in everywhere. I'm not sure why you would want to do this to a program, but it's good to have the freedom if you ever need it.

```
program Simple{  file parms:  }(input,output);
var{iables}
  A{input},C{input},C{output}       : Integer;
begin {  main body of program  }
  Write{ to screen }('Enter two numbers:    ');
  ReadLn(A); ReadLn(B);
  C {sum} := A {1st value} + B {2nd value};
  WriteLn{to screen}('The sum is ',C)
end. {  of pro(* bye! *)gram Simple  }
```

Well, *almost* anywhere. Had you inserted comments in the strings *'Enter two numbers:'* and *'The sum is '*, they wouldn't have been comments. Instead, they would have been printed to the screen along with the other text within the single quote marks.

You now know enough to start creating your own Pascal programs. A quick reading of the next two chapters, especially the first section of each, will give you even more information to play around with. Feel free to do so; you can always come back to this section later.

8.4 ADVANCED PROGRAM STRUCTURE

Program structure in Standard Pascal follows a definite format. Some sections can be omitted, but no rearranging is allowed. However, rearranging *is* allowed by TURBO Pascal. Here is the complete format of a Standard Pascal program:

```
program Name(file parameters);

label
  < label declarations >
const
  < const declarations >
type
  < type declarations >
var
  < variable declarations >

  < subprograms >

begin
  < main body of program >
end.
```

TURBO Pascal is more flexible than Standard Pascal in its program structure. The **label**, **const**, type, and **var** sections can be placed in any order and can occur more than once. They also can be mixed with subprograms. This allows logical grouping of definitions, such as:

```
const
   XMax      =  8;
   YMax      = 10;
   ShipMax   = 15;
type
   XRange    = 1..XMax;
   YRange    = 1..YMax;
var
   Sector    : array[XRange,YRange] of 0..ShipMax;

const
   .
   .
   .
```

A second advantage of this freedom is the ability to define *typed constants*, which will be discussed below.

As you've seen, a program starts with the *program statement*. The program statement is followed by the *declaration section*. Here you define any labels, constants, data types, and/or variables which will be used throughout the program. Each section begins with the appropriate reserved word (**label**,and so forth), followed by one or more corresponding declarations. Here's a brief description of each kind:

label : one or more unsigned integers (0..9999) or identifiers, separated by commas, with the whole list ended with a semicolon:

```
label
   10,ProcExit,4213,Error;
```

const : an identifier and a constant value, separated by an equals sign (=) and ended with a semicolon; also, a typed constant (see below):

```
const
   XMax        = 8;
   Yes         = True;
   Answer      = 'Y';
   G           = 6.673E-08;
```

type : an identifier and a type definition, separated by an equals sign (=) and ended with a semicolon:

```
type
   XRange        = 1..XMax;
   language      = (BASIC,FORTRAN,Ada,Pascal,
                    FORTH,C,COBOL,ALGOL);
```

var : one or more identifiers separated by commas, followed by a colon and a type identifier or type definition, and ended with (you guessed it!) a semicolon:

```
var
   A,B,C         : Integer;
   Flag          : Boolean;
   Choice        : language;
   X             : XRange;
```

The **const-type-var** sequence allows definitions to cascade down, so that you can change an entire program by merely changing a few constants and recompiling. Combined with the ability to repeat that sequence, this gives you great power and flexibility in defining your data structures.

8.4.1 Typed Constants

TURBO Pascal lets you declare what it calls *typed constants.* These aren't really constants, though they are declared in the **const** section. Instead, they are pre-initialized variables. Suppose you had two flags, *Flag1* and *Flag2*, that you were going to use throughout your program. *Flag1* needed an initial value of True, while *Flag2* needed one of *False*. You might do the following:

```
program Whatever;
var
   Flag1,Flag2    : Boolean;
begin
   Flag1  := True;
   Flag2  := False;
   .
   .
   .
end.
```

This is fine, but if you have many such variables, it can consume space and time to do all of the initialization. With TURBO Pascal, you could instead do this:

```
program Whatever;
const
   Flag1          : Boolean = True;
   Flag2          : Boolean = False;
begin
   .
   .
   .
end.
```

Flag1 and *Flag2* can still be used as variables; that is, you can still assign values to them through the course of your program. However, they will now start out with the desired values and will not have to be explicitly initialized.

For those of you who know something about Pascal, yes, you can have typed constants for data structures such as arrays, strings, sets, and records. All the details can be found in the **TURBO Pascal Reference Manual**, Chapter 13.

8.4.2 Subprograms

Between the definitions and the main body of the program you place any subprograms (procedures and functions) that you might define. I'll talk more about these in Chapter 11, but a brief explanation isn't out of place. A subprogram has exactly the same structure as a program, except that it starts with a

procedure or **function** statement and ends with a semicolon (instead of a period). A subprogram is executed just by mentioning its name along with any parameters it might take. Pascal includes a set of *predefined subprograms*; that is, procedures and functions that already exist without you having to define them. You saw some in the example earlier in the chapter: *Write*, *WriteLn*, and *ReadLn*. The two tables at the end of this chapter list the standard subprograms of TURBO Pascal. As a quick example, here's our sample **program** with the main body converted into a **procedure** named *CalculateSum*:

```
program Sample;
{
      purpose      to demonstrate a subprogram
      last update  28 August 1985
}
procedure CalculateSum;
{
      purpose      read in two values and print the sum
      last update  28 August 1985
}
var
   A,B,C            :  Integer;
begin {  main body of procedure CalculateSum  }
   Write('Enter two numbers:    ');
   ReadLn(A); ReadLn(B);
   C := A + B;
   WriteLn('The sum is ',C)
end; {  of proc CalculateSum  }

begin {  main body of program Sample  }
   CalculateSum
end. {  of program Sample  }
```

When the program runs, it only has one statement to execute: *CalculateSum*. That tells it to execute the statements in the main body of the **procedure** *CalculateSum*. When it's all done, it goes back to the main body and, having no more statements, stops. You can do an awful lot with subprograms, but I won't tell you what until Chapter 11.

8.4.3 Block Statements

Following all the *subprograms* is the *main body* of the program itself, consisting of the *reserved words* **begin** and **end**, with some number of *statements* between them. This is just a special case of what is known as a *block statement*. A block statement has the sequence **begin...end**, with zero or more statements (*separated by semicolons!*) in between. A block statement can be used anywhere a regular statement can be used, and several places where a regular statement can't. For example, the main body of a *program* is just a block statement followed by a *period*, while the main body of a *subprogram* is a block statement followed by a *semicolon*. The real advantage of block statements shows up in *control structures* (see Chapter 10). For example, the **if...then** statement is defined as

```
if <Boolean expression>
  then <statement>
```

But you often wish to execute more than one statement when a given condition is true. Solution:use a block statement, like this:

```
if Max < Min then begin { force min <= max }
  temp      := max;
  max       := min;
  min       := temp
end;
```

You also used a block statement back in the section on comments. You were using a **with** statement, which allows you to easily see the fields of a record data type. It has the format:

```
    with <identifier(s)> do <statement>
```

The statement here was the block statement:

```
begin
  Write('SYSTEM:     ',Name);
  Write(' Population:    ',Pop);
  if Indx = CurStat.Loc.System
    then Write(' <current location> ');
  WriteLn
end;
```

You can nest block statements; that is, have block statements within block statements. For example, you could have a block statement in the **if...then** statement, yielding:

```
with Starmap^[Indx] do begin
  Write('SYSTEM:    ',Name);
  Write(' Population:    ',Pop);
  if Indx = CurStat.Loc.System then begin
    Write(' <current location> ');
    LocationFound := True
  end;
  WriteLn
end;
```

Simple, huh? You'll use block statements a lot throughout the book, especially after Chapter 10.

That's all for advanced program structures for now. Before continuing to the next chapter, here are the two tables I promised you back in the early parts of this chapter. First is a table of *predefined procedures*, followed by a table of *predefined functions*.

PREDEFINED PROCEDURES

Assign(F,N)	Assign file F to file name N
BDos(R)	make BDOS call with regs R (CP/M-86)
BDos(F,P)	make BDOS call to F with P (CP/M)
Bios(F,P)	make BIOS call to F with P (CP/M)
BlockRead(F,D,N)	read N blocks from D to file F
BlockWrite(F,D,N)	write N blocks from D to file F
Chain(F)	chain to file F
Close(F)	close file F
ClrEol	clear to end of current screen line
ClrScr	clear entire screen
CrtExit	send terminal reset string
CrtInit	send terminal init string
Delay(M)	delay M milliseconds
Delete(S,P,L)	delete section of string S
DelLine	delete current screen line
Dispose(P)	recover memory used by P^
Erase(F)	delete file F

Execute(F)	execute file F
FillChar(V,L,D)	fill V with data D for L bytes
GetMem(P,I)	allocate I bytes of RAM for P^
GotoXY(X,Y)	move cursor to X,Y (1,1=upper left)
Halt	stop program execution
HighVideo	sets high intensity display mode
Insert(S,D,P)	insert string D into S
InsLine	insert line on screen
LongSeek(F,P)	special Seek routine (MS-DOS)
LowVideo	switch to "dim" video output
Mark(P)	mark heap pointer at P^
Move(S,D,L)	move L bytes from S to D
MSDos(R)	make call to MS-DOS with regs R (MS-DOS)
New(P)	create memory for P^
NormVideo	switch to normal video output from dim
Randomize	init random seed (see Chapter 20)
Read(P1,...)	read items in from keyboard
Read(F,P1...)	read items from file F
ReadLn(P1,...)	as "read", but moves to new line at end
ReadLn(F,P1...)	ditto, but from file F (textfile only)
Release(P)	reset heap pointer to P^
Rename(F)	rename file F
Reset(F)	open file for input
Rewrite(F)	open file for output
Seek(F,P)	move to record P in file F
Str(N,S)	convert number N to string S
Val(S,N,P)	convert string S to number N (error at P)
Write(P1,...)	write items out to screen
Write(F,P1...)	write items out to file F
WriteLn(P1,..)	as "write", but starts new line at end
WriteLn(F,P1...)	ditto, but to file F (textfile only)

PREDEFINED FUNCTIONS

Abs(A)	absolute value of A (Real, Integer)
Addr(V)	address of variable V (Pointer)
Addr(SubP)	address of subprogram SubP (Pointer)
ArcTan(X)	arctangent of X (Real)
BDos(F,P)	does BDOS call; returns reg A (CP/M)
BDosHL(F,P)	ditto; returns HL pair (CP/M)

Bios(F,P)	does BIOS call; returns reg A (CP/M)
BiosHL(F,P)	ditto; returns HL pair (CP/M)
Chr(I)	character with ASCII value I (Char)
Concat(S,..,S)	concatenation of strings (String)
Copy(S,P,L)	substring at P length L (String)
Cos(X)	cosine of X (Real)
EOF(F)	end-of-file test on file F (Boolean)
EOLn(F)	end-of-line test on file F (Boolean)
Exp(X)	exponential of X (Real)
FilePos(F)	current record in file F (Integer)
FileSize(F)	total records in file F (Integer)
Frac(X)	fractional portion of X (Real)
Hi(I)	upper byte of I (Integer)
Int(X)	integer portion of X (Real)
KeyPressed	keyboard status flag (Boolean)
Length(S)	length of string S (Integer)
Ln(X)	natural logarithm of X (Real)
Lo(I)	lower byte of I (Integer)
MemAvail	bytes/paragraphs available (Integer)
Pos(P,S)	position of str P in str S (Integer)
Ptr(I)	pointer to address I (Pointer)
Odd(I)	odd/even test of I (Boolean)
Ord(Sc)	ordinal value of scalar variable (Integer)
Pred(Sc)	predecessor of scalar value (same type)
Random	random value from 0.0 to 0.999... (Real)
Random(I)	random value from 0 to I-1 (Integer)
Round(X)	rounded-off value of Real (Integer)
ShL(I)	Shift left (Integer)
ShR(I)	Shift right (Integer)
Sin(X)	sine of X (Real)
SizeOf(V)	size in bytes of variable V (Integer)
SizeOf(T)	size in bytes of data type T (Integer)
Sqr(A)	A * A (Real, Integer)
Sqrt(A)	square root of A (Real)
Succ(Sc)	successor of scalar value (same type)
Swap(I)	I with upper, lower bytes swapped (Integer)
Trunc(X)	truncated value of X (Integer)
UpCase(C)	C converted to uppercase (char)

9. PREDEFINED DATA TYPES

In the first two chapters of this part of the book, we've referred to constants, variables, expressions, and data types. Well, now I am going to teach you more about them. Let's start with variables.

9.1 VARIABLES

Think of Tupperware®. Think of pages and pages of different plastic containers, designed to hold a variety of items. Some are designed for liquids, others for dry goods, yet others for lunch meats. Similar or identical containers can hold similar or identical items. Items can be transferred from one container to another.

In much the same way, you can use *variables* in your programs. You can declare lots and lots of variables, designed to hold different kinds of information. Some are designed for numbers, others for characters, yet others for logical values. Similar or identical types of variables can hold similar or identical values. One variable can receive the value of another.

Actually, the Tupperware® analogy is weak in a few spots, but it should convey the basic idea:variables hold stuff for later use. And, instead of buying them at a home party, you get create variables by *declaring* them. Remember our sample program?

```
program Sample;
var
  A,B,C,        : Integer;
begin
  Write('Enter two numbers:   ');
  ReadLn(A); ReadLn(B);
  C := A + B;
  Write('The sum is ',C)
end. { of program Sample }
```

For your program, you needed three variables: two to hold the values you typed in, and one to hold the sum of the other two. So you created them by declaring them. You declared them by (1) listing their names (*identifiers*), and (2) saying what type of variables they were (in this case, **Integer**). You did all this in the variable declaration section, which started with the reserved word **var** and ended when you reached the main body of the program. Here you get back to the Great Underlying Rule of Pascal (or, GURP): *All identifiers must be declared before they are used*. In your case, it means that **A**, **B**, and **C** must all be declared before you can use them in your program. Suppose you added a variable to your program:

```
program Sample;
var
  A,B,C        :  Integer;
begin
  Write('Enter two numbers:   ');
  ReadLn(A);
  ReadLn(B);
  C := A + B;
  D := A -B;
  WriteLn('The sum is ',C);
  WriteLn('The difference is ',D)
end. { of program Sample }
```

Question: is this an acceptable Pascal program? Answer: **no!** Why not? Because you're using an identifier, *D*, which you haven't defined. If you were to compile this program, you'd get an error stating "Unknown identifier" when the compiler reached the statement *D* := A –B. How do you fix it? Like this:

```
program Sample;
var
  A,B,C,D        :  Integer;
begin
  Write('Enter two numbers:   ');
  ReadLn(A);
  ReadLn(B);
  C := A + B;
  D := A -B;   WriteLn('The sum is ',C);
  WriteLn('The difference is ',D)
end. { of program Sample }
```

Now *D* has been properly defined as an integer variable, the same as the others used. But why did you decide to make them all of type Integer? And just what does that mean, anyway? Let's look at the four basic data types of Pascal so see if you can get an answer.

9.2 INTEGER

You know what *integers* are: they're counting numbers (0, 1, 2, 3, ...) with negative numbers (–1, –2, –3, ...) thrown in. The largest integer you can work with depends upon the amount of space used for integer variables. TURBO Pascal allocates two bytes, giving a maximum positive value of 32767 (known as *MaxInt*). The maximum negative value is –(*MaxInt*+1). So, TURBO Pascal allows the following range of values for integers:

-32768 -32767 -32766 . . . 0 . . . 32765 32766 32767

Remember your discussion of constants and expressions? An *integer constant* is simply a string of digits (no commas, please) with an optional - (minus sign) in front. An *integer expression* (formula) contains integer constants and/or integer variables and/or integer operators. The result of an integer expression is some integer value. Your sample program has two such expressions:

a + b	add the values of a and b
a – b	subtract the value of b from that of a

You usually find expressions on the right side of assignment statements (as in your sample program). The program *evaluates* the expression using the current contents of any variables involved and then stores the resulting value in the variable on the left side of the :=. The integer operators that can appear in an expression are:

=	addition
-	subtraction (and unary minus)
*	multiplication
div	division
mod	remainder

Addition, subtraction, and multiplication all work just like you'd expect them to. The rest may need a little explaining, though. Let's start with **div** and **mod**. Remember when you first learned division? When you divided 4 into 11, you didn't get 2.75. You got 2 (the quotient) with a remainder of 3. Why? You were doing integer division, which required that the results also be integers. Well, that's what **div** and **mod** do:*A* **div** *B* returns the quotient, while *A* **mod** *B* returns the remainder. Maybe the following examples will help:

```
 8 div 4 = 2        8 mod 4 = 0
 9 div 4 = 2        9 mod 4 = 1
10 div 4 = 2       10 mod 4 = 2
11 div 4 = 2       11 mod 4 = 3
12 div 4 = 3       12 mod 4 = 0
```

Two other things to keep in mind about **div** and **mod**. First, if *A* is less than *B*, then *A* **div** B will equal *0*. In other words, while *200* **div** *200* equals *1*, *199* **div** *200* equals *0*. Second, if *B* equals *0*, then both *A* **div** *B* and *A* **mod** *B* will result in some sort of error when they are executed. Make sure that you keep both situations in mind when working with these integer operations.

O.K., you've got integer constants, integer variables, and integer operators, and you can combine these into integer expressions. You're all done, right? Well, consider the following program:

```
program Sample;
var
  A,B,C       :  Integer;
begin
  A := 10;
  B := 5;
  C := 5 * A + 2 * B;
  Write('C = ',C)
end. { of program Sample }
```

Question: what value will this program print out? Well, that depends upon how you *evaluate* the integer expression 5 ∗A × 2 ∗ B. Here are four different ways you could evaluate it (remember, *A* = 10 and *B* = 5):

```
((5 * 10) + 2) * 5      = 260
(5 * 10) + (2 * 5)      =60
5 * (10 + 2) * 5        = 300
5 * (10 + (2 * 5))      = 100
```

As you can see, the *order of evaluation* can make a big difference. Which brings you back to your question:What will Pascal do with this expression? *Answer*: it will use *operator precedence*. Pascal will carry out all multiply, divide, and remainder operations *before* any addition or subtraction. A complete list of operator precedence is given in the first of two tables at the end of this chapter. So the correct (according to Pascal) evaluation is:

```
(5 * 10) + (2 * 5)    =  50 + 10  =  60
```

Having solved that problem, let's tackle the next one. How would you evaluate the expression *A* **div** *2 * B*? You have two choices (again, *A* = 10 and *B* = 5):

```
(10 div 2) * 5         = 25
10 div (2 * 5)         = 1
```

You would probably choose the first one, and you'd be right. When operators of equal precedence show up, you take them in the order they come: left to right.

Next question: what if you don't like the order of evaluation? What if you wanted *5 * A + 2 * B* to equal 260 instead of 60? *Answer*:do just what you did above, namely, use parentheses. Any part of an expression within parentheses is evaluated before it's combined with anything outside of the parentheses. This applies to nested parentheses, as well. Our examples above, well worth repeating, show how this works:

```
((5*10)+2)*5   =  (50+2)*5    =52*5   =  260
5*(10+2)*5     =  5*12*5      =60*5   =  300
(5*10)+(2*5)   =  50+10               =  60
5*(10+(2*5))   =  5*(10+10)   =5*20   =  100
```

So, assuming that you do want an answer of 260, you have to modify your program like this:

```
program Sample;
var
  A,B,C        :  Integer;
begin
  A := 10;
  B := 5;
  C := ((5 * A) + 2) * B;
  Write('C = ',C)
end. { of program Sample }
```

Actually, since multiplication has precedence over addition, you could drop the innermost set of parentheses and write:

```
    C := (5 * A + 2) * B;
```

What started out as a short discussion of integer expressions has gotten rather long. And you're not done yet. Let's modify your program a little more:

```
program Sample;
var
  A,B,C        :  Integer;
begin
  A := 100;
  B := 200;
  C := ((5 * A) + 2) * B;
  Write('C = ',C)
end. { of program Sample }
```

What value will you get when this program runs? Well, let's look at it:

((5*100)+2)*200 = (500+2)*200 = 502*200 = **100400**

The variable *C*, being an *integer*, can only hold values up to *MaxInt*, which equals 32767. You've caused a *value range error*. This simply means that you've tried to give a variable some value outside of its *allowable range*. What will happen? The variable *C* will end up with the lower 16 bits of the result, which in this case will come out as -30672. So, you need to **BE CAREFUL**. The worst kind of program bugs are those that allow your program to keep running with values that no longer have any relation to your original data. A few simple steps can check the result to see if it is within the realm of realism.

In addition to the operators described above, Pascal offers a few predefined functions that work on integers. They're listed below, with the data type of the result given in parentheses:

Abs(I)	returns absolute value of I (integer)
Odd(I)	returns true if I is odd (boolean)
Pred(I)	returns I-1 (integer)
Random(I)	returns random integer from 0 to I-1
Sqr(I)	returns I*I (integer)
Sqrt(I)	returns square root of I (real)
Succ(I)	returns I+1 (integer)

9.2.1 Integers as Unsigned Values

Sometimes you need to use integers as unsigned 16-bit values. The most common reason is to represent some address in memory (RAM). There is, however, a problem you run into: once you get above 32K, you have to use a negative number. The sequence goes like this:

```
0, 1, 2, ..., 32766, 32767, -32768, -32767, ..., -2, -1
```

Hardly convenient or intuitive. TURBO Pascal, however, gives you a way around this:hexadecimal (base 16) constants. A hex value can have up to four digits, where each digit can be *0* through *9* or *A* (=10) through *F* (=15). All you have to do is start the constant with a dollar sign (**$**). Here, for example, is the same sequence expressed as hex constants:

```
$0, $1, $2, ..., $7FFE, $7FFF, $8000, $8001, ..., $FFFE, $FFFF
```

Since memory addresses are often expressed in hexadecimal anyway, this makes for easy, convenient manipulation of address values.

TURBO Pascal provides other ways of treating integers as unsigned values. In addition to the operators and functions described above, TURBO provide six *operators*, two *functions*, and a *procedure* that let you do bit and byte manipulation on integer values. First, the operators:

not	logical negation
and	logical AND
or	logical OR
xor	logical exclusive OR
shl	shift left
shr	shift right

The first operator, **not**, is a unary operator; that means that it only affects the integer value it precedes. It differs from unary minus (-) in that it flips each bit within the value from 0 to 1 or 1 to 0, while a - does an arithmetic negation. An example using hex constants may make things clear:

```
  - $0001  = $FFFF (-1)
not $0001  = $FFFE (-2)
```

The unary minus changed **1** to **-1**, which has a hex value of $FFFF. The NOT operator flipped each bit in $0001, which means that it changed all bits from 0's to 1's except for the very last, which changed from a 1 to a 0. The result—$FFFE—happens to be equivalent to a signed value of **-2**.

The next three operators perform the standard logical functions defined in the second of two tables at the end of this chapter. They can be used to combine or mask values. For example, if you want to see if a particular bit is set (=1), you could do something like this:

```
var
  BitSet      :  Boolean;
  Val,Mask    :  Integer;
begin
  .
  .
  .
  BitSet := (Val and Mask) <> 0;
```

The Boolean variable *BitSet* will be set to *True* if any of the bits in *Val* match any of the bits in *Mask*. By the same token, the statement:

```
Val := Val or Mask;
```

will cause each bit in Val to be set to 1 if the corresponding bit in Mask is also 1.

The last two operators, **shl** and **shr**, allow you to shift all the bits within an integer to the left or right, respectively. The first operand is the value to be shifted; the second, how many *places* to shift. Suppose that we set *Val* := $0810, which has a bit pattern of (0000,1000,0001,0000). Here are some expressions with their resulting values:

Val **shl** 1	$1020	(0001,0000,0010,0000)
Val **shr** 1	$0408	(0000,0100,0000,1000)
Val **shl** 5	$0200	(0000,0010,0000,0000)
Val **shr** 4	$0081	(0000,0000,1000,0001)

As you can see, **shl** and **shr** shift in 0's and throw away whatever gets shifted out.

As mentioned, TURBO Pascal also provides two functions and a procedure for byte manipulation:

Lo(I)	Returns lower byte of I
Hi(I)	Returns upper byte of I
Swap(I)	Swaps upper and lower bytes of I

These do just what you think they would do. For example, assuming that *Val* = $0810, then

Lo(Val)	returns $0010
Hi(Val)	returns $0008
Swap(Val)	returns $1008

Note that both *Lo* and *Hi* set the upper byte to $00.

9.3 BYTE

The data type Byte is simply a subrange of *Integer*, consisting of the values **0** through **255**. *Byte* and *Integer* expressions and variables can be freely mixed. In fact, *Byte* variables act like *Integer* variables in all respects but this:any value outside of the range 0..255 which is assigned to a *Byte* variable is pared down to the 0..255 range. For example, if *Small* is declared to be of type *Byte*, then the following assignments result in the corresponding values for *Small*:

```
Small := -1;                   255
Small := -2;                   254
Small := 256;                    0
Small := Round(1040.0);         16
Small := Round(104000.0)   *error*
```

In short, assigning a value to a *Byte* variable is equivalent to assigning that value to an integer variable, then assigning the lower byte of that integer to the *Byte* variable. Confused? Well, if Large is of type *Integer*, then *Small* := –1 is equivalent to:

```
Large := -1;
Small := Lo(Large);     or     Small := Large and $00FF;
```

There is one other difference:*Byte* variables take up only one byte of memory, while *Integer* variables take up two. Therefore, if you want to be economical with your computer's memory and only need to store small values, you should use *Byte* variables.

9.4 REAL

Suppose that you want *1¼* to equal *2.75* instead of *2*. Suppose that you want*199/200* to equal *0.995* instead of *0*. Suppose that you want to find the answer to *502*200*. What do you do? You use *real* numbers, instead of *integers*.

Reals differ from integers in two important respects. First, they have a decimal point with additional digits following it. In other words, they have values such as *2.75*, *0.995*, *-421.0*, and so on. Second, they can (but do not have to) have *exponents*. The exponent represents some power of 10 by which the rest of the real number is multiplied. For example, the value 100400 can be written as:

100400.0 or 100400.0E+0	(100400.0 * 1)
10040.0E+1	(10040.00 * 10)
1004.0E+2	(1004.000 * 100)
100.4E+3	(100.4000 * 1000)
10.04E+4	(10.04000 * 10000)
1.004E+5	(1.004000 * 100000)
0.1004E+6	(0.100400 * 1000000)

Note the format:*mantissa* E *exponent*. The *mantissa* is simply a real number, say, *100.4*. The exponent is some integer value with an explicit sign (+ or -), say, +*3*. And the resulting value is:

```
<mantissa> * 10 <exponent>
```

such as *100.4 * 1000*, or *100400.0*. Of course, you don't always have to use the exponent format—just if you're working with very large or very small numbers. (A negative exponent gives you a very small number, moving the decimal point to the left according to the value of the exponent.)

If you choose to not use exponents, you can just write real numbers as you normally would:

3.1415926
-3546.3
0.0034
56793834.21

The same rules of *operator precedence* mentioned above apply to real numbers. In addition, TURBO Pascal offers some predefined functions for use with reals. Note that the trigonometric functions (arctan, cos, sin) assume that the angle involved is in *radians* (360 degrees = 2*pi radians):

Abs(x)	returns absolute value of x (real)
ArcTan(x)	returns arctangent of x (real)
Cos(x)	returns cosine of x (real)
Exp(x)	returns e to the x power (real)
Frac(x)	returns fractional portion of x (real)
Int(x)	returns integer portion of x (real)
Ln(x)	returns natural log of x (real)

Random	returns random number from 0.0 to 0.99... (real)
Round(x)	returns integer nearest to x (integer)
Sin(x)	returns sine of x (real)
Sqr(x)	returns x*x (real)
Sqrt(x)	returns square root of x (real)
Trunc(x)	returns integer portion of x (integer)

You need to be careful mixing reals with integer values. TURBO Pascal will automatically convert integer numbers to reals, but not vice versa. Let's suppose that you wanted to find the area of a circle of some radius. You might write this program:

```
program FindRadius;
{
}    note: Pi is a predefined Real constant = 3.1415926536
var
  Radius,Area        :  Integer;
begin
  Write('Enter radius'    '); ReadLn(Radius);
  Area := Pi * Radius*Radius;
  WriteLn('The area is ',Area)
end; { of program FindRadius }
```

TURBO Pascal will not compile this program; instead, you'll get a "Type mismatch" error. Why? Well, the expression *Pi * Radius*Radius* is, by default, a real expression, and you can't assign a *real* value to an *integer* variable. However, you can use *Trunc* or *Round* to make the conversion. *Trunc(X)* returns the integer portion of X; in other words, it drops the fractional portion of X. Here are some sample results:

Trunc(3.1415926)	=	3
Trunc(-32.3)	=	-32
Trunc(421.7)	=	421
Trunc(0.5)	=	0
Trunc(543832.32)	=	**error** (too large for integer)

Round(X), on the other hand, returns the integer nearest to X. A special case occurs when X is exactly between two integers, i.e., the fractional portion is 0.5. TURBO Pascal will round away from zero, which means *Round(22.5)* = 23 and *Round(-22.5)* = -23. Here are some other results:

```
Round(3.1415926)    =          3
Round(-32.3)        =        -32
Round(421.7)        =        422
Round(0.5)          =          1
Round(542832.32)    =   **error**(too large for integer)
```

Note that if you use *Round* or *Trunc* on a real value that's too large, you will get a run-time error.

You can now rewrite your program like this:

```
program FindRadius;
{
    note: Pi is a predefined Real constant = 3.1415926536
}
var
  Radius,Area        :  Integer;
begin
  Write('Enter radius:    '); ReadLn(Radius);
  Area := Round(Pi*Radius*Radius);
  WriteLn('The area is ',Area)
end; { of program FindRadius }
```

Two issues come up with reals or, rather, two aspects of the same issue. The issue is this:how many bytes are used to represent a real? With TURBO Pascal, the answer is:6 bytes. The first aspect deals with the allowable range for the exponent. Unless you're dealing with really large numbers, you won't have worry too much about this—TURBO has a range of -38 to +38, and 10 raised to the power of 38 is a *very* large number. This, of course, doesn't mean that the issue will never come up. It's just very unlikely.

More critical is the issue of *precision* in the mantissa. In plain English, how long can the real number be? With TURBO Pascal, about 11 digits. This provides enough precision to be able to do financial and other calculations without too much worry. Even so, you may end up with some slightly (or not-so-slightly) wrong values should you push too close to the limit or should you have to do a long series of calculations. This is due to *cumulative round-off error*, or **CROE**. CROE is a fancy way of saying that bad values tend to propagate themselves. Too many bad values can work together to produce results that have no relation to your initial values.

What causes these mysterious errors? Here are four major factors behind CROE. First, decimal numbers are represented in binary form. Did you know that the number 0.1 is impossible to express exactly in binary form? It's an infinite fraction, much like 1/3 in decimal form (0.333333...). Several of these errors can cause values to shift slightly.

Second, values with different exponents are normalized before addition. (This means that their exponents are made equal and their mantissas adjusted accordingly. If you add 1.0E+04 (10,000) and 1.5E+00 (1.5), then the latter is normalized to 0.00015E+04 before the two are added together. With limited precision, you might lose some digits off during normalization.

Third, when nearly identical values are subtracted, the least significant (and least accurate) digits become the most significant digits. If you subtract 1.4356876523E+05 from 1.4356876527E+05, you should end up with 4.0E-05. But if you have 11 significant digits, then the last digit in each number (3 or 7) is the one most likely to be wrong due to the other errors mentioned here. *The result*: your answer may be totally wrong. Furthermore, that answer is then used to "fill" 11 digits. Combine that with the binary/decimal problem, and you get an actual answer of 3.9815902710E-05...not quite what you'd expect.

Finally, when two numbers of some precision are multiplied together, the result has twice as many digits, half of which are then thrown away. For example, if you multiply the two numbers above, you get 2.06111990355E+10. The "5" at the end represents the lower 12 digits of the answer (11 digits plus itself). Again, due to round off and internal representation, errors can creep in, though this is usually less serious than the pitfalls mentioned above.

An error always starts out at the lower end of a number and moves up towards the front. The more calculations you have, the farther it can move. The fewer digits of precision you have, the faster it can move. *The moral*: if you're using real numbers, be aware of your limits. If you're going to do a lot of number crunching, you might want consider using the TURBO-87 option (assuming that you're running on an 8086-based computer with an 8087 math co-processor). This will not only extend your precision to 16 digits (8 bytes), but it also gives you a much wider exponent range (-307 to +308) and, of course, tremendously speeds up all your real number calculations.

9.5 CHAR

The data type *Char* is simply the set of ASCII characters. More accurately, a variable of type Char can have 256 values, including the 128 characters in the standard ASCII set. You may remember from Chapter 7 the discussion on printing and non-printing characters. Printing characters can be represented as the character itself within two single quote marks:

```
'a'   '$'   ' '   'J'
```

Remember the *control characters*? Those are the characters with ASCII values 0 through 31, and they are usually expressed in terms of the printing characters ranging from 64 to 95. For example, the character with the ASCII value of 7 is commonly called "control-G"; 'G' has an ASCII value of 71 (=7+64). But since these characters have no printable equivalent, there's no way to represent them within quotes. However, TURBO Pascal lets you represent them with the notation:

```
^char,
```

where *char* is the corresponding printable character. You can use:

^G for "control-G"
^[for ESC (ASCII code 27)

and so forth.

That takes care of codes 0 through 31, but what about characters with ASCII codes 127 (DEL) through 255? TURBO again comes to the rescue, with the notation *#val*, where *val* is a byte constant (0...255). For example, you could represent DEL as **#127** or **#$7F** (yes, you *can* use hex constants). Here, then, are some different ways of representing the same characters:

NUL	#0	#$00	^@
ctrl-G (BEL)	#7	#$07	^G
ESC	#27	#$1B	^[
blank (space)	#32	#$20	' '
the digit 0	#48	#$30	'0'
the letter A	#65	#$41	'A'
DEL	#127	#$7F	
ASCII 237	#237	#$ED	

Suppose, now, you wanted to put a control-G (which causes a beep on printing) within a string. If you wrote:

```
ErrorMsg := 'ERROR: ^G You entered the wrong value';
```

then it would print:

```
ERROR: ^G You entered the wrong value
```

and there would be *no beep*. Not to worry:TURBO Pascal makes insertion of special characters simple. All you have to do is to break the string up into two parts and put the control-G between them:

```
ErrorMsg := 'ERROR: '^G' You entered the wrong value';
```

This automatically inserts the control-G character into the string contained in *Error*. You can do the same thing with the *#val* characters. In fact, you can build a string consisting of only special characters:

```
FiveBeeps := ^G^G^G^G^G#13#10;
```

FiveBeeps now has a length of 7 and contains 5 control-G's (BEL), a carriage return (CR), and a line feed (LF). Writing this to the screen would cause 5 beeps and would then move the cursor to the start of the next line.

TURBO Pascal has a few functions that deal with characters:

Chr(I)	returns the character corresponding to I (char)
Ord(C)	returns the integer corresponding to C (integer)
Pred(C)	returns the character preceding C (char)
Succ(C)	returns the character following C (char)
UpCase(C)	returns the upper-case value of C (char a...z)

You'll find additional discussions of characters in Chapter 14, which talks about strings.

9.6 BOOLEAN

George Boole, a 19th-century mathematician, developed a whole set of rules for numbers that could have only two values:*0* and *1*. It was called *Boolean algebra*. When computers appeared a century later, most were based on components that could represent one of two values: *0* and *1*. All of Boole's work, and the work of those who followed him, found immediate application.

Pascal allows you to declare variables of type Boolean. However, instead of using *0* and *1*, Pascal uses *False* and *True* to represent boolean values. Boolean variables and expressions aren't terribly useful all by themselves; however, they are critical for *control structures*, which is the topic of Chapter 10. Boolean expressions will be covered in more depth there.

Pascal provides the logical operators **not, and**, **or**, and **xor** for use with boolean values. The first of the two tables at the end of this chapter shows the precedence for these operators, and the second table shows what results you get (substituting False for 0 and True for 1).

Boolean values can also be returned by using the comparison operators to compare other values:

```
=    <>
<
>
<=
>=
IN
```

For example

```
3 > 5
```

will return *False*, while

```
7 <= 10
```

will return **True**. Again, more about this will be discussed in following chapters.

OPERATOR PRECEDENCE

–	Unary minus (arithmetic negation)
not	Logical negation
*, /, div, mod, and, shl, shr	Multiplying operators
*, –, or, xor	Adding operators
=, < >, <, >, <= , >=, in	Relational operators

LOGICAL OPERATOR TRUTH TABLES

0 = False 1 = True

exp1	exp2	(exp1 **and** exp2)	(exp1 **or** exp2)	(exp1 **xor** exp2)
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

10. CONTROL STRUCTURES

One of the great buzz words (buzz phrases?) of computer science is *flow of control*. This term refers to the order in which the statements in your program are executed. Up until now, you've only looked at *sequential* flow of control. Simply put, sequential flow of control means that each statement is executed exactly once, starting with the first statement and ending with the last. Nice, but it won't get you far.

To provide a little variety, Pascal has a number of *control structures*, special constructs that allow you to have *nonsequential* execution of statements. There are three types of control structures in Pascal: *conditional*, *iterative*, and *case*.

The goal of this section is to explain what these constructs are and how you use them. First, though, you need to look at a few basics: *statements* and *boolean expressions*.

10.1 STATEMENTS

The primary unit of execution in a Pascal program is the *statement*. At the simplest level, there are two kinds of statements: *procedure calls* and *assignments*. You've looked at built-in procedures, such as *WriteLn*; a *procedure call* is the act of using such a procedure:

```
    WriteLn('The sum is ',C);
```

Assignment statements assign some value to a variable. They take the form:

```
    <variable> := <expression>;
```

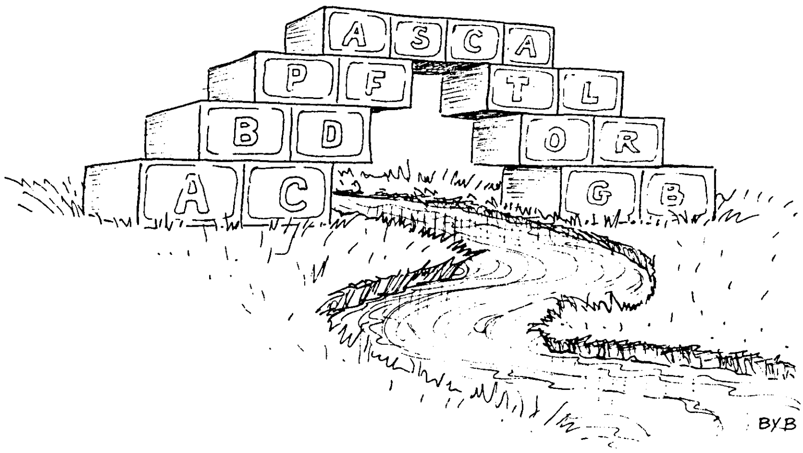
where *expression* is, well, an expression that resolves to a value appropriate to the data type of *variable*. For example, if the variables Score and Maximum are of type Integer, then the following statements are valid:

```
    Score := 10;
    Maximum := 32*Score div 17;
    Score := Score + 10;
    Maximum := Succ(Maximum);
```

All but one of the control structures we'll be looking at will execute only one statement. This doesn't sound very impressive or convenient, and, in fact, it would make for a very serious limitation *if* you could only use procedure calls and assignment statements. Such is not the case. Pascal considers the following structure to be a single statement:

```
begin
  <statement 1>;
  <statement 2>;
  .
  .
  .
  <statement n>
end;
```

This structure is known as a block or *compound* statement. It can be used anywhere that a simple statement can; you can even use one block statement within another. Compound statements are most often used in conjunction with control structures, as you'll soon see.



—Blocks (compound structures)—

10.2 BOOLEAN EXPRESSIONS

A moment ago, you were looking at expressions—integer expressions, in particular. Those shown were integer expressions because they returned integer values. A boolean expression, then, returns a boolean value—in other words, either *True* or *False*. For example, suppose that the integer variables *Score* and *Maximum* have each been set to some value, say, *Score* = 10 and *Maximum* = 0. You can now create the following boolean expressions, with the corresponding results:

Score >Maximum	True	(is greater than)
Score =Maximum	False	(is equal to)
Score <Maximum	False	(is less than)
Score >= Maximum	True	(is greater than or equal to)
Score <= Maximum	False	(is less than or equal to)
Score <> Maximum	True	(is not equal to)

In a Pascal program, such expressions would actually return *True* or *False*, the two possible values of the predefined data type *Boolean*. We could define a boolean variable, *NewMax*, and then have the following assignment statement:

```
NewMax := (Score > Maximum);
```

Depending upon the current values of *Score* and *Maximum*, this would set *NewMax* to either *True* or *False*. For example, given the values defined above, NewMax would receive a value of *True*.

You can create more complex expressions by using the Boolean operators **and**, XOR, **or**, and **not**. **not** flips the value, changing *True* to *False* and vice versa. **and**, **xor** and **or** combine two expressions to create a new one. The table at the end of this chapter is a truth table showing how these operators work.

Using these operators, you can create such expressions as

```
(Score > Maximum) or (Score > 30000)
(Score > 10000) and (Score <= 20000)
not (NewMax or (Maximum = 0))
```

If we assume that *Score, Maximum*, and *NewMax* have the values given above (10, 0, and *True*), then the first expression resolves to *True*, while the other two resolve to *False*. Having discussed all this, you're now ready to go on to *control structures*.(Decisions Illustration goes here)

—Decisions, Decisions, Decisions—

10.3 CONDITIONAL EXECUTION

In writing programs, you usually have groups of statements that you want to be executed only if some condition has been met. Not surprisingly, this is known as *conditional execution*. In Pascal, conditional execution takes the form:

```
if <boolean expression>
  then <statement>;
```

The **if...then** structure works quite simply. If *boolean expression* returns a value of *True*, then *statement* is executed; otherwise, it is ignored. For example,

```
if Score > Maximum
  then Maximum := Score;
```

will set *Maximum* to *Score* **if and only if** *Score* is greater than *Maximum.* You could also do the following:

```
NewMax := (Score > Maximum);
if NewMax
  then Maximum := Score;
```

If you really want to get fancy, you can do this:

```
if Score > Maximum then begin
  Maximum := Score;
  WriteLn('Congratulations!');
  WriteLn('Your new high score is ',Maximum)
end;
```

Tied closely to the idea of conditional execution is that of *alternation*. Alternation means that one set of statements is executed if a condition is *True*, and another set if that condition is *False*. You could do it this way:

```
if <condition>
  then <statement1>;
if not <condition>
  then <statement2>;
```

Pascal, however, provides a simpler way:

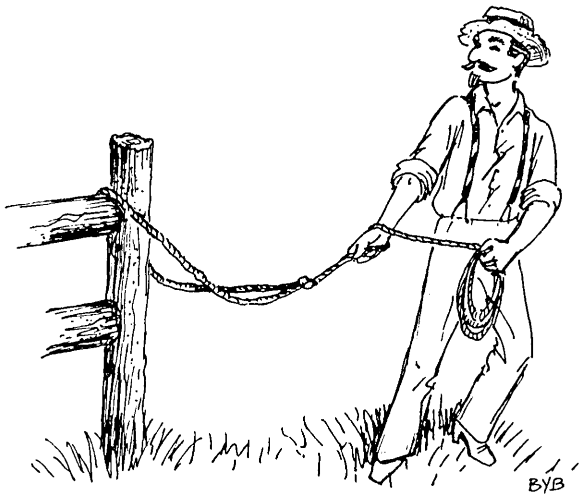
```
if <condition>
  then <statement1>
  else <statement2>;
```

By using the **if...then...else** construct, you can amplify your example as follows:

```
if Score > Maximum then begin
  Maximum := Score;
  WriteLn('Congratulations!');
  WriteLn('You have set a new high score of ',Maximum)
end
else begin
  WriteLn('Your score was ',Score);
  WriteLn('Nice try, but you can do better')
end;
```

10.4 ITERATION

Conditional execution adds a lot to your programming capabilities, but you still have a major handicap: You can execute each statement only once. You need the ability to repeat a section of code some number of times or until some condition is met. This looping process is known as *iteration*. Niklaus Wirth must have really liked loops, because Pascal has not one, not two, but three forms of iteration: **for...do, while...do**, and **repeat...until**. Let's look at each.



—Looping—

10.4.1 The FOR...DO Loop

There are situations where you want to have a statement executed some number of times. Often, you want or need to know how many times you've gone through the loop so far. Or, you might want a variable to step through a range of values, executing a statement (or group of statements) once for each value.

The **for...do** loop allows you to do this. It uses the format:

```
for <var> := <exp1> to <exp2> do
  <statement>;
```

When the loop is encountered, *var* is assigned the value of *exp1*. If *var* is less than or equal to *exp2*, then *statement* is executed. *var* is then increased by one, and the test is repeated. This goes on until *var* is greater than *exp2*. If *exp1* is greater than *exp2*, then *statement* is never executed. For example, you could write:

```
for Indx := 1 to 10 do
  WriteLn('n = ',Indx,' n*n = ',Indx*Indx);
```

This will display the integers from 1 to 10, along with their squares. If you want to decrement (decrease) *var* instead of incrementing it, you can use the format:

```
for <var> := <exp1> downto <exp2> do
  <statement>;
```

Now, *var* decreases each time until it is less than *exp2*, and *statement* is never executed if *exp1* is less than *exp2*.

The **for...do** loop in Pascal sacrifices convenience for a gain in power. You cannot specify a *step* value by which to change *var*, as you can in most other languages. To do that, you have to create your own loop using one of the other loop constructs (described below). Also, *var* cannot be of type Real; again, you have to use another type of loop.

So much for the inconvenience; where's the power? Well, *var* doesn't have to be of type Integer; it can be of any scalar data type: Char, Byte, Boolean, or any *declared scalar type* (DST) you care to define. Given appropriate definitions, the following loops are all valid:

```
for Indx := 30 downto 20 do
  <statement>;

for Ch := 'A' to 'Z' do
  <statement>;

for Flag := True downto False do
  <statement>;

for Day := Mon to Fri do
  <statement>;
```

Now you can see why there is no *step* capability. Mixing numeric and non-numeric values could be very confusing, especially for those programmers who write compilers. Besides, there is something clumsy about a statement like

```
for Month := January to December step 2 do
  <statement>;
```

while the statement

```
for Month := January to December step February do
  <statement>;
```

doesn't make any sense at all. So there is no *step* capability in **for** loops.

As with **if...then...else**, you can use a compound statement in place of *statement*, to wit:

```
for Indx := 20 to 30 do begin
  <statement 1>;
  .
  .
  .
  <statement n>
end;
```

10.4.2 The WHILE...DO Loop

Since the **for...do** loop can't easily meet all your iterative needs, Pascal give you two others, both controlled by boolean expressions. The first is the **while...do** loop:

```
while <boolean expression> do
  <statement>;
```

This loop will cause *statement* to be executed as long as *boolean expression* resolves to a value of *True*. It will never be executed at all if *boolean expression* is initially *False*.

If you wish, you can replace **for...do** loops as follows:

```
<var> := <exp1>;
while <var> <= <exp2> do begin
  <statement 1>;
  .
  .
  .
  <var> := Succ(<var>)
end;
```

If you want to use a *step* value, then you need to change the last statement to reflect that. For example, to have a step value of **3**, you could change the last statement in the loop to read

```
    <var> := <var> + 3
```

Of course, you can do other things than imitate **for** loops. For example, here's a binary search program. It searches a sorted list for a given value until it either finds it or determines that it is not in the list. The advantage of a binary search is that it only takes a maximum of (n log 2) tries to search a list of **n** objects. This means that a list of 1024 elements will take at most 10 tries through the loop.

```
procedure BinarySearch(DKey : ListVal;
var Index   :   Integer; var Found : Boolean);
{
    This procedure does a binary search for DKey in List,
    where List = ARRAY[IMin..IMax] OF ListVal
    List is sorted such that List[I] <= List[I+1]
    It searches for Index such that DKey = List[Index]
    If found, then Found = True else Found = False
}
```

```
var
  Upper,Lower        : Integer;
begin
  Upper := IMax;Lower := IMin; Found := False;
  while not Found and (Lower <= Upper) do begin
    Index := (Upper + Lower) DIV 2;
    if DKey < List[Index]
      then Upper := Pred(Index)
    else if DKey > List[Index]
      then Lower := Succ(Index)
    else Found := True
   end
end; { of proc BinarySearch }
```

10.4.3 The REPEAT...UNTIL Loop

The third loop is **repeat...until**. It takes the format:

```
repeat
  <statement 1>;
  <statement 2>;
  .
  .
  .
  <statement n>
until <boolean condition>;
```

This differs from **while...do** in three important ways. First, the statements within the loop are *always* executed at least once. In other words, all the statements in the loop are executed before *boolean condition* is resolved. Second, the statements continue to execute as long as *boolean condition* is false; the **while** loop executes while *boolean expression* is true. Third, the **repeat** loop can directly execute more than one statement and, therefore, doesn't need a compound statement (**begin...end**) to handle multiple statements.

One common use of **repeat** loops is to condition input, that is, to prompt the user for some value and to continue to prompt until the value entered is one of those that you decide to allow. For example, the following routine prompts the user to enter an integer within the range [low...high]. It repeats the prompt if (1) there is an I/O error (due to bad numeric format), or (2) the value is not within the proper range:

```
program TestGetInteger;
type
   Prompt        = string[80];
   .
   .
   .
procedure GetInteger(var Val : Integer; Msg : Prompt;
                          Low,High : Integer);
begin
   {$I-,R-} { turn off I/O, range checking }
   repeat
     Write(Msg);
     ReadLn(Val)
   until (IOresult=0) and (Low<=Val) and (Val<=High)
   {$I+,R+} { turn everything back on }
end; { of proc GetInteger }
```

This provides "bulletproof" integer input. No matter what the user tries to do, this routine will continue to ask for a proper value until the one entered has the proper format and is within the proper range.

10.5 THE CASE STATEMENT

When writing programs, you sometimes want to perform one of a number of actions depending upon the current value of a variable. For example, you might want to print out a menu, accept the user's choice, and then act accordingly. One way you can handle this is to use repeated **if...then...else** statements:

```
 repeat
   Write('Enter direction:   U)p,D)own,L)eft,R)ight:');
   ReadLn(Ch)         { Ch is of type Char }
until (Ch IN ['U','u','D','d','L','l','R','r']);
if (Ch = 'U') or (Ch = 'u')
   then Y := Y - 1
else if (Ch = 'D') or (Ch = 'd')
   then Y := Y + 1
else if (Ch = 'L') or (Ch = 'l')
   then X := X - 1
else if (Ch = 'R') or (Ch = 'r')
   then X := X + 1;

GotoXY(X,Y)
```

However, the **if...then...else** chain can get a little tiresome (and difficult to follow) if there are a lot of different conditions to check. So Pascal includes an additional flow-of-control structure called the *case* statement.

The **case** statement requires a *scalar variable* which has been set to some value. Scalar variables include those of type Integer, Byte, Char, Boolean, and any declared scalar type (DST). (Variables of type Real are not considered scalar.)The **case** statement has this format:

```
case <variable> of
   <constant list 1> : <statement 1>;
   <constant list 2> : <statement 2>;
   .
   .
   .
   <constant list n> : <statement n>
end;
```

where a *constant list* is a list, separated by commas, of constants of the same data type as the variable. When the **case** statement is encountered, the program compares the current value of the variable against each of the constant lists. When a match is found, the corresponding statement is executed and the rest of the **case** statement is skipped. You could now rewrite your first example as follows:

```
repeat
   Write('Enter direction: U)p,D)own,L)eft,R)ight:');
   ReadLn(Ch)          { Ch is of type Char }
until (Ch IN ['U','u','D','d','L','l','R','r']);

case Ch of
   'U','u' :     Y := Y - 1;
   'D','d' :     Y := Y + 1;
   'L','l' :     X := X - 1;
   'R','r' :     X := X + 1
end;
```

The above example ensures that *Ch* would only have one of the specified values in the **case** statement. This is done by checking *Ch* against a *set* of allowable values (you'll learn about sets in a later chapter). What if you removed that restriction? What if *Ch* had a value other than those listed in the **case** statement? Under the original Pascal definition (by Wirth and Jensen), the result was undefined; that is, the outcome was uncertain.

TURBO Pascal, however, takes the commonsense approach that, in such a case, *all* of the statements are skipped and execution resumes right after the end of the entire **case** statement. In fact, TURBO Pascal goes one step farther and allow an **else** section at the end of the list of choices. This section is executed only if the variable does not match any of the constant lists. With such an extension, you might rewrite your example as follows:

```
Write('Enter direction: U)p,D)own,L)eft,R)ight:    ');
ReadLn(Ch);
case Ch of
   'U','u' :     Y := Y + 1;
   'D','d' :     Y := Y -1;
   'L','l' :     X := X -1;
   'R','r' :     X := X + 1;
else
   WriteLn('Illegal entry');
   WriteLn('Please use U,D,L or R')
end;
```

10.6 SAMPLE PROGRAM

Now that you've looked at all of the different flow-of-control structures, let's put them all together in a program. The listing at the end of this chapter shows a simple (very simple!) Pascal program that uses all of the constructs you've learned about.

The first construct you encounter is a **while** loop. It tests the boolean variable *Done* and will loop as long as *Done* has a value of *False*.

The next construct is a **repeat** loop. The statements in this loop prompt the user for a character. The program will not leave the loop until the user enters one of the valid command characters.

Within the **repeat** loop is your third construct, an **if...then...else** statement that tells the user whether or not one of the command characters has been entered.

Following the **repeat** loop is your fourth construct, a **case** statement. It performs one of six actions depending upon the character value the user has entered. Four of the commands change the values of X or Y. One command sets *Done* equal to *True* and so allows the program to exit the **while** loop.

The remaining command ('F' or 'f') executes a block statement which contains your last construct, a **for** loop. This loop executes the statement *Write(Dot,Bell)* five times (the value of the constant *FireMax*).

10.7 CONCLUSION

These control structures greatly add to your programming capabilities. They help you to escape from the "every statement exactly once" approach of your earlier programs while maintaining a clearly-defined path of execution through the code. By combining the various structures together, you can create exactly the path you need.

Even with these aids, you still are limited to writing one big program and having to duplicate code if you need it in more than one place. In the next chapter, you will learn about the solution: *procedures* and *functions*. You've already seen some standard Pascal procedures and functions, such as *WriteLn* and *ReadLn*; in the next chapter, you'll learn how to write your own.

Before we go on to the next chapter, here are the table and the sample program I promised you a while ago:

BOOLEAN TRUTH TABLES

Exp1	Exp2	(Exp1 **and** Exp2)	(Exp1 **xor** Exp2)	(Exp1 **or** Exp2)
False	False	False	False	False
True	False	False	True	True
False	True	False	True	True
True	True	True	False	True

CONTROL STRUCTURES PROGRAM EXAMPLE

```
program Example;
{
    Example                    Sample Pascal program
                               demonstrates control structures
                               note the use of typed constants
}
const
   Bell                        = ^G;
   FireMax                     =  5;
   Dot                         = '.';
   Done                        : Boolean = False;
   X                           : Integer = 0;
   Y                           : Integer = 0;
   CmdSet                      : set of Char = ['L','R','U','D','F','Q'];
var
   Ch                          : Char;
   Count                       : Integer;
```

```
begin
  while not Done do begin
    WriteLn('X,Y = (',X,',',Y,')');
    repeat
      Write('CMD:   L)eft, R)ight, U)p, D)own, F)ire, Q)uit:   ');
      Read(Kbd,Ch); WriteLn;
      if Ch >= 'a'            { convert to upper case }
        then Ch := Chr(Ord(Ch)-32);
      if not (Ch IN CmdSet)
        then WriteLn('Illegal entry—try again')
        else WriteLn('Command accepted')
    until Ch in CmdSet;
    case Ch of
      'U'  :  Y := Y + 1;
      'D'  :  Y := Y -1;
      'R'  :  X := X + 1;
      'L'  :  X := X -1;
      'F'  :  begin
                Write('Firing');
                for Count := 1 to FireMax do
                  Write(Dot,Bell);
                WriteLn
              end
      'Q'  :  Done := True
    end { of case }
  end { of while }
end. { of program Example }
```

11. PROCEDURES AND FUNCTIONS

In previous chapters, you learned how to write simple Pascal programs. *Period.* True, you've learned all about boolean expressions, flow-of-control structures, and other such esoterica. But the fact remains that all you can do is write one big program that starts at **begin** and finishes at **end**. If you have a set of commands that you want to use in five different places, then you would have to write that code in all five places. (Of course, if you merely want to execute the same code five times, you could put in a **for** loop, but we covered that in the previous chapter.)

That limitation ends now. This chapter will discuss the use of *procedures* and *functions*, which together are known as *subprograms*. You'll first hear about the general concept of a subprogram. You'll then see how (and where) to place subprograms in your main program. Then you'll touch on the idea of *scope*, which governs where a subprogram may be called from. You'll end with a discussion on *parameters* for subprograms and I'll give you a taste of some of the different things you can do with them.

11.1 SUBPROGRAMS

As you've learned in previous chapters, a Pascal program has the following structure:

```
program Whatever;
  <declaration section>
begin
  <program statements>
end. { of program whatever }
```

As mentioned above, you can get by with just this for small programs, but you'll find this structure very limiting with big programs, especially if you have sections of code that are repeated throughout the program. The solution? Pascal allows you to create *subprograms*. A subprogram is indeed like a miniature program contained within a larger one. Here, for example, is the structure for defining a **procedure** (one of two types of subprograms):

```
procedure Whichever;
  <declaration section>
begin
  <procedure statements>
end; { of procedure Whichever }
```

Look familiar? The only visible differences between this format and the format for a program are (1) the word **procedure** is used, and (2) the **end** is followed by a *semicolon* instead of a *period*. (A few more differences will crop up later.) As with a program, execution starts right after the **begin** statement and finishes at the **end** statement.

The next question is: where do you put subprograms in your program? Answer: after the declaration section and before the opening **begin** of your program. Indeed, you can think of your subprograms as being the last part of your declaration section (after all of your **label**, **const**, **type**, and **var** statements). Some of you may then wonder if you can declare a subprogram within a subprogram. Yes, you can, just like you would in a regular program. And you could declare subprograms within *that* subprogram, and so on *ad infinitum* (well, not quite; the TURBO Pascal compiler *does* have its limits).

Now that you've got the subprogram in place, the next question is, how do you use it? Answer: simply by stating its name. Let's put our procedure within our program and call it a few times:

```
program whatever;
const
  Maximum             = 10;
type
  CRange              = 0..Maximum;
var
  Indx                : CRange;
```

```
procedure Whichever;
const
  Star                = '*';
type
  Days                = (Mon,Tues,Wed,Thur,Fri,Sat,Sun);
var
  Day                 : Days;
begin { main body of proc Whichever }
  for Day := Mon to Fri do
    Write(Star)
end; { of proc Whichever }

begin { main body of program Whatever }
  for Indx := 0 to Maximum do begin
    Whichever;
    WriteLn
  end
end. { of program Whatever }
```

In this example, **const**, **type**, and **var** statements have been deliberately included in both the main program and the subprogram so that (1) you won't be confused about where subprograms go, and (2) you can see how similar the two structures are. Look over the example carefully and see if you can figure out just what it will do. In case you're not sure, remember that Write and WriteLn are built-in procedures (i.e., they're always there—you don't have to define them). *Write(Star)* will print an '*' on the screen, while *WriteLn* will start a new line.

Finished? If you guessed that this program would print out eleven lines, each consisting of five asterisks ('*****'), you're right. The procedure *Whichever* prints out five asterisks each time it is called. The program *Whatever* calls *Whichever* eleven times, starting a new line after each call.

11.2 SCOPE

Before you can go any farther into the intricacies of subprograms, you must first deal with the concept of *scope*. In a BASIC program, all identifiers (variables, functions, line numbers) are *global* in scope. In other words, any identifier can be referenced from anywhere in the program. Indeed, any identifier can be declared anywhere in the program, and variables are declared merely by referring to them.

In Pascal, things are quite different. The Great Underlying Rule of Pascal is that a given identifier must be declared before it is used. *Again: any constant, data type, variable, or subprogram you care to use has to be explicitly defined first.* **One last time:each and every identifier appearing in any program (or subprogram) statements needs to have been previously described in a declaration section.**

Having made that pronouncement, you must now understand what is meant by "previously declared." In the simplest sense, it means that if you start at the **program** statement and move down, you must run across the declaration of each identifier before you find any other use of it. You can easily check that in a program containing no subprograms, since you need only to verify that each identifier used in the program statements is found in the declaration section (and that, of course, you are following the same rule within the declaration section itself). But consider this example:

```
program Scope1;
var
  A                    : Integer;


procedure SetB;
var
  B                    : Integer;
begin { proc SetB}
  B := 2*A
end; { of proc SetB }

begin { main body of prog scope1 }
  A := 2;
  SetB;
  WriteLn(A+B)
end. { of program Scope1 }
```

Here, your simple rule has been met—both **A** and **B** have been defined as being variables of type Integer before being used—and yet this program will not compile. Why? Because *B*, having been declared in the procedure *SetB*, is only recognized within that procedure. In other words, the variable *B* is *local* to the procedure *SetB*. By contrast, the variable *A* is recognized both within *SetB* and in the main program itself.

So you now must modify your simple rule to this:*each identifier must have been previously declared in an enclosing program or subprogram before being used.* By *enclosing*, I mean that the definition of the identifier must come after its declaration and before the **begin** statement of the program or subprogram in which is was defined. In the example above, you can see that the statement *WriteLn(A+B)* comes after the **begin** statement of the procedure *SetB*.

Here's another case to consider:

```
program Scope2;
var
  A                    : Integer;

procedure SetA;
var
  A                    : Integer;
begin
  A := 4
end { of proc SetA }

begin { main body of program Scope2 }
  A := 3;
  SetA;
  WriteLn(A)
end. { of prog Scope2 }
```

Two questions: (1) will this program compile, and (2) if it does, what value will it print out? *Two answers*: (1) **yes**, and (2) **3**. This brings up a corollary to our rule:the most recent declaration of a given identifier will always be used. The procedure *SetA* assigned its own local variable A to 4 and left the global variable A untouched.

An editorial comment. Those of you used to BASIC or FORTRAN may chaff at this discipline that Pascal requires of you. After all, it's so nice to make up variables as you go along. However, an excellent reason for these restrictions exists. There are two great sources of program bugs in BASIC and FORTRAN: misspelled variables and side effects. It is not uncommon to type "INDEX" in one statement and mis-type "INDX" in another and have the program (BASIC or FORTRAN) go merrily on its way. It is also not unusual to (unknowingly) use the same variable in several different parts of the program, often with bizarre results (especially if those sections call one another). Pascal takes great strides towards eliminating those bugs. You can still create them, it is true, but you will really have to work at it.

11.3 PARAMETERS

Perhaps the most common use of subprograms is to perform the same set of operations using different sets of values. For example, you might want a procedure which would swap two integer values. The question is, which two values should it swap? One solution is this:

```
program TestSwap;
var
  S1,S2,Alpha,Bravo,Charlie,Delta,Eagle,Foxtrot
                                   : Integer;

procedure Swap;
var
  Temp                             : Integer;
begin
  Temp := S1;
  S1 := S2;
  S2 := Temp
end; { of proc Swap }

begin { main body of program TestSwap }
  Alpha := 1; Bravo := 2; Charlie := 3;
  Delta := 4; Eagle := 5; Foxtrot := 6;
```

```
   S1 := Alpha; S2 := Eagle;
   Swap;
   Alpha := S1; Eagle := S2;

   S1 := Bravo; S2 := Delta;
   Swap;
   Bravo := S1; Delta := S2;

   S1 := Charlie; S2 := Foxtrot;
   Swap;
   Charlie := S1; Foxtrot := S2

end. { of prog SwapTest }
```

Now, a little thought will show just what a lousy approach this is. You are executing more statements before and after each call to Swap then you would if you just did each swap directly. Luckily, you don't really have to do this. Instead, you can define Swap as having a list of *parameters*. For example:

```
program TestSwap;
var
   Alpha,Bravo,Charlie,Delta,Eagle,Foxtrot
                                    :  Integer;

procedure Swap(var S1,S2 : Integer);
var
   Temp                             :  Integer;
begin { proc Swap }
   Temp := S1;
   S1 := S2;
   S2 := Temp
end; { of proc Swap }

begin { main body of program TestSwap }
   Alpha := 1;   Bravo := 2;   Charlie := 3;
   Delta := 4;   Eagle := 5;   Foxtrot := 6;

   Swap(Alpha,Eagle);
   Swap(Bravo,Delta);
   Swap(Charlie,Foxtrot)

end. { of prog TestSwap }
```

A lot better, no? The integer variables *S1* and *S2* are no longer global variables but instead are *parameters* of the procedure

Swap. Each time *Swap* is called, the actual variables in the call (say, *Alpha* and *Eagle*) are used in place of *S1* and *S2*. Because of this, *S1* and *S2* are called *formal* or *dummy parameters*, while *Alpha* and *Eagle* are known as *actual parameters*.

There is one other twist in parameter lists. The prefix **var** before any formal parameter means that the corresponding actual parameter is directly substituted for the formal parameter. In other words, if the formal parameter has its value changed, the actual parameter is also changed. If there is no **var** prefix, then the actual parameter cannot be affected by anything that happens to the formal parameter. For example, here's a procedure that forces a variable to be within two values:

```
procedure Condition(Min : Integer; var Valu : Integer;
                                   Max : Integer);
begin
  if Min > Max
    then Swap(Min,Max);
  if Valu < Min
    then Valu := Min
  else if Valu > Max
    then Valu := Max
end; { of proc Condition }
```

Why do you want to have formal parameters that do not return values? Well, if you make the call *Condition(20,Alpha,0)*, you don't want the procedure to return a value to the constants "20" and "0", and, indeed, Pascal would not let you. If you defined Condition with the parameter list *(var Min,Valu,Max : integer)*, then you would get a compiler error with the above call. Why? Because the compiler knows that it cannot return a value to a constant or an expression, but only to a variable.

11.4 FUNCTIONS

As you've seen, a procedure can change the values of parameters passed to it. The calling program (or subprogram) can then use those modified parameters for whatever purpose it has. Suppose, for example, that you wrote a procedure to find the integer square root of a given value:

```
procedure ISqrt(Valu : Integer; var Root : Integer);
var
  OddSeq,Square              :  Integer;
begin
  OddSeq := -1;
  Square :=0;
  repeat
    OddSeq := OddSeq + 2;
    Square := Square + OddSeq
  until Valu < Square;
  Root := Succ(OddSeq div 2);
  if Valu <= Square - Root
    then Root := Pred(Root)
end; { of proc ISqrt }
```

This procedure would take *Valu*, find its square root, and set *Root* to that value. The calling program might use it as follows:

```
repeat
  Write('Enter value:    '); ReadLn(Able);
  ISqrt(Able,Baker);
  WriteLn('The square root is ',Baker)
until Able = 0;
```

There are many other cases where you might want to do something similar, that is, return a value based on certain variables or conditions. After a while, it can get a little clumsy and/or tiring to always set aside one parameter for the returned value. So, instead, you can use a *function*. A function acts just like a procedure, but with one difference: it returns a Real or a scalar value. (Brief review:scalar data types include Integer, Byte, Char, Boolean, and any defined scalar type—DST.) For example, you might rewrite *ISqrt* as follows:

```
function ISqrt(Valu : Integer) :   Integer;
var
  OddSeq,Square,Root        :  Integer;
begin
  .
  .
  .
  ISqrt := Root
end; { of func ISqrt }
```

You could now use it in your program like this:

```
repeat
  Write('Enter value:    '); ReadLn(Able);
  WriteLn('The square root is ',ISqrt(Able))
until Able = 0;
```

A function can be used anywhere that a constant or an expression of the same data type could be used. Suppose you wanted to find both the integer square- and fourth-roots of a given value. You could rewrite the program this way:

```
repeat
  Write('Enter value:    '); ReadLn(Able);
  WriteLn('The square root is ',ISqrt(Able));
  WriteLn('The fourth root is ',ISqrt(ISqrt(Able)))
until Able = 0;
```

When using a function, care must be taken to ensure that you have set it to some value before exiting. As shown above, you do this by assigning some value to the function name. You must also be careful about using the function name in any other way within the function itself. For example, you couldn't rewrite ISqrt to look like this:

```
procedure ISqrt(Valu : Integer; var Root : Integer);
var
  OddSeq,Square            :  Integer;
begin
  OddSeq := -1;
  Square :=0;
  repeat
    OddSeq := OddSeq + 2;
    Square := Square + OddSeq
  until Valu < Square;
  ISqrt := Succ(OddSeq div 2);
  if Valu <= Square - ISqrt
    then ISqrt := Pred(ISqrt)
end; { of proc ISqrt }
```

This would cause a compiler error. Why? Because Pascal considers the other uses of *ISqrt* in this statement to be additional calls to the function itself, and you don't have a parameter for each one. "Can a function call itself?", you might ask. Yes, it can...

11.5 RECURSIVE SUBPROGRAMS

As mentioned above, the Great Underlying Rule of Pascal is:*no identifier (constant, data type, variable, or subprogram) may be used unless it has been previously declared in the main program or in an enclosing subprogram.* The corollary is that all such identifiers may be used, which means that a subprogram can call itself. For example, let's suppose you're writing a graphics package and want a procedure which will fill a blank area on the screen. Let's also suppose that you've already written two other graphics subprograms: *PlotXY(X,Y)*, which plots a point at X,Y; and *Plotted(X,Y)*, a boolean function which returns *True* if the point X,Y has been plotted, and *False* otherwise. Here, then, is your procedure:

```
procedure Fill(X,Y : Integer);
begin
  if not Plotted(X,Y) then begin
    PlotXY(X,Y);          { plot this location }
    Fill(X+1,Y);          { check point to the right }
    Fill(X-1,Y);          { check point to the left }
    Fill(X,Y+1);          { check point below }
    Fill(X,Y-1)           { check point above }
  end
end; { of proc Fill }
```

Can you see how this works? Each time *Fill* is called, it checks to see if the location (X,Y) has been plotted. If it has, then *Fill* doesn't do anything. Otherwise, *Fill* plots a point at (X,Y) and then tries to fill in each of the four adjacent points to (X,Y).

One cautionary note about recursive subprograms: they can quickly use up a lot of memory, causing your program to behave erratically or to blow up. You could easily come up with a example where Fill would continue to call itself until you had over 100 levels of nested subroutine calls...and a lot of systems just can't handle that many levels of nesting. So be careful.

11.6 FORWARD DECLARATIONS

Are there any more situations where the Great Underlying Rule of Pascal gets in the way? Well, occasionally, you may wish to call a subprogram before it has been declared, out of convenience or necessity. For example, you might have the following program:

```
program  Example;
var
  Alpha                    :  Integer;
procedure Test1(var A : Integer);
begin
  A := A - 1;
  if A > 0
    then Test2(A)
end; { of proc Test1 }

procedure Test2(var A : Integer);
begin
  A := A DIV 2;
  if A > 0
    then Test1(A)
end; { of proc Test2 }

begin { main body of program example }
  Alpha := 15;
  Test1(Alpha)
end. { of proc Example }
```

As you can see, *Test1* calls *Test2*, and *Test2* calls *Test1*. As it stands, this program won't compile; you'll get an "Unknown identifier" error when it finds the reference to *Test2* within *Test1*. If you swapped *Test1* and *Test2*, then you'd get a similar error within *Test2*. So which do you declare first? The answer:both. How? By declaring *Test2* with a **forward** declarations:

```
program Example;
var
  Alpha                    :  Integer;

procedure Test2(var A : Integer); forward;
procedure Test1(varA : Integer);
begin
   .
   .
   .
end; { of proc Test1 }

procedure Test2;
begin
   .
   .
   .
end; { of proc Test2 }

begin { main body of program example }
   Alpha := 15;
   Test1(Alpha)
end. { of proc Example }
```

The *forward* declaration of *Test2* contains only that information necessary to resolve any references to it, namely its name and parameter list. The actual body of *Test2* (which, you will notice, no longer has a parameter list) occurs after *Test1*. Now, *Test1* can call *Test2* (because of the **forward** statement) and *Test2* can call *Test1* (since the latter precedes the former).

11.7 EXTERNAL SUBPROGRAMS

TURBO Pascal allows you to call assembly language procedures and functions that you have assembled separately. These are known as *external* subprograms; you can learn more about them in Chapter 22.

12. DECLARED SCALAR TYPES

In Chapter 9, we talked about four scalar data types: Integer, Byte, Boolean, and Char (Real isn't really a true scalar data type). As you may recall, *scalar* simply means that each data type has a fixed range of distinct values; for example, the data type Byte has 256 different values (0...255), while Boolean has exactly two (False and True). These four data types are very useful, but sometimes they aren't quite enough. Let's look at a brief example to see why:

```
program Example;
var
  Month,Day,DayOfWeek,Year          :  Integer;
begin
  Year := 1989;
  Month := 8;
  Day := 24;
  DayOfWeek := 3;
  .
  .
  .
end. { of program Example }
```

These values together represent some date. The year is obviously 1989, and the day is the 24th of some month—but what is the month, and what day of the week is it? You can assume (probably correctly) that the programmer is using standard numbering for the months (1=January, 2=February, etc.) and by some quick counting come up with "August". But the "day of the week" question is even harder to deal with. What day does the programmer assume is the first? Is that day numbered "0" or "1"? Unless you know the answer to those two questions, the statement "DayOfWeek := 3" doesn't really tell you anything.

This is a common problem in programming. You use a range of numbers—such as 1 through 12—to represent some corresponding set of values—such as January through December. But deep into the program, you lose track of what numbers represent what values. For example, the value you assigned *DayOfWeek* could easily represent Tuesday, Wednesday, or Thursday, depending upon whether you started the week on Sunday or Monday, and whether you used 0 or 1 to represent that first day. As it turns out, August 24th, 1989, falls on a Thursday—so your example assumes that Monday is the first day of the week and equals 0.

Now consider the following variation of the example:

```
program Example;
type
  Months        = (January,February,March,April,May,June,July,
                   August,September,October,November,December);
  Weekdays      = (Monday,Tuesday,Wednesday,Thursday,Friday,
                   Saturday,Sunday);
var
  Year,Day      :  Integer;
  Month         :  Months;
  DayOfWeek     :  Weekdays;
begin
  Year := 1989;
  Month := August;
  Day := 24;
  DayOfWeek := Thursday;
  .
  .
  .
end. { of program Example }
```

Now the code is completely clear as to the date and day of the week. All uncertainty and ambiguity is gone. Instead of having to second-guess the programmer, you can easily see what each statement means.

In defining Months and Weekdays, you have created two *declared scalar types* (DSTs). To make a DST, you simply (1) come up with a name for it, and (2) enter its list of values. The name and the values both follow the rules for creating identifiers: starts with a letter or underscore, followed by 0 or more letters, digits, and underscores; up to 127 characters long; case (upper- and lower-) doesn't matter; all characters (including underscores) are significant. The DST is defined in a **type** section; it has the format

```
   <name>       = (<first value>,...,<last value>);
```

The DSTs defined in the example were nice but not completely necessary. Since both months and days of the week are sequential lists, translation is easy once you know where to start. If you know that Monday is the first day of the week, and that its corresponding value is 0, then it's simple to equate "3" with "Thursday". But sometimes you work with arbitrary lists of values, values which don't really fall into an apparent order. Let's look at a second example:

```
program Example2;
var
  ShipClass              :  Integer;
begin
  ShipClass := 7;
  .
  .
  .
end.
```

Assuming that this is a portion of a "Trek" program, then *ShipClass* probably tells you something about what type of starship you are dealing with. The question is, what type *are* you dealing with? Unlike months and days of the week, there is no one way of ordering a list of different classes of starships. You can't even make a guess; you just don't have enough information. Now, let's look at the same example using a DST:

```
program Example2;
type
  ShipType               = (NoShip,Constitution,Enterprise,Reliant,
                             Loknar,Larson,Chandley,Excelsior,Baker);
var
  ShipClass              : ShipType;
begin
  ShipClass              := Reliant;
  .
  .
  .
end.
```

Once again, you now have no question as to what type of ship you're working with. Careful, clear definition of DSTs can greatly improve the readability of the programs you write.

Declared scalar types (and, indeed, all scalar types) are inherently ordered. In other words, there is a lowest value, a highest value, and some number of distinct values in-between. In your data type **Months**, January is the lowest value, December is the highest value, and February through November are the values in-between. Pascal will let you use these values just as you would integers, with one major exception: you can't do arithmetic with them. It's illegal (and meaningless) to write a statement like *ShipClass* := Reliant + Enterprise;. It is, however, perfectly legal to use DSTs in the following manner:

```
for ShipClass := Constitution to Baker do ...

if ShipClass >= Excelsior then ...

case ShipClass of
   NoShip                  :  ...
   Constitution,Enterprise :  ...
   Reliant                 :  ...
   Loknar,Chandley         :  ...
   Larson,Baker            :  ...
   Excelsior               :  ...
end;
```

In other words, DSTs can be treated almost like numeric values. As a matter of fact, each element in a DST has an implied numeric or *ordinal* value. The very first element has an ordinal value of 0; each following element then has a value one greater than the element before it. For example, here are all the elements of your DST ShipType, along with their ordinal values:

Element	Ordinal Value
NoShip	0
Constitution	1
Enterprise	2
Reliant	3
Loknar	4
Larson	5
Chandley	6
Excelsior	7
Baker	8

Pascal provides some functions to help you use this ordered relationship. Given a variable or constant value of some DST, you can find (1) the element that comes before it (its predecessor), (2) the element that comes after it (its successor), and (3) its numeric (ordinal) value. The first function is called **Pred**; the second, *Succ*; and the third, *Ord*. For example, let's say that you've set **ShipClass:** = Reliant. These functions will now yield the following results:

Pred(ShipClass)	Enterprise
Succ(ShipClass)	Loknar
Pred(Pred(ShipClass))	Constitution
Succ(Succ(ShipClass))	Larson
Pred(Succ(ShipClass))	Reliant
Succ(Pred(ShipClass))	Reliant
Ord(Pred(ShipClass))	2
Ord(ShipClass)	3
Ord(Succ(ShipClass))	4

Note that for all scalar types except Integer, *Ord* returns a value greater than or equal to 0, since all scalar types except integer start with an ordinal value of 0. For values of type Integer, *Ord* returns the actual integer value, which can fall anywhere in the range -32768 through 32767. Of course, using the *Ord* function on any Integer (or Byte) expression is a waste of time anyway, so the issue is not a critical one.

Here's another example. You've seen that you can use DSTs in **for** statements, but, if you wanted to, you could build your own loops:

```
ShipClass := NoShip;
repeat
   ShipClass := Succ(ShipClass);
   .
   .
   .
until ShipClass = Baker;
```

This loop would execute once for each element from Enterprise to Baker. If you wanted to go in reverse order, you could write

```
ShipClass := Baker;
while ShipClass > NoShip do begin
   .
   .
   .
   ShipClass := Pred(ShipClass)
end;
```

which would start at Baker and go downwards to Enterprise. Of course, you could achieve the same effect with

```
for ShipClass := Baker downto Enterprise do begin
   .
   .
   .
end;
```

There is a subtle point to notice in your definition of the DST ShipType. Even though you are dealing with eight types of ships, you've defined nine elements; the extra one is your first element, **NoShip**. When you define a DST, you should consider putting in the local equivalent of **0**; that is, some element that essentially means "none of the above". There are two good reasons for this. First, you will find yourself in situations where you really *are* dealing with none of your regular values. For example, you might have a list of ships with which you are attacking:

```
var
  Ship        :  array [1..10] of ShipType;
```

(If you're not familiar with what an array is, don't worry—you'll run into them later. Once you know what they are, come back and look at this again, and it'll all make a lot more sense.) Suppose that *Ship*[5] := Reliant, but that during the course of the game, that ship gets destroyed. How do you keep track of that information within the program? You have to come up with some means of remembering that *Ship*[5] is gone. If you've included NoShip in your definition of *ShipType*, then you can just set *Ship*[5] := NoShip, and presto! it disappears. This leads to the following loop, which will ignore all destroyed ships (assume Indx is of type Integer):

```
for Indx := 1 to 10 do
  if Ship[Indx] <> NoShip then begin
    .
    .
    .
  end;
```

The code in the center of the loop will be executed for each existing ship; all others will be ignored.

The second subtle reason of a "none of the above" element was demonstrated earlier, when you created your own loops with **while** and **repeat** statements. In both cases, you either started or ended with *ShipClass* equal to *NoShip*; in other words, *NoShip* acted as a "doorstop" of sorts. Without it, you would have had a much harder time setting up the loops without going out of range, either with *Succ(ShipClass)*, where *ShipClass* = Baker, or *Pred (ShipClass)*, where *ShipClass* = Constitution. The result is either a range error (if range checking is turned on) or an undefined value for *ShipClass* (if range checking is turned off). Either case can cause problems.

There are times when you don't need (or want) a "none of the above" element. Take, for example, your DST Months. If you confine your loops to **for** statements, then you don't need to define a *NoMonth* element. On the other hand, it could still be useful for two reasons. First, it would allow you to have a true "undefined" date, rather than just some default date (1/Jan/00) that was understood to be undefined. Second, it would force the ordinal values of the elements to match the conventional values used; *Ord(April)* would equal 4 instead of 3. The lesson, then, is think carefully about how you define your DSTs.

TURBO Pascal gives you an additional feature for working with declared scalar types, one not found in Standard Pascal. You can convert any scalar type to any other scalar type with the same numeric (ordinal) value. For example, you know that *Ord(Reliant)* give you the Integer value 3; by the same token, *ShipType(3)* returns the *ShipType* value *Reliant*. Likewise, *Boolean(NoShip)* returns *False*, and *ShipType(False)* returns *NoShip*, since *Ord(NoShip)* = Ord(False).

There are two things that you would probably like to do with DSTs, but which you can't: directly read and write them. For example, the statement

```
    WriteLn('Ship Class:    ',ShipClass);
```

will produce an error when you compile it. The same is true for a statement such as

```
    ReadLn(ShipClass);
```

There are, however, some ways around this limitation. If you are interested in writing out elements of a DST, you will need to create an array (list) of strings (yes, we'll talk more about both strings and arrays later). You could do the following:

```
program example3;
type
  ShipType      = (NoShip,Constitution,Enterprise,Reliant,
                   Loknar,Larson,Chandley,Excelsior,Baker);
var
  ShipClass     : ShipType;
  ShipName      : array [ShipType] of string[12];
```

```
begin
  ShipName[NoShip]       :=  'No ship';
  ShipName[Constitution] :=  'Constitution';
  ShipName[Enterprise]   :=  'Enterprise';
   {etc.}
  ShipName[Baker]        :=  'Baker';
  .
  .
  .
  WriteLn('Ship Class:   ',ShipName[ShipClass]);
  .
  .
  .
end.
```

Now you can write out the "value" of *ShipClass* by printing the appropriate string in *ShipName*. Similar arrays can be set up for each DST that you want to be able to write out. If you have several lists like this, you may want to store the strings out in a text file (and you'll read about those later on, as well) and then read them in at the start of the program. In that case, your code might look like this:

```
var
  ShipClass :ShipType;
  ShipName  :array [ShipType] of string[12];
  InFile    :Text;
begin
  Assign(InFile,'SHIPNAME.DAT');
  Reset(InFile);
  for ShipClass := NoShip to Baker do
    ReadLn(InFile,ShipName[ShipClass]);
  Close(InFile);
  .
  .
  .
end.
```

This assumes that you have a text file SHIPNAME.DAT out on the disk with the contents:

```
No ship
Constitution
Enterprise
(and so on...)
```

Using a separate data file has two advantages. First, it reduces the size of the source and executable files, often by quite a bit if you have long arrays to ilitialize. Second, it allows you to change the string associated with each element without having to recompile the program.

You have several options for reading in DST values. The first is to make use of the array you've set up for writing the DST value out (if, indeed, you have set up such an array). You prompt the user for a string and search though the array until you've found it:

```
var
  TempStr : string[12];
  Found   : Boolean;
  { and the rest of the declarations }
begin
  .
  .
  .
  Write('Enter Ship Class:'); ReadLn(TempStr);
  Found := False; ShipClass := Baker;
  while not found and (ShipClass > NoShip) do
    if ShipName[ShipClass] = TempStr
      then Found := True
      else ShipClass := Pred(ShipClass);
  .
  .
  .
end.
```

Here, you start at the end of the array and work backward to (but not including) *NoShip*. When you find the matching string, *Found* gets set to *True*, and you fall out of the loop with *ShipClass* set to the correct value. If *ShipClass* gets to *NoShip*, then you fall out of the loop with *Found* = False and must take appropriate action. You could, for example, put this code with the loop

```
repeat
  .
  .
  .
until Found;
```

which would force the user to keep entering the string until one matched.

This is a nice, straightforward method, and one that meshes nicely with your output routine. The only real problem is one of matching cases (as in "UPPER" and "lower"). For example, if the user enters 'RELIANT', but *ShipName*[*Reliant*] = 'Reliant', then the two aren't going to match, and *Found* will remain *False*. So, you'll need to put some code in right after *"ReadLn(TempStr);"* to convert *TempStr*'s case to match that of the strings in ShipName. In your example, you would need to convert *TempStr*[*1*] to upper case and the rest of *TempStr* to lower case.

If you don't want to go to all that trouble—and especially if you don't want or need *ShipName* for output—then there are other ways of reading in values and converting them to DSTs. You could, for example, take advantage of the retyping functions in TURBO Pascal by having the user enter the ordinal value of the element desired, then converting it to the appropriate element:

```
var
  Indx      :Integer;
  ShipClass :ShipType;
begin
  .
  .
  .
  WriteLn('Ship Classes:');
  for ShipClass := NoShip to Baker do
    WriteLn(ShipName[ShipClass],':    ',Ord(ShipClass));
  repeat
    Write('Enter value:    '); ReadLn(Indx)
  until (Ord(NoShip) <= Indx) and (Indx <= Ord(Baker));
  ShipClass := ShipType(Indx);
  .
  .
  .
end.
```

This example assumes that you did set up *ShipName* but decided to use numeric (rather than string) input. It first writes out the names of the different ship classes, along with their ordinal values. It then goes into a **repeat** loop which won't let the user out until a correct value is entered. Finally, it converts that value to the appropriate *ShipType* element and assigns it to ShipClass.

There are other methods, such as prompting for a single character, then using a **case** statement to convert from the character to the DST element, but we'll leave those for now. The upshot is this: declared scalar types can tremendously aid program development, documentation, and maintenance, especially if you think carefully in how they are declared and used.

12.1 SUBRANGES

Besides creating brand-new scalar types, you can also define *subranges* of existing (or newly-declared) scalar types. TURBO Pascal already defines one such subrange: the type Byte, which is a subrange of Integer. To declare a named subrange, you simply use the form:

```
type
SubRange= FirstVal..LastVal;
```

where *FirstVal* and *LastVal* are values of a scalar type such that *FirstVal* is less than *Lastval*. The data type *SubRange* can now legally have any value from *FirstVal* through *LastVal*. Subranges can also be directly declared, that is, a given variable can be directly declared as having a subrange value, as opposed to having to first declare the subrange. Here are some examples of subranges (with some accompanying declarations):

```
const
  XMax       = 8;
  YMax       =  10;
  ShipMax    =  40;
type
  Days       =  (Mon,Tues,Wed,Thurs,Fri,Sat,Sun);
  ShipType   =  (NoShip,Constitution,Enterprise,Reliant,
             Loknar,Larson,Chandley,Excelsior,Baker);

  Cruiser    =  Constitution..Reliant;
  Nibble     =  0..15;
  UpperCase  =  'A'..'Z';
  LowerCase  =  'a'..'z';
  WeekDays   =  Mon..Fri;
  WeekEnd    =  Sat..Sun;
  XRange     =  1..XMax;
  YRange     =  1..YMax;
```

```
var
  SX         :  XRange;
  SY         :  YRange;
  ShipCount  :  0..ShipMax;
  Alphas     :  ' '..'~';
```

Subranges are often used as index ranges for arrays, as data types for both arrays and records, and as part of the declaration of a constant set. In all these functions, they serve to limit the amount of RAM used and to set boundaries as to what values can be used.

12.2 DIRECT DECLARATIONS

The example above showed that you can declare subranges directly; in other words, you can write

```
var
  ShipCount    :0..ShipMax;
```

rather than

```
type
  ShipRange    =0..ShipMax;
var
  ShipCount    :ShipRange;
```

Being able to directly declare subranges like that saves you from having to come up with names for every subrange that you want to use. What you may not realize is that you can do the same for declared scalar types, and, indeed, for any data type (array, string, record, set, pointer, file). However, there are a few restrictions on doing this. First, all of the variables of that data type have to be declared at the same time and in the same place. You can't write

```
var
  Day1       : (Mon,Tue,Wed,Thu,Fri,Sat,Sun);
  Day2       : (Mon,Tue,Wed,Thu,Fri,Sat,Sun);
```

You'll get an error when the compiler hits the second declaration, since the second set of days uses the same identifiers as the first set. Instead, you would have to write

```
var
  Day1,Day2          : (Mon,Tue,Wed,Thu,Fri,Sat,Sun);
```

Second, you won't be able to pass any such variables as parameters to a subroutine or procedure, since you won't have a data type as which to declare the formal parameter. (There is a limited exception to this—untyped **var** parameters—but that's a special case.) Finally (and this is specific to DSTs) you won't be able to do any retyping, since there isn't a DST identifier to use.

Well, this is the end of the chapter on scalars. If you take a look at the program diskette supplied with TURBO Pascal Tutor, you will find a program called SCALARS.PAS. You should study this source code and make sure you understand how it works. Then compile it (by using your TURBO Pascal diskette, of course) and see how it runs.

13. ARRAYS

In previous sections, you've learned about the five predefined data types—*Integer, Byte, Real, Boolean*, and *Char*—as well as declared scalar types (*DSTs*). A variable of one of these types can hold only one value at a time. For example, if you define:

```
var
  Index        : Integer;
```

then *Index* has only one specific value at any moment. However, there are situations where you'd like to have a list of values, such as a list of numbers or characters. That's where **arrays** come in.

We've talked some about the "Trek" game, so that will be the example here. Let's suppose that the game area is an 8 by 10 grid. Each square in the grid is called a sector. Each sector contains (1) 0 or more stars, (2) 0 or more enemy ships, and (3) possibly one starbase. Now, with what you've learned in previous chapters, how would you represent the grid?

Tough, huh? Now you see why we need arrays. Simply put, an array is a collection of variables of identical type, each one of which may be referenced by a unique index value. For example:

```
var
  List        : array[1..10] of Integer;
```

The array List is a collection of 10 integer variables, namely, *List*[*1*], *List*[2], *List*[3], *List*[4], *List*[5], *List*[6], *List*[7], *List*[8], *List*[9], and *List*[*10*]. You can use any of those 10 variables anywhere you could use any integer variable. Furthermore, the index value doesn't have to be a literal value ('1' through '10'); it can be any expression that resolves to an integer in the range 1..10. For example, the statement

```
for Index := 1 to 10 do
  List[Index] := 0;
```

would set each of those ten variables to 0.

An array definition takes the form

```
    array[index range] of data type;
```

The *index range* can be an implicit or explicit subrange of any scalar data type (integer, char, boolean, DST), subject, of course, to certain limits. For example, it is extremely unlikely that you would want to define

```
var
  BigArray    :  array[Integer] of Char;
```

since that would allocate space for some 65,000+ characters; the implied subrange being -32,768...32767 for TURBO Pascal. Implicit subranges are usually used with DSTs; for example:

```
type
  Days         =  (Sun,Mon,Tues,Wed,Thur,Fri,Sat);

var
  Regular     :  array[Mon..Fri] of Integer;
  Overtime    :  array[Days] of Integer;
  Present     :  array[Days] of Boolean;
```

The array *Regular* has an explicit index range (Mon..Fri), while the arrays *Overtime* and *Present* have an implicit index range (Sun...Sat).

The *data type* of any array can be almost any data type- Integer, Boolean, Char, Real, DST, arrays, records, sets, pointers, strings, or even files. In fact, arrays with multiple index ranges (known as *multi-dimensional arrays*) are just arrays of arrays (of arrays...). For example, you might define your game information as follows:

```
var
  Stars,Ships,Base :   array[1..8] of array[1..10] of Integer;
```

To reference the elements of such an array, you can write statements like these:

```
Stars[3][2] := 0;

if Base[X][Y] = 0
  then WriteLn('No starbase present')
  else RefuelShip;

Danger := 5 * Ships[X][Y];
```

In writing this, you are selecting an array, then a particular element of that array; for example, *Ships*[*X*][*Y*] refers to the *Yth* element of the *Xth* array. Both notations can get a little tiring, especially for arrays with three or more index ranges, so Pascal allows the following shorthand:

```
var
  Stars,Ships,Base      :  array[1..8,1..10] of Integer;

Stars[3,2] := 0;

if Base[X,Y] = 0
  then WriteLn('No starbase present')
  else RefuelShip;

Danger := 5 * Ships[X,Y];
```

This makes for simpler code, but it can hide the fact that you are working with arrays of arrays. Consider another solution for your game using a 3-dimensional array:

```
type
  SectItem      = (Stars,Ships,Bases);
  SectArray     = array[1..8,1..10] of Integer;
var
  Sector  :       array[SectItem] of SectArray;
  Temp      :       SectArray;
```

Now, you could refer to the number of stars in Sector 3-2 as *Sector*[*Stars,3,2*], *Sector*[*Stars*][3,2], or *Sector*[*Stars*][*3*][2]. What you might not realize is that you could also do the following:

```
    Temp := Sector[Stars];
```

This single statement copies all of the values *Sector*[*Stars,1,1*] through *Sector*[*Stars,8,10*] into the locations *Temp*[*1,1*] through *Temp*[*8,10*]. Why? Because both *Sector*[*Stars*] and *Temp* are defined as being of type *SectArray*, and TURBO Pascal will allow direct assignment of identical array types. In fact, any operation you could do on Temp, you could also do on *Sector*[*Stars*], *Sector*[*Ships*], and *Sector*[*Bases*]. The key to making this work is to define an array type—such as *SectArray*—and then using that in all the appropriate declarations.

This approach has some real advantages, but other considerations are needed. For example, each and every location in *Sector* is an integer, 16 bits' worth of information yielding over 65,000 values. But you don't really need all that space in each location. You only need one bit (yes or no) for base information, and only a few bits more each for ships and stars. So, if you're concerned about space, this might be a better solution to the problem:

```
const
  ShipMax   =  15;
  StarMax   =  9;
  XMax      =  8;
  YMax      =  10;
type
  XRange    =  1..XMax;
  YRange    =  1..YMax;
var
  Stars     :  array[XRange,YRange] of 0..StarMax;
  Ships     :  array[XRange,YRange] of 0..ShipMax;
  Base      :  array[XRange,YRange] of Boolean;
```

This approach has two advantages. First, it prevents you from getting an "illegal" value for the numbers of stars, ships, or bases in a given sector. If *Ships[3,2]* is ever less than 0 or greater than *ShipMax*, you'll get a range error, which will tell you that something has gone wrong somewhere. Second, it will reduce the amount of memory required for your arrays. The array Sector requires 480 bytes of memory (RAM), while the arrays*Stars, Ships,* and *Base* require a total of just 240 bytes —only half the amount. The difference may not seem like much, but if you have lots of arrays—or if your arrays are large—the savings can be significant.

Another (less preferred) way of indexing is to use the {$R+} compiler option to flag "out of range" indexing errors. The drawback to this method is that it uses more space and slows down compilation.

Yet another solution would be to define an array of records; however, that will have to wait until you reach the appropriate section in this tutorial.

13.1 PACKED ARRAYS

The discussion on storage space brings up another issue. Standard Pascal defines defines two kinds of arrays:regular arrays and *packed* arrays. It also provides two procedures, *Pack* and *Unpack*, to convert between the two types. TURBO Pascal doesn't distinguish between the two, though it allows (and ignores) the keyword *Packed.* Instead, TURBO Pascal automatically attempts to pack all arrays, that is, store them in the smallest possible space. The lowest space used is one byte per element; for example, the array *Base* sets aside one byte for each location, even though it technically needs only one bit. Why the extra space? It allows any element of an array to be passed as a **var** parameter to a procedure or function. Other Pascal implementations don't allow elements of packed arrays to be used in that manner. One more point: since they aren't needed, the procedures *Pack* and *Unpack* aren't defined.

13.2 ARRAY INITIALIZATION

Often you will want to initialize an array, setting all of its elements equal to a single value. For example, suppose that you wanted to set all of the elements in *Base* to *False*, so that you could later set just the ones desired to *True*. One way of doing this would be:

```
for X := 1 to XMax do
  for Y := 1 to YMax do
    Base[X,Y] := False;
```

Of course, this takes a while to do and uses up a bit of space for code and variables. On the other hand, you could make use of two built-in TURBO Pascal procedures. The first is *FillChar*, which takes the format

```
    FillChar(Dest,Length,Data);
```

where *Dest* is the variable (of any sort) to be filled, *Length* is the number of bytes to initialize, and *Data* is the value to which to set each byte (and can be expressed either as a character or as a byte value). You know what you want to fill—*Base*—and you know what you want to fill it with—0, which is *Ord(False)*, that is, the numeric equivalent of *False*. Now you just need the length in bytes . . . which brings you to the next procedure: *SizeOf. SizeOf* can take as its argument any variable or the name of any data type.It returns the size of that variable (or of a variable of that type) in bytes. So, to initialize Base, you could write:

```
FillChar(Base,SizeOf(Base),0);
```

This statement will set all bits and bytes in *Base* = to 0. The combination of *FillChar* and *SizeOf* is a hard one to beat, especially for array initialization.

13.3 ORDER OF ELEMENTS

The elements of an array are stored in a specific order. The order is different for CP/M-80 systems than for all other version of TURBO Pascal. The following description applies to all versions of TURBO Pascal *except* CP/M-80:

If the array is one-dimensional—that is, if it has only one index— then the elements are stored in ascending order. For example, the array *List* (defined as **array**[0..9] **of** Integer) stores its elements in the order *List*[*0*], *List*[*1*], *List*[*2*], and so on... basically what you would expect. But what about multi-dimensional arrays? The array *Stars* is defined as

```
var
  Stars       : array[XRange,YRange] of 0..StarMax;
```

where *SRange* = 1...8. So, the question is, are the elements in Stars stored as *Stars*[*1,1*], *Stars*[*2,1*], *Stars*[*3,1*], etc., or are they stored as *Stars*[*1,1*], *Stars*[*1,2*], *Stars*[*1,3*], and so on? Pascal itself gives you the answer to that question. Remember that your definition above is just shorthand for

```
array[XRange] of array[YRange] of 0..Starmax;
```

In other words, *Stars'* first index doesn't select an element, it selects an **array**[YRange] **of** 0..Starmax. The second index selects an element within that array, and those elements are stored sequentially, just as in *List*. So *Stars*[*1,1*] says to pick the first element of the first array; *Stars*[*1,2*], the second element of the first array; and so on. So the elements in *Stars* are stored in the order

```
Stars[1, 1]
Stars[1, 2]
Stars[1, 3]
Stars[1, 4]
Stars[1, 5]
Stars[1, 6]
Stars[1, 7]
Stars[1, 8]
Stars[1, 9]
Stars[1,10]
Stars[2, 1]
Stars[2, 2]
   .
   .
   .
Stars[8, 9]
Stars[8,10]
```

All you need to do is remember that the index furthest to the right—the last index—changes the fastest. If you have the array

```
var
  BigOne      :  array[0..3,0..4,0..5,0..2] of Byte;
```

then you can quickly work out that the elements are stored as

```
BigOne[0,0,0,0]
BigOne[0,0,0,1]
BigOne[0,0,0,2]
BigOne[0,0,1,0]
   .
   .
   .
BigOne[3,4,5,1]
BigOne[3,4,5,2]
```

The above description generally applies to CP/M-80 systems, with the following important difference:

The elements of the array are stored in *descending* rather than ascending order. When you address the array, the first element is located just as in other versions; however, additional elements will be found in descending memory locations rather than in ascending memory locations

14. STRINGS

—String—

When Niklaus Wirth designed Pascal, he did so in a punched-card/mag tape/mainframe environment, where fixed-length data were the rule. At least, that's probably the reason he was satisfied to store a character string as a **array**[1..n] **of** Char. At any rate, Standard Pascal does not (currently) have a predefined data type for strings. String constants such as

```
const
  FileName     =  'B:STARS.DAT';
  LifeName     =  'WHATEVER YOU WANT';
```

are considered to be **array**[1..n] **of** Char, where **n** equals the number of characters in the string.

The early mainframe (that is, very large) computers were *batch-oriented* systems. "Jobs" were "submitted" using large decks of punched cards or reels of magnetic tapes, and the results came out on a high-speed line printer. But the arrival of interactive operating systems ("timesharing") and the CRT terminal started a new approach to computer use. When the minicomputer showed up, so did the first code designed to interactively manipulate text: word processing programs. The explosive growth of the microcomputer market over the last 10 years has been matched by an equal growth in word processors and the number of people using them. For all its reputation as a number-crunching machine, the computer is used most often to move words, not values.

The basic concept behind text manipulation is that of a *string*. A string is simply a sequential list of characters of some length. For our purposes, the characters belong to the ASCII character set (the back of the **TURBO Pascal Reference Manual** gives you the list). The string can contain letters, digits, and punctuation. It can even have non-printing (control) or special characters. You can pick out parts of the string, add to it, take away from it, combine it with other strings, print it out, read it in—in short, you can manipulate it.

TURBO Pascal allows you to declare a variable to be of type **string**, followed by a length specification. For example, you could define the following:

```
var
  MyName    :  string[80];
  Token     :  string[15];
  BigString :  string[255];
```

Note that you must specify a length for each variable. This defines the maximum number of characters that each string can hold. The variable *MyName* could hold up to 80 characters. *Token* could only hold up to 15 characters, so that the statement

```
    Token := 'this is too long a string for token';
```

would only store the first 15 characters ('this is too lon') into *Token*. The last variable, *BigString*, represents the maximum length possible for a string—255 characters.

The data type **string**[n] can be thought of as an **array**[0..n] **of** Char. For example, you can reference individual characters in Token using the notation *Token*[*1*], *Token*[2], and so on. The first location, *Token*[*0*], contains the current length of *Token*. If you execute the statement

```
    Token := 'this string';
```

then *Token*[*0*] contains the value 11, since there are 11 characters in'this string'. However, you could not do something like this:

```
program Example;
var
  Token     :  string[15];
  Len       :  Integer;
begin
  Token:  = 'this string';
  Len := Token[0];
  WriteLn('The length of token is  ',Len)
end.
```

Why not? Because *Token*[*0*] is of type *Char*, and you can't assign a character to an integer. You could, however, substitute the statement

```
    Len := Ord(Token[0]);
```

which would return the ordinal (numeric) value of *Token*[*0*], which happens to be 11.

Using the array notation, you can play with any individual character of a string. As mentioned above, each element of a string is a variable of type *Char*, and you can treat it as such. For example, you might want a procedure to convert all letters in a string to uppercase ('A'..'Z'):

```
procedure LowToUp(var Str : string[255]);
{
    purpose       converts characters in Str to upper case
    last update   27 Oct 85
}
var
  Indx,Len        : Integer;
begin
  Len := Length(Str);      { more on this later }
  for Indx := 1 to Len do
    Str[Indx] := UpCase(Str[Indx]) {built-in TURBO func}
end; { of proc LowToUp }
```

A caveat (warning) is in order. You should avoid trying to mess with any elements beyond the current length of the string. TURBO Pascal won't give you any sort of error, but you need to be aware that you've just changed a portion of the string that won't print out unless you change the length as well. (Note that if you use the {$R+} compiler directive, and error message will display.)

You've already seen that you can assign string constants to string variables (*Token* := 'this string'). You can also read and write strings through textfiles (which will be discussed in a later section). Most notably, you can read and write strings through the predefined textfiles, *Input* and *Output*. For example, this program allows you to type in a line (up to 80 characters!) and then writes it back out to the screen. It continues to do this until you type, "I quit!":

```
program Echo;
var
  Line            : string[80];
begin
  WriteLn('Entering echo mode—type "I quit!" to exit');
  repeat
    ReadLn(Line);
    WriteLn(Line)
  until Line = 'I quit!'
end. { of program Echo }
```

Now, let's make the following modifications to your program (don't worry about the file stuff; yes, you'll learn about that later, too):

```
program MicroWord;
var
  Line       :  string[80];
  OutFile    :  Text;
begin
  Write('Enter file name:   '); ReadLn(Line);
  Assign(OutFile,Line); Rewrite(OutFile);
  WriteLn('Entering insert mode—type "I quit!" to exit');
  repeat
    ReadLn(Line);
    if Line <> 'I quit!'
      then WriteLn(OutFile,Line)
  until Line = 'I quit!';
  Flush(OutFile); Close(OutFile)
end. { of program MicroWord }
```

Voila! You've just written a *word processor*. However complex or sophisticated they may seem, all word processing programs eventually boil down to the program above. Start with this program, add modifications, and eventually you'll have your own text editor.

14.1 STRING COMPARISONS

Just like numbers, strings can be compared to each other. In the program MicroWord, you checked to see whether the string variable *Line* was equal to the string constant 'I quit!'. The comparison is simple. First, the lengths of the two strings are compared. If they're different, then the strings are not equal. If they're the same, then the characters in the two strings are compared, starting with the first one and continuing until (1) two characters are different or (2) all characters have been compared. In case (1), the strings are not equal; in case (2), they are. You could even write a function to show this comparison:

```
function StrEqual(Str1,Str2 : BigStr) : Boolean;
{
    purpose      show how strings are compared for equality
                 Note well:this function is *not* necessary,
                 since "Str1 = Str2" will perform the same
                 comparison.
}
var
  Len,Indx      :  Integer;
  Flag          :  Boolean;
begin
  StrEqual := False;
  Len := Length(Str1);
  if Len = Length(Str2) then begin
    Indx := 1;
    Flag := True;
    while Flag AND (Indx <= Len) do
      if Str1[Indx] = Str2[Indx]
        then Indx := Indx + 1
        else Flag  := False;
    StrEqual := Flag
  end
end; { of func StrEqual }
```

Once again, please understand that you do *not* need this function. This just shows how the Boolean expression *Str1* = Str2 comes up with a value of *True* or *False*.

Similar comparisons occur when you want to see if one string is "greater than" or "less than" another. For example, let's suppose you're sorting a list of names into alphabetical order. At some point, you'll compare two strings to find which comes before (is less than) the other. The statement

```
    if Str1 > Str2 then ...
```

will take some action if and only if *Str2* comes before *Str1*. The comparison algorithm can be described as follows:

1. Point to the first character of each string

2. Compare the two characters

3. If they are not the same, go to 8.

4. Get the next two characters
5. If they're both there, go to 2.
6. If only one string has characters left, then it is greater than the other
7. Otherwise, neither string has characters left, so neither is greater than the other (they're identical!). Stop
8. One character has a greater ASCII value than the other. The string that character came from is greater than the other string. Stop

To summarize, you have three cases. First, both strings are the same length and have the same contents. In that case, they're equal. Second, both strings have the same contents up to the end of one string; the other string has additional characters beyond that point. In that case, the shorter string is less than the longer one. In the last case, the strings cease to match at some point (it may even be the first character). In that case, the string whose unmatched character has the lower numeric (ordinal) value is less than the other string.

14.2 STRING FUNCTIONS AND PROCEDURES

Another useful aspect the TURBO Pascal string definition is that it includes more that just the data type *string*. It also defines a set of functions and procedures which work on strings. The table at the end of this chapter lists all of them, but we'll go over each one at a time (and not necessarily in the order given).

The most commonly-used function is probably *Length(St)*, which was used in a few of the examples above. It's really just another way of writing *Ord(St*[*0*]*)*; that is, it returns the current length of *St*. This is not to be confused with the maximum possible length of *St*. For example, the program

```
program LengthTest;
type
   SmallStr        =   string[15];
var
   Test            :   SmallStr;

procedure ShowLength(St : SmallStr);
begin
   WriteLn('length of <',St,'> is ',Length(St))
end; { of proc ShowLength }

begin
   Test := 'hello, there';
   ShowLength(Test);
   Test := 'hi';
   ShowLength(Test);
   Test := '';{ null string }
   ShowLength(Test)
end. { of program LengthTest }
```

will produce the output

```
     length of <hello, there> is 12
     length of <hi> is 2
     length of < > is 0
```

The next most commonly-used function is probably *Concat*. You can use it to patch several strings together. It's handy for inserting string variables in the middle of fixed messages. For example, this program

```
program Concatenation;
var
   Name,Message:   string[30];
begin
   Write('Please enter your name:');
   ReadLn(Name);
   Message := Concat('Hello, ',Name,' how are you?');
   WriteLn(Message)
end. { of program Concatenation }
```

produces this output:

```
Please enter your name:   Deirdre
Hello, Deirdre, how are you?
```

Besides the explicit function *Concat*, TURBO Pascal also lets you piece together strings using the plus sign (+). For example, you could change the third statement in your program above to read

```
Message := 'Hello, ' + Name + ', how are you?';
```

This has exactly the same effect as the *Concat* function, and you may find it easier to use.

There are two things you have to be aware of when concatenating strings. First, as mentioned above, if the resulting string is longer than the variable can hold, the extra characters will be thrown away. More importantly, if the resulting string is longer than 255 characters, you'll get a run-time error, and your program will come to a screeching halt (unless you have used the {} compiler option, in which case you would have gotten an error during compilation).

Right up there with *Concat* is the function *Copy*, which allows you to pull out part of a string (called a *substring*). It takes as parameters the string itself and the substring's location and length. The code

```
Name := 'Deirdre Ann Webster';
Middle := Copy(Name,9,3);
WriteLn('Your middle name is ',Middle);
```

would produce the output "Your middle name is Ann".

Of course, the example above depended upon knowing right where 'Ann' started in the string. Suppose you knew what string you were looking for, but didn't know where it started? You could use the function *Pos* to find it. Given a substring or pattern, and the string in which to search for it, *Pos* returns the location of the start of the substring. If it can't find the substring, *Pos* returns 0. Your code could now look like this:

```
Name := 'Deirdre Ann Webster';
Loc := Pos('Ann',Name);
Middle := Copy(Name,Loc,3);
WriteLn('Your middle name is ',Middle);
```

The *Delete* procedure lets you remove a section of a string. Like *Copy*, it requires the string, the starting position, and the number of characters to delete. For example, you could change the code above to read

```
Name := 'Deirdre Ann Webster';
Loc := Pos('Ann',Name);
Delete(Name,Loc,4); { need to cut out an extra blank }
WriteLn('Your name is ',Name);
```

which would then write out "Your name is Deirdre Webster", having deleted the string 'Ann '.

Combined with *Pos* and *Copy, Delete* can be used to *parse* a string, that is, to pull off a chunk at a time. Let's suppose that you want to write a procedure that will pull the first word off a string, where a "word" is defined as any substring starting with a non-space character and followed by a **space**. Our procedure might look like this:

```
procedure Parse(var Line,Word : BigStr);
{
    purpose      removes first word in <line> and returns it in <word>
}
const
  Space          = ' ';
var
  Indx,Len       : Integer;
begin
  while Pos(Space,Line) = 1 do{ remove leading blanks }
    Delete(Line,1,1);
  Len := Pos(Space,Line);          { look for blank }
  if Len = 0 then begin { no blanks left }
    Word := Line;          { get word }
    Line := ''             { zero out line }
  end
  else begin { get word and delete from line }
    Word := Copy(Line,1,Len-1);          { get all but blank }
    Delete(Line,1,Len)                   { delete word plus blank }
  end
end; { of proc Parse }
```

The next procedure, Insert, is the reverse of the *Copy/Delete* operation: it takes one string and stuffs it somewhere inside another. You just specify the string you want to insert, the string into which it's to be inserted, and the location of the the insertion. For example, if you wanted to put Deirdre's middle name back in, you could use the following code:

```
    Name := 'Deirdre Webster';
    Middle := 'Ann';
    Insert(Middle,Name,Pos('Webster',Name));
    Insert(' ',Name,Pos('Webster',Name));
    WriteLn('Your full name is ',Name);
```

Note that you use the *Pos* function to figure out where to insert 'Ann' and also where to insert a blank (to separate 'Ann' and 'Webster').

Token expansion is one good use for *Insert*. Suppose you were writing a program to take a form letter and put in the appropriate names, dates, and so on. Within the form letter, these fileds might be represented by *tokens*; for example, the salutation might look like this:

```
   Dear <title> <last name>:
```

The program could scan through the form letter, list all the tokens (which all have the form <...>), then get the information to replace them (interactively, from a file, etc.). The following procedure, then, might be of use:

```
procedure Replace(var Line : BigStr; Token,Sub : TokStr);
{
   purpose         look for Token in Line and replace with Sub
}
var
  Indx,Len         : Integer;
begin
  repeat
   Indx := Pos(Token,Line);
   if Indx > 0 then begin
     Delete(Line,Indx,Length(Token));
     Insert(Sub,Line,Indx)
   end
  until Indx = 0
end; { of proc Replace }
```

The statements

```
   Line := 'And so, <title> <last>, the entire <last> family';
   Replace(Line,'<title>','Dr.');
   Replace(Line,'<last>','Lewis');
   WriteLn(Line);
```

would produce

```
   And so, Dr. Lewis, the entire Lewis family
```

This should give you a clue on how all that "personalized" junk mail that you receive is generated.

14.3 NUMERIC CONVERSIONS

TURBO Pascal provides two procedures for converting numbers to strings and vice versa. These two procedures work in a fashion similar to Read and Write, which you've had some exposure to and which will be discussed more in a later section. The first procedure, *Str*, will convert a number into a string, formatting it much as it would for text output. The number can be either *Integer* or *Real*, and you can specify the width format just as you can for output. Here are some calls to *Str*, along with the resulting strings (for our purposes, X = 4.281953E3 and I = 14916; S is defined as **string** [12]):

```
call to Str          contents of S

Str( X:12:3,S);     "    4281.953"
Str( X:12:0,S);     "        4282"
Str( X:10:7,S);     "4281.9530000"
Str(-X:12:5,S);     " -4281.95300"
Str( X: 5:4,S);        "4281.9530"
Str( I:12,S);       "       14916"
Str( I: 5,S);       "14916"
Str(-I: 7,S);       "-14916"
Str( I: 3,S);       "14916"
```

Notice that *Str* does, indeed, behave like numeric output. The string length is set equal to the field width; if the width is too small (such as X:10:7, X:5:4, or I:3), it is increased to fit the number. If the field is wider than is necessary, then the number is *right-justified*, that is, blanks are put in front of the number to fill out the remaining space. In the case of real numbers, rounding off is done when needed.

The second procedure, *Val*, converts from a string to a number (again, either *Real* or *Integer*). The string itself must contain exactly a number and nothing else; no characters other than digits, except for '+','-','.', and 'E' in the appropriate places. And, of course, the number in the string must be of the same type as the variable to which *Val* is converting it. Since there are so many chances for error, *Val* has a third parameter—a result value—

which tells you whether or not there were any problems. If the result is 0, then there were no problem during the conversion. If the result is greater than 0, then it indicates the character (*S*[*Result*]) at which it ran into problems. Here are a few examples:

If S holds the string:	then VAL(S) =	result code	
"14916"	14916	0	
" 32"	<undef>	1	{space}
"4281.953"	4281.953	0	
"-332.3"	-332.3	0	
"-332.3 "	<undef>	7	{space}
"4,281"	<undef>	2	{comma}

Before doing any heavy-duty numeric conversions, you would be well advised to read the *Formatted Output* section of Chapter 18, to get a good understanding of the concept of field widths.

A word of caution for those of you using 8-bit (CP/M) systems: do *not* use *Str* or *Val* within a function that is itself called within a *Write* or *WriteLn* statement. Strange and undesirable things will happen as a result. Instead, call the function beforehand, assigning its value to some variable, then use that variable in the Write/WriteLn statement. Those of you with 16-bit machines (CP/M-86, MS-DOS) needn't worry about any of this.

14.4 STRINGS AS PARAMETERS

You've probably noticed in these examples that whenever we pass a string to a procedure or function, we've defined that parameter as something like *BigStr* or *TokStr*, rather than **string**[255] or the like. For example, we used

```
procedure Parse(var Line,Word : BigStr);
```

instead of

```
procedure Parse(var Line,Word : string[255]);
```

In TURBO Pascal, you cannot directly declare a parameter as being a string of some given length. Instead, you must declare a data type that is equivalent to a string of some length, then use that data type in the parameter declaration, like this:

```
program ParseText;
type
  BigStr        =  string[255];
  .
  .
  .
procedure Parse(var Line,Word : BigStr);
```

You have to take one other factor into account. When you call a procedure or function that has string parameters, and those parameters are declared as **var**, that is, they can be changed by the procedure, then the string variables you pass must be declared to have the same length as the parameters. For example, suppose you added a procedure like this to our example above:

```
procedure DoParsing;
var
  TLine,TWord        : string[80];
begin
  Write('Enter line:    '); ReadLn(TLine);
  WriteLn('Parsed line: ');
  while Length(TLine) > 0 do begin
    Parse(TLine,TWord);
    WriteLn('<',TWord,'>')
  end
end; { of proc DoParsing }
```

If you tried to compile the program with this addition, you would get a type mismatch error when it got to the line *Parse(TLine, Tword);*. Why? Because *TLine* and *TWord* are declared to be of length 80, while *Parse* is expecting two strings of length 255. This, of course, can cause real problems if you're trying to write some general-purpose routines (such as *LowToUp*) to handle all different strings. The reason for the error is to prevent you from returning too long a string or indexing into "random" memory (that is, beyond the end of the string). You can, however, disable this error checking by putting a *compiler option* at the start of your program (more about these later in the book). All you have to do is to place the following comment somewhere before the call to *Parse*; you could, in fact, put it at the top of your file, like this:

```
{$V-}
program ParseText;
  .
  .
  .
```

TURBO Pascal will then no longer check to see if the string lengths match. Like most "disable" options, you should use this with caution; if you aren't careful with passing different length strings (that is, strings with different defined maximum lengths) to the same procedure, you could get some bizarre errors. One method would be to disable the checking for each specific call, such as

```
    {$V-}  { turn off string checking }
    Parse(TLine,Tword);
    {$V+}  { turn on string checking}
```

That way, you will be turning it off for only those places where you actually do not need it.

14.5 STRINGS, CHARACTERS, AND ARRAYS

TURBO Pascal makes it easy to mix references to strings, characters, and arrays. You can use strings and characters interchangeably; in other words, you can use a string almost anywhere you use a character, and vice versa. There are, however, a few exceptions. First, you can not use a character as a parameter to a procedure or function where a **var** parameter of some string type is expected. For example, if you have a variable *Ch* of type *Char*, you couldn't write

```
    UpToLow(Ch);
```

since *UpToLow* wants to pass a string back. The {$V–} option won't even work (and it's probably just as well). Second, if you assign a string to a character, the string must have a length of exactly 1; a longer or smaller (null) string will result in a run-time error.

In a similar fashion, you can use variables that are declared as **array of** *Char* as strings, again with a few restrictions. You can't assign string variables to arrays, nor use arrays as **var** string parameters. You can assign string constants to arrays if the constant is exactly the same length as the array.

Here is a program which demonstrates many of the ways in which you can mix strings, characters, and arrays of characters. Try to predict ahead of time what your output will look like; then key it in and run it, and see how close you were.

```
program Test;
type
   BigStr        =  string[255];
var
   St1,St2       :  string[40];
   Ch1,Ch2       :  Char;
   Value,Code    :  Integer;
   Ar1,Ar2       :  array [0..9] of Char;

procedure PutLen(St : BigStr);
begin
   WriteLn(St,':   ',Length(St))
end; { of proc PutLen }
```

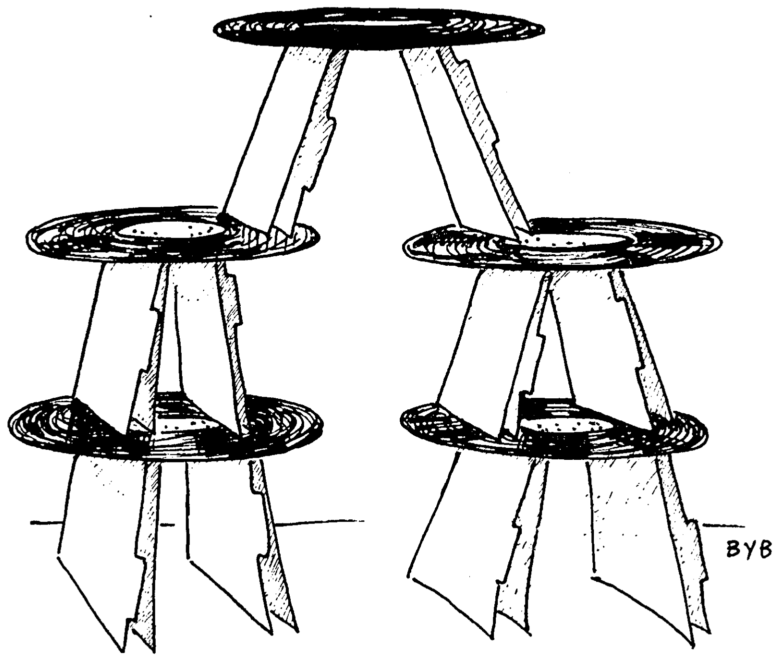
```
{ main body of program Test }
begin
  Ch1 := 'A'; Ch2 := 'Z';
  St1 := 'The alphabet goes from ' + Ch1 + ' to ' + Ch2;
  Ar1 := '0123456789';
  PutLen(Ar1);
  St2 := 'j';
  Ch1 := St2;
  Ch2 := '1';
  PutLen(St1);
  PutLen(St2);
  WriteLn('Ch1 in St2:    ',Pos(Ch1,St2));
  WriteLn('Ch2 in Ar1:    ',Pos(Ch2,Ar1));
  Val(Ch2,Value,Code);
  WriteLn(Ch2,' ',Value,' ',Code);
  St1 := Ch2;
  Ar2 := '0000000012';
  Val(Ar2,Value,Code);
  WriteLn(Ar2,' ',Value,' ',Code);
  St2 := Ar1;
  PutLen(St2)
end. { of program Test }
```

Now, here is the table I promised you earlier in the chapter. This sort of brings all of the string procedures and functions together into one location, so you can easily compare what they do.

STRING PROCEDURES AND FUNCTIONS

Concat(S1,S2,...,Sn)	Function which returns string composed of S1 through Sn concatenated together; the plus sign (+) can also be used
Copy(St,Index,Size)	Function which returns string composed of St[Index]..St[Index+Size-1]
Delete(St,Index,Size)	Procedure which deletes Size characters St starting at St[Index]
Insert(St1,St2,Index)	Procedure which inserts St1 into St2 starting at St2[Index]
Length(St)	Function which returns current length of St
Pos(Pat,St)	Function which returns position (index) of Pat within St
Str(Val,St)	Procedure which converts Val (Integer or Real) into a string and stores it in St
Val(St,Val,Index)	Procedure which converts St into Val (Integer or Real) and sets Index to the position of any error occurring (0 = none)

15. RECORDS



—Records and Files—

You've learned about *arrays*, which allow you to create collections of *objects* of the same type, *indexed* by some sort of *value*. But what if you want a collection of objects of *different types*? You could declare an array for each data type, but that could get tedious and complicated. Pascal offers another solution:*records*.

The **record** data structure is massively useful, if handled correctly. It allows you to glue together a lot of different data types into a single structure. The basic definition of a record is:

```
record
  <Ident1>   : <DataType1>;
  <Ident2>   : <DataType2>;
  .
  .
  .
end;
```

A record consists of a number of *fields* (*Ident1, Ident2*, and so on). Each identifier can be of any data type or structure, including integer, boolean, real, char, DST, array, record, set, or pointer. To refer to a field, you give the variable's name, a period, and then the field's name, such as:

```
    VarName.Ident1
```

In the section on arrays, a few attempts were made to define the information for sectors in the "Trek" game. Let's see if a new approach, using records, might not work best. Suppose you define the following:

```
program Trek;
const
  StarMax                  =  9;
  ShipMax                  =  15;
  XMax                     =  8;
  YMax                     =  10;
type
  XRange                   = 1..XMax;
  YRange                   = 1..YMax;
  ASector =
    record
      Ships                :  0..ShipMax;
      Stars                :  0..StarMax;
      Base                 :  Boolean
    end;
var
  Sector                   :  array[XRange,YRange] of ASector;
```

Note well that *Ships, Stars,* and *Base* are not elements of a declared scalar type, nor are they separate arrays; they are field identifiers for the record type *ASector*. Accordingly, you would refer to the number of stars in sector 3-2 as *Sector[3,2].Stars*. Note also that in setting up this record, you've restricted the possible values of the fields *Stars* and *Ships* to a very small subrange of integers, and you've changed Base into a boolean variable. You thus avoid two problems: (1) having your variables take on legal but nonsensical values (such as -34 stars), and (2) using too much memory.

You may remember that in the section on Arrays, you used two approaches: a 3-dimensional array of integers that used up 480 bytes, and three separate arrays that used a total of 240 bytes. This array of records also uses only 240 bytes and has the additional advantage of holding all of the sector information in one spot (rather than spread out over three arrays).

This record example is a fairly simple one:just three fields, each on its own line. Let's work on a more complicated example. Suppose that you wanted to model each of the stars in a sector as a solar system, complete with planets. You might make a fairly simple definition for the planets and systems:

```
type
   PlanetClass      =  (Terroid,Jovoid,Asteroid);
   StarClass        =  (O,B,A,F,G,K,M,R,N);
   StarSize         =  (SubDwarf,Dwarf,Normal,Giant, SuperGiant);
   Planet           =  record
      Dist,Angle,Radius,Mass,Gravity
                     :  Real;
      Class          :  PlanetClass
    end;
   System =
    record
      SX,SY          :  Byte;  { location w/in sector }
      Class          :  StarClass;
      Size           :  StarSize;
      Mass,Radius,Temp,Luminosity
                     :  Real
    end;
```

Not a whole lot of information, but enough to get read-outs on the "bridge status display." Notice that you can define several fields of the same type at the same time (*SX* and *SY*; *Dist*, etc.). Had you wanted to, you could have given each one a separate line; it isn't necessary, but it can help for documentation purposes.

15.1 THE "WITH" STATEMENT

Let's suppose you have the following definitions:

```
const
  XMax                 = 8;
  YMax                 = 10;
  ShipMax              = 15;
  StarMax              = 9;
type
  XRange               = 1..XMax;
  YRange               = 1..YMax;
  ASector =
   record
     Starbase          : Boolean;
     SbX,SbY           : Byte;
     Ships             : 0..ShipMax;
     Stars             : 0..StarMax
   end;

var
  Sector               : array[XRange,YRange] of ASector;
```

You've defined an array of records which you can now use in your program. If you wished to clear the entire sector of starbases and enemy ships, you might do this:

```
for X := 1 to XMax do
  for Y := 1 to YMax do begin
   Sector[X,Y].Starbase := False;
   Sector[X,Y].SbX := 0;
   Sector[X,Y].SbY := 0;
   Sector[X,Y].Ships := 0;
   Sector[X,Y].Stars := 0
  end;
```

As you can see, this could get very tedious, especially for more complicated records and/or actions. Luckily, Pascal gives you a shorthand means of referring to all those fields: the **with** statement. This takes the form

```
   with <record id> do <statement>;
```

Within the *statement*, you can refer to all of the fields of the *record id* without having to specify it each time. Instead, the base address of *record id* is calculated when the **with** statement is found, and that address is used to offset all of the fields. This makes it very handy for use in loops. For example, you could rewrite your initialization code as:

```
for X := 1 to XMax do
  for Y := 1 to YMax do
    with Sector[X,Y] do begin
      Starbase := False;
      SbX := 0;
      SbY := 0;
      Ships := 0;
      Stars := 0
    end;
```

Note that the **with** statement is inside of the two **for** statements. This is critical, for it ensures that *Sector*[*X, Y*] refers the desired element of the array. If you had written

```
with Sector[X,Y] do
  for X := 1 to XMax do
    for Y := 1 to YMax do begin
      .
      .
      .
    end;
```

then the base address would be calculated just once: before you entered the two loops, and using whatever the values of X and Y were before the loops started.

You can nest **with** statements, and you can also list more than one record identifier in a single **with** statement. For example, you could write the following:

```
type
  RecType1 =
    record
      Field1       :  Integer;
      Field2       :  Real
    end;
  RecType2 =
    record
      Field3       :  string[20];
      Field4       :  Boolean
    end;
var
  Rec1             :  RecType1;
  Rec2             :  RecType2;
begin
  with Rec1 do begin
    Field1         :=  32;
    Field2         :=  17.76;
    with Rec2 do begin
      Str(Field1,Field3);
      Field4       :=  (Field2 > 3.14159)
    end;
  Field1           :=  Length(Rec2.Field3)
  end
end.
```

or you could modify the main body to read

```
begin
  with Rec1,Rec2 do begin
    Field1         :=  32;
    Field2         :=  17.76;
    Str(Field1,Field3);
    Field4         :=  (Field 2 > 3.14159);
    Field1         :=  Length(Field3)
  end
end.
```

In the first example, you had to explicitly identify *Rec2* when assigning the first character of *Field4* to *Field3* of *Rec1*. In the second example, you could do a direct assignment.

There are a few pitfalls you much watch for in using the **with** statement. First, if you have more than one record defined in a given **with**, or if you have nested statements, make absolutely sure that the records don't have fields with the same name. Suppose you changed the definition of *RecType2* to read:

```
RecType2 =
  record
     Field1          :  string[20];
     Field2          :  Boolean
end;
```

Now *Rec1* and *Rec2* have fields with the same names. If you then write something like

```
with Rec1,Rec2 do begin
   WriteLn(Field1);
   WriteLn(Field2)
end;
```

which record's fields will be written out? The answer: *Rec2*, since it was the last defined in the list. Likewise, if you have nested **with** statements, the record last appearing has precedence over any others. In this example, problems are unlikely to occur since the data types of the identically-named fields are different; an attempt to assign a value will either go to the correct field or will result in a compiler error. But what if both records had *Field1* defined as type *Integer*? In that case, you might unknowingly assign a value to the wrong field. The moral:*avoid identically-named fields unless you're very sure of what you're doing*.

A similar problem can occur when you create a field that has the same name as a variable:

```
program BadExample;
type
   RecType =
     record
       Name              :  string[25]
       Age               :  Integer
     end;
```

```
var
   Rec              :  RecType;
   Name             :  string[25];
begin
   Name := 'J. Michael Browning';
   with Rec do begin
     Name := 'Bob Trammel';
     WriteLn(Name)
   end;
   WriteLn(Name)
end.
```

Can you guess what the output of this program is? If you guessed

```
     Bob Trammel
     J. Michael Browning
```

you're absolutely correct. Again, much like the rules of scope you learned about in Chapter 11, the last declaration takes precedence. In this case, the **with** statement acts as a declaration. Again: *be careful with identical identifiers.*

15.2 VARIANT RECORDS

Occasionally, you may define a record with redundant or mutually exclusive fields. For example, consider the following record definition:

```
type
  .
  .
  .
  Government           =  (Federation,Klingon,Romulan,Trader);
  AShip =
   record
     SX                :  XRange;
     SY                :  YRange;
     QX,QY             :  Byte;
     Energy            :  0..1023;
     Source            :  Government;
     Phaser,Disruptor,Beam1,Beam2
                       :  Byte;
     Photon            :  0..15;
     Torpedo           :  0..7;
     Plasma            :  0..9;
     Cloaked           :  boolean
   end;
```

Let's suppose that Federation ships have Phaser and Photon weapons; Klingons have Disruptors and Torpedos; Romulans have Beam and Plasma weapons, along with Cloaking devices; and Traders have no weapons at all. That means that each and every ship record has four unused fields and, therefore, wasted space. If you only have a few records, the space won't matter much; if you have a lot, it can become significant. Yet, the records are so similar that it seems silly to define three different records. Furthermore,if you want to put all the ships in a single array or file, you have to stay with one record. What do you do?

Well, Pascal allows you to create a record with conditional fields, called a *variant record*. To create a variant record, you define a *tag field*, which is just a field of some scalar type (often a declared scalar type). This field goes after all the *fixed* (non-varying) fields and is used as the argument in a **case** statement. For each possible value of that field, you define the desired exclusive fields. For example, you might rewrite your definition of *AShip* as follows:

```
type
  .
  .
  .
  Government          = (Federation,Klingon,Romulan,Trader);
  AShip =
   record
     SX               :  XRange;
     SY               :  YRange;
     QX,QY            :  Byte;
     Energy           :  0..1023;
     case Source      :  Government of
       Federation     :  (Phaser        :  Byte;
                      :  Photon         :  0..15);
       Klingon        :  (Disruptor     :  Byte;
                      Torpedo           :  0..7);
       Romulan        :  (Beam1,Beam2   :  Byte;
                      Plasma            :  0..9;
                      Cloaked           :  Boolean);
       Trader         :  ( )
  end;
```

For each value of Source, you define the fields that exist for that case, enclosed within parentheses. Note that even if there are no fields at all, you still need to put in the parentheses.

You may wonder what good this all is. First, it cuts down on the size of the record. Why? Because the fields for each variant occupy the same space. In other words, the fields *Phaser*, *Disruptor*, and *Beam1* all reside at the same location in memory, rather than each having their own separate slot. The result is less memory required for each record of that type. The original definition required 15 bytes per record; this variant record takes up only 11 bytes. That may not seem like a whole lot, but consider this example:

```
type
  .
  .
  .
  Government          =  Federation,Kling,Romulan, Trader);
  Sides               =  (Forward,FPort,FStarboard,APort,
                         AStarboard,Aft);
  Weapons             =  array[Sides] of Byte;
  AShip =
    record
      SX              :  XRange;
      SY              :  YRange;
      QX,QY           :  Byte;
      Energy          :  0..1023;
      case Source     :  Government of
        Federation    :  (Phaser        :  Weapons;
                        Photon          :  array[Sides] of 0..15);
        Klingon       :  (Disruptor     :  Weapons;
                        Torpedo         :  array[Sides] of 0..7);
        Romulan       :  (Beam1,Beam2   :  Weapons;
                        Plasma          :  array[Sides] of 0..9;
                        Cloaked         :  Boolean);
        Trader        :  ()
    end;
```

You've replace a single byte value with an array of six bytes in several places. A variable of this type now takes up 26 bytes. However, if you declared this as a non-variant record, it would require 50 bytes, or nearly twice as much room. If you had an array or file of these records, the difference could be very important, indeed.

There is a second, more esoteric aspect of variant records. Since certain fields now occupy the same space, you can set the value of one field and have it transfer over to another. As mentioned above, the fields *Phaser*, *disruptor*, and *Beam1* all occupy the same space. Because of that, we could do the following:

```
var
  ThisShip      :  AShip;
begin
  with ThisShip do begin
    Phaser[1] := 72;
    WriteLn(Disruptor[1])
  end
end.
```

which would write out "72". What good is this capability? Well, it allows for certain advanced programming tricks. For example, assuming an 8-bit system, you could define something like this:

```
type
  BigStr        = string[255];
  StringPointer=
    record
      case Flag :  Boolean of
        True    :  (Addr : Integer);
        False   :  (Ptr: ^BigStr)
    end;
var
  VStr          :  StringPointer;
begin
  VStr.Addr     :=   $B000;
  VStr.Ptr^ := 'This string is being written at $B000'
end.
```

Now, you can write a string at any given location in memory. Of course, since you don't always know what is sitting at a given memory location, this can be dangerous. That's why it's for advanced programming efforts.

Here's one more variation on variant records. Sometimes you want the variations, but you don't really care about the tag field. The *StringPointer* above is a good example. You're interested in *Addr* and *Ptr*, but you don't need or want *Flag*. In those cases, Pascal allows you to dispense with the tag field altogether. For example, you could rewrite *StringPointer* as:

```
type
  BigStr        = string[255];
  StringPointer =
    record
      case Boolean of
        True        : (Addr        : Integer);
        False       : (Ptr         : ^BigStr)
      end;
```

Note that you no longer have the tag field, *Flag*; you just have the tag field type, *Boolean*. This is known as a *free union* (as opposed to a *discriminated union*, one with a tag field).

So much for records. Now, let's move on to sets...

16. SETS

There are times when you want to test a scalar variable (Integer, Byte, Char, Boolean, DST) to see if its current value belongs to a set or collection of values. For example, suppose you want to write a subroutine that will write out a prompt for the user and then accept and return only a character belonging to an allowable set of characters. Using what you know now, how would you do that?

If you're really clever, you might come up with something like this:

```
type
  CharSet                    :  array[Char] of Boolean;
var
  OKSet                      :  CharSet;
```

You could then set each location in *OKSet* to the appropriate value (*True* or *False*) and use that to check the characters read in. Unfortunately, this approach tends to eat up lots of memory (128 bytes per *CharSet*) and program space. Fortunately, it's not necessary. Pascal handles sets for you by letting you define **sets**.

A set can be defined for any scalar data type, though there is one restriction: the numeric (ordinal) values of the data type (or subrange of a data type) used must range between 0 and 255. For example, you could not define a **set of** Integer, but you could define a **set of** Byte, or even a **set of** 21..47. This restriction also means that you can never have more than 256 items in a set.

A set constant is enclosed by brackets ("[" and "]"), with the elements of the set defined within. You can list each element separately; however, that could get tedious for sets with lots of elements (remember, you can have up to 256!). If a number of the elements are in contiguous (numerical or ordinal) order, you can use subrange notation as a type of shorthand. Here are some examples of sets:

```
  []                         { empty set—contains nothing }
  [1,3,5,7,9]                { set of Byte }
  ['A'..'Z']                 { set of Char }
  [Mon,Tue,Thur]             { set of Days }
  [Jan..Jun,Aug,Oct..Dec]    { set of Months }
  [Loknar,Chandley]          { set of ShipType }
```

As mentioned, a set can hold up to 256 different elements, all of the same data type. It stores the presence of each element as a single bit; as a result, a set can be at most 32 (256/8) bytes and is not always that large.

The usefulness of sets is not always obvious. In fact, it can be hard at times to think of anything to do with them. However, here's an example that may show one very good use for sets:

```
program CharTest;
{ $V-}                       { to avoid any problems passing strings }
type
  CharSet                    = set of Char;
  Prompt                     = string[80];
var
  Cmd                        : Char;

procedure GetChar(var Ch : Char; Msg : Prompt;
                      OKSet : CharSet);
begin
  repeat
    Write(Msg); ReadLn(Ch);
    Ch := (UpCase(Ch);      { force to be upper case }
     until Ch in OKSet
end; { of proc GetChar }
{  main body of CharTest  }
begin
  repeat
    GetChar(Cmd,'CharTest>   S)peak, C)ount, Q)uit:   ',
              ['S','C','Q']);
      case Cmd of
      'S'  : WriteLn('Woof! Woof!');
      'C'  : WriteLn('1, 2, 3, 4, 5, 6, 7, 8, 9, 10')
    end
  until Cmd = 'Q'
end. { of prog CharTest }
```

The procedure *GetChar* is extremely useful for input processing. It prompts the user with a message, then accepts a single character response. It converts the character to uppercase, then checks to see if it's a valid command. If not, it continues to prompt until a correct selection is entered. To see how and why this works, you need to understand about set comparisons.

16.1 SET COMPARISONS

The key of *GetChar* is the set comparison **in**, which takes the form:

```
<element> in <set>
```

The term *element* must be an expression of the same type as *set*'s base type. If *set* is a **set of** *Char*, then *element* must resolve to a single character. This expression returns *True* if and only if *element* is currently in *set*; otherwise, it returns *False*. In this example, *Ch in OKSet* returns *True* if the value of *Ch* is in the set *OKSet*. Here, *Ch* must be *S*, *C*, or *Q* in order for this expression to be *True*. Note that *GetChar* converts any letters received to upper case, so that this expression will be *True* if the user types *s*, *c*, or *q*, as well. Without this conversion, those letters would be ignored, since lower case letters are different from upper case letters.

Besides testing for membership in a set, you can also make comparisons between sets themselves. Two comparisons are obvious: equality (=) and inequality (< >). The other two, "<=" and ">=", are for **set inclusion**. The expressions

```
Set1 <= Set2
Set2 >= Set1
```

will return *True* if all the elements in *Set1* are also in *Set2*. Suppose you have the following:

```
type
  ShipType                  = (Constitution,Enterprise,Reliant,Loknar,
                            Larson,Chandley,Excelsior,Baker);
  ShipSet                   = set of ShipType;
var
  AllShips,Frigates,Destroyers,Cruisers,Battleship,Temp
                            : ShipSet;
begin
  AllShips                  := [Constitution..Baker];
  Frigates                  := [Loknar,Chandley];
  Destroyers                := [Larson,Baker];
  Cruisers                  := [Constitution..Reliant,Excelsior];
  Battleship                := [Excelsior];
  Temp                      := [Constitution..Baker];
  .
  .
  .
end.
```

Given this, the following expressions have the values shown:

AllShips = Temp	True
AllShips = Cruisers	False
Frigates < > Destroyers	True
BattleShip <= Cruisers	True
Temp >= AllShips	True
Destroyers <= AllShips	True
Cruisers <= Battleship	False
[ ] <= <any set>	True

16.2 SET OPERATIONS

As has been mentioned, sets represent collections of values of a scalar data type (Integer, Byte, Boolean, Char, or a declared scalar type). You've seen how to compare sets; you can also perform several operations on them. Let's assume the following definitions:

```
type
  ShipType                  = (Constitution,Enterprise,Reliant,Loknar,
                            Larson,Chandley,Excelsior,Baker);
  ShipSet                   = set of ShipType;
var
  Set1,Set2,Set3,AllSets,Empty
                                       : ShipSet;
```

```
begin
  AllSets              := [Constitution..Baker];
  Set1                 := [Constitution..Reliant,Excelsior];
  Set2                 := [Loknar,Larson,Chandley,Baker];
  Set3                 := [Enterprise,Loknar,Baker];
  Empty                := [];
  .
  .
  .
end.
```

The first operation to look at is *set intersection*, which uses the multiplication symbol (*). The intersection of two sets is a set containing all the elements common to both sets. For example, here are some intersections, along with the resulting sets:

```
Set1 * Set2             []
Set2 * Set3             [Loknar,Baker]
Set3 * AllShips         [Enterprise,Loknar,Baker]
AllShips * Empty        []
```

In each case, the result is a set containing the elements that appear in both sets. In two cases (the first and the last), there were no elements in common, so the result was the empty set.

The second operation is *set union*, which uses the plus sign (+). The union of two sets is the set containing all the elements in both sets. Here are some examples:

```
Set1  + Set2            [Enterprise..Baker] ( = AllShips)
Set2  + Set3            [Enterprise,Loknar,Larson,Chandley,Baker]
Set3  + Set1            [Constitution..Loknar,Excelsior,Baker]
AllShips + Empty        [Enterprise..Baker]
```

The third and last operation is *set difference*, which uses the minus sign (-). The difference of two sets is the set containing all of the elements in the first set that are not in the second set. Note well that, unlike set intersection and set union, the set difference operation is *not* commutative; that is, *Set1 -Set2* is not the same as *Set2 -Set1*. Here are some examples:

AllShips - Set1	[Loknar..Chandley,Baker] (= Set2)
AllShips - Set2	[Constitution..Reliant,Excelsior] (= Set1)
Set1 - AllShips	[]
Set1 - Set2	[Constitution..Reliant,Excelsior]
Set1 - Set3	[Constitution,Reliant,Excelsior]
Set3 - Set1	[Loknar,Baker]
Set3 - Set2	[Enterprise]
Set2 - Set3	[Larson,Chandley]

Note that if *SetA* - *SetB* = [ ] (the empty set), then *SetA* and *SetB* are mutually exclusive, that is, they have no elements in common.

17. POINTERS AND DYNAMIC ALLOCATION

—Pointers—

From time to time, you will find yourself wanting to be able to create and destroy data structures while a program is actually executing. Let's recall the *Ship* data type from the chapter on records:

```
const
   XMax             =  8;
   YMax             =  10;
type
   XRange           =  1..XMax;
   YRange           =  1..YMax;
   Government       =  (Federation,Klingon,Romulan,Trader);
   Sides            =  (Forward,FPort,FStarboard,APort,
                       AStarboard,Aft);
   Weapons          =  array[Sides] of Byte;
```

```
  AShip =
   record
     SX                   : XRange;
     SY                   : YRange;
   QX,QY                  : Byte;
     Energy               : 0..1023;
     case Source          : Government of
       Federation         : (Phaser          : Weapons;
                             Photon          : array[Sides] of 0..15);
       Klingon            : (Disruptor       : Weapons;
                             Torpedo         : array[Sides] of 0..7);
       Romulan            : (Beam1,Beam2     : Weapons;
                             Plasma          : array[Sides] of 0..9;
                             Cloaked         : Boolean);
       Trader             : ( )
    end;
```

Now, let's suppose that the number of ships in existence during the course of a game varies widely. What kind of data structure would you use to be sure you could handle all (reasonable) cases?

One possible solution, using arrays, might look like this:

```
var
  Ships                 : array[1..MaxShips] of AShip;
  ShipCount             : 0..MaxShips;
```

where *MaxShips* is a **const** value representing the maximum number of ships allowed during the game. The variable *Ship-Count* would be initially set to 0. Each time a new ship was created, the next free slot in *Ships* would be used:

```
if ShipCount < MaxShips then begin
  ShipCount := ShipCount + 1;
  CreateShip(Ships[ShipCount])
end
else WriteLn('No more space for ships');
```

The procedure *CreateShip* is one that you would write. It would set all of the fields of *Ships*[*ShipCount*] to the appropriate initial values. When a ship was destroyed, say *Ships*[*Indx*], then you would "shuffle down" all the ships from *Indx+1* to *ShipCount* like this:

```
for Jndx := Indx+1 to ShipCount do
  Ships[Jndx-1] := Ships[Jndx];
ShipCount := ShipCount - 1;
```

This method is fine, but has two drawbacks. First, you (the programmer) must decide ahead of time the maximum number of ships possible (*MaxShips*). The program will never be able to handle any more ships than that. Second, space (memory) will always be allocated for the maximum number of ships, regardless of how many are actually in use. If you define *MaxShips* to be 100, then you can never have more than 100 ships, and space for 100 ships will always be set aside even if you only have one or two. These are common limitations in most languages, but Pascal does offer a way around them: *pointers*.

17.1 POINTERS

Suppose you modify the definition of *Ships* as follows:

```
var
  Ships                    : array[1..MaxShips] of ^AShip;
  ShipCount                : 0..MaxShips;
```

If you'll look closely, you'll see that *Ships* is no longer an **array of** *AShip* but of ^AShip. The notation ^ on a data type refers to a **pointer** to the data type. In other words, *Ships* is no longer 100 records (or whatever *MaxShips* is), but 100 pointers to records. "What's the difference?", you may ask. Simple. The record is some collection of data. The pointer is an address. If you still don't see the distinction, consider this: what's the difference between 100 people and the phone numbers of 100 people? With either one, you can talk to 100 people, but you need an auditorium for the former; for the latter, you just need a phone and sufficient room for the people you choose to summon. The analogy is a little rough, but you should get the idea.

The next question is, how do you use a pointer? Or, better put, how do you use the record that the pointer points to? Answer:by pointing to it. If *Ships* is an **array of** ^AShip (that is, pointers to records of type *AShip*), then *Ships*[*1*] is a pointer—that is, an address—and *Ships[1]^* is the record to which it is pointing. In other words, by sticking a carat (^) at the end of the pointer variable, you now refer to the data structure to which it points. For example, if you wanted to set the energy level of all currently allocated ships to 1023, you would do the following:

```
for Indx :=Number to ShipCount do
   Ships[Indx]^.Energy := 1023;
```

Note well that you cannot write *Ships[Indx].Energy* := 1023, since *Ships[Indx]* is a pointer, not a record of type *AShip*. The carat makes a big difference.

The third question is, how do you point the pointer? In other words, how do you assign an address to the pointer, and how do you ensure that the address the pointer contains represents an area of memory that is not being used by anything else (operating system, program, other pointers, etc.)? Answer: Pascal does it for you via the predeclared procedure *New*. For example, your ship initialization routine could look like this:

```
if ShipCount < MaxShips then begin
   ShipCount := ShipCount + 1;
   New(Ships[ShipCount]);
   CreateShip(Ships[ShipCount]^)
end
else WriteLn('Maximum number of ships allocated');
```

Note that the procedure *New* takes a pointer as its argument. It allocates the necessary amount of memory and sets the pointer to the appropriate address. Also, note that there is a special pointer value called nil which is used to indicated pointing at nothing. Its special value is usable on all pointer types.

17.2 THE HEAP

The procedure *New* creates a new copy of the appropriate type of data structure. For example, the statement

```
   New(Ships[ShipCount]);
```

creates a new record of type *AShip*, the address of which *Ships*[*ShipCount*] now contains. You may recall that a variable of type *AShip* takes up 26 bytes. The question is, just where are those 26 bytes? They're not in the array *Ships*, which just contains pointers. Instead, they have been allocated in an area of RAM known as the **heap**. Loosely put, the heap consists of whatever memory is left over after allocating space for the operating system (CP/M, CP/M-86, MS-DOS), TURBO Pascal (if loaded), the run-time library, your program, and any variables declared within the program (such as *Ships*).

If you're using an 8-bit system (with a maximum of 64K), have TURBO Pascal loaded, and are running a large program, then the heap may be very small; possibly as little as 1K, enough to hold about 40 *AShip* records. On the other hand, if you're using a 16-bit system with lots of RAM, then the heap can be very big. How big? A program running under MS-DOS 2.0 on a 512K system has over 430K of memory in the heap, or enough space for about 17,000 *AShip* records. Because of those variations, and because of the different operating systems involved, it's hard to discuss specifics of how the heap works; that information is best gleaned from the OS-specific appendices in the **TURBO Pascal Reference Manual**.

Having read all that, you're probably still wondering how the heap is used. When your program starts running, a special variable (*HeapPtr*) points to the start (or bottom) of the heap. When you call *New(Ptr)*, then *Ptr* gets the value of *HeapPtr*, and *HeapPtr* is increased by the size of the data structure that *Ptr* points to. In other words, the sequence is something like this:

```
    Ptr := HeapPtr;      HeapPtr := HeapPtr + SizeOf(Ptr^);
```

Repeated calls to *New* causes *HeapPtr* to "grow" upwards, reducing the space remaining on the heap. You can find out at any time just how much free space is left on the heap via the standard function *MemAvail*. For 8-bit systems, *MemAvail* returns the number of bytes left on the heap; for 16-bit systems, the number of **paragraphs** (16-byte chunks). *MemAvail* returns an Integer value, which means that it will be negative if there are over 32K bytes/paragraphs free. You can convert it to a positive number with the following code:

```
var
  TrueFree        : Real;
begin
  TrueFree := MemAvail;        { convert to real value }
  if TrueFree < 0.0
    then TrueFree := TrueFree  + 65536.0;
  WriteLn('Space available:    ',TrueFree:7:0)
end.
```

17.3 LINKED LISTS

You've partially solved your problem—you no longer have to allocate space for the maximum number of ships—but you still have to decide ahead of time what that maximum number is. And since that means that you always have to allocate space for that many pointers, you may not have gained that much of an advantage. Is there no way around this problem?

Obviously there is, or else you wouldn't be reading this. The answer lies in creating what is known as a **linked list**. At this point, there is a temptation to refer you to Chapter 2 in the book **Fundamental Algorithms** (2nd ed.) by Donald E. Knuth (Addison-Wesley; Reading, Massachusetts; 1973) and leave it at that. However, having brought the subject up, there results a certain obligation to tell you *something* about it. Here goes.

Let's modify the definition of the **record** *AShip* as follows:

```
type
.
.
.
AShipPtr              =  ^AShip;
AShip =
  record
    SX                :  XRange;
    SY                :  YRange;
    QX,QY             :  Byte;
    Energy            :  0..1023;
    Shields,Beams     :  Byte;
    Next              :  AShipPtr
  end;
```

You may realize that you just have violated the Great Underlying Rule of Pascal, namely that no identifier can be referenced until it has been declared. As you can see, you defined *AShipPtr* to be a pointer to type *AShip* **before** you defined *AShip*. This ability is a necessary exception, since without it linked lists would be impossible (or, at least, very messy).

Having modified your data structure, you now define your variables as follows:

```
var
  First,Last          :  AShipPtr
  ShipCount           :  Integer;
```

At the start of the game, you would initialize your variables as follows:

```
    First := nil;
    Last:= nil;
    ShipCount := 0;
```

(The predefined identifier **nil** represents the value a pointer has when it's not pointing at anything.)When you wish to create a new ship, you call your procedure *AddShip*:

```
procedure AddShip;
begin
   if First = nil then begin
     New(First);
     Last := First
   end
   else begin
     New(Last^.Next);
     Last := Last^.Next
   end;
   Last^.Next := nil;
   ShipCount := ShipCount + 1;
   CreateShip(Last^)
end; { of proc AddShip }
```

(Note that you have a special test for creating a ship when none exist.) The pointer *First* always points to the first ship in the list. Each ship then points to the next one in the list via the field *Next*. The pointer *Last* always points to the last ship in the list. If there is only one ship in the list, then *First* and *Last* both point to it. If there are no ships at all in the list (the list is **empty**), then both *First* and *Last* equal **nil**.

Now that you can create the list, how do you reference a particular ship in it? With the array it was easy: all you had to do was index into the array and hey, presto! there it was. With the linked list, you have to look for it. Here's a procedure to do just that. It takes the index value and returns a pointer to the appropriate ship (if it exists):

```
procedure FetchShip(Index : Integer;
                  var Ptr : AShipPtr);
begin
   Ptr := First;         { point at first ship in list }
   while (Index > 0) and (Ptr <> nil) do begin
     Index := Index - 1;
     Ptr := Ptr^.Next
   end
end; { of proc FetchShip }
```

If *Index* is less than or equal to 0, then *Ptr* points to the first ship in the list. If *Index* is greater than the number of ships in the list, then *Ptr* gets set to **nil**. Otherwise, *Ptr* points to the appropriate ship.

17.4 DEALLOCATION AND MEMORY MANAGEMENT

At this point, you need to know about how to reclaim the memory used by a pointer when you no longer need the data structure it points at. TURBO Pascal provides two approaches to memory management. One method is to use the predefined procedure *Dispose* to reclaim the memory pointed to by a given pointer. Going back to your array of pointers (*Ships*), you could delete a particular ship this way:

```
if Ships[Indx] <> nil
  then Dispose(Ships[Indx]);
```

This would set *Ships[Indx]* equal to **nil** and make the memory that it had used available for other pointers. If you just wrote

```
    Ships[Indx] := nil;
```

then the memory would not be reclaimed because the pointer would simply be pointing at nil.

Even when memory is reclaimed (via *Dispose*), there arise the problems of **memory management**, **fragmentation**, and **garbage collection** (honest, that's what it's called!). Without going into all of the gory details (again, see Knuth), here's a brief, simple description of the problem.

Think of the heap (the memory used for dynamic allocation) as a long, narrow shelf. Each time you create a variable using *New*, you place a wooden block (width = amount of memory needed) somewhere on the shelf where it will fit. Some decision has to be made about where to place it. That's memory management. Each time you destroy a variable using *Dispose*, you remove the corresponding block from the shelf. After repeated calls to *New* and *Dispose*, you can find that the free space on the shelf is tied up in lots of small, often useless chunks between blocks. That's fragmentation. To reclaim all of those little chunks, you shove all of the wooden blocks together and move them to one end of the shelf. Now, all of the free space is in one large chunk. That's garbage collection. It all sounds easy, but in practice it can be a real headache.

TURBO Pascal supports *New, Dispose*, and *MaxAvail* (which returns the size of the largest free block of memory on the heap), but it does not have a garbage collection facility. If, however, all the data structures you create and destroy are the same type, or even just the same size, then fragmentation and garbage collection are not problems. Why? Because any gap that appears because of a given item being disposed will be exactly the right size for any new item being created. In other words, if all the wooden blocks on the shelf are the same size, then you'll never have a problem adding and deleting them (except, of course, when you run out of space altogether).

If you foresee problems using Dispose, you might consider TURBO's alternate memory management scheme, a very simple one. When you create variables using New, the blocks are placed next to each other, starting at the left end of the shelf (low memory). At any point, you can use the procedure Mark to get the address where memory for the next variable would be allocated. You can then continue to create variables. Later, you can call the procedure Release with the value you received from Mark. This effectively clears everything off the shelf that's to the right (above) the address passed to Release. Variables now created will be allocated starting at the address read when you called Mark.

There are some important things to note here. Let's suppose that you executed the following code:

```
First := nil;
ShipCount := 0;
Mark(TempPtr);
for Indx := 1 to 10 do
   AddShip;
Release(TempPtr);
```

You have created 10 ships, then reset your next-free-location pointer (called the **heap pointer**) to where it was before those ten ships were created. This has done nothing to affect the linked list of ships. All the information is still there, all of the pointers are still correct. The call to *Release* has no effect until the next time you call *New*. At that time, the newly-created data structure would start to overwrite the linked list. If you then read the linked list, funny things could result. The moral? If you use *Mark* and *Release*, be sure to set to **nil** any pointers pointing into the freed-up memory.

18. FILES

Throughout this book, you've probably noticed the heavy emphasis Pascal places on data structures: scalar variables, declared scalar types (DSTs), records, arrays, sets, and the like. It is no coincidence that when Niklaus Wirth, the father of Pascal, wrote a book on software development, he entitled it **Algorithms = Data Structures + Programs**. You've seen all kinds of data structures in the examples in this book, and you've been shown just how powerful they can be.

But, as always, a new problem has cropped up. Sure, you can define packed records with variant tag fields, or you can create arrays of sets of DSTs. But what happens when the program ends, as it sooner or later must? All your data structures quietly vanish away, and the information they hold is lost. Of course, you often don't care; when the program is done, you're done, too, and you no longer need the information. There are times, however, when you would like to save that data for the next time you run the program. For example, if you are playing a game, you may want to (or have to) stop for a while (hours, days, weeks). It could be disastrous to have to start over again from scratch.

There is, of course, a solution to this problem: **files**. A file is a collection of data structures, all of the same type. It is similar to an array, but with one big difference: most of the information is out on a disk. If that doesn't tell you much, don't worry. The following example should help clear things up:

```
const
  ShipMax            =  40;
  XMax               =   8;
  YMax               =  10;
type
  XRange             =  1..XMax;
  YRange             =  1..YMax;
  AShip              =
    record
      SX             :  XRange;
      SY             :  YRange;
      QX,QY          :  Byte;
      Energy         :  0..1023;
      Shields,Beams  :  0..255
    end;
var
  ShipFile           :  file of AShip;
  Ship               :  array[1..ShipMax] of AShip;
  ShipCnt            :  0..ShipMax;
```

The array *Ship* is a collection of 40 records of type *AShip*. All of those records are always in memory, and any one of them can be referred to using the notation *Ship[Indx]*, where *Indx* is an integer variable or expression resolving to a value from 1 to 40 (*ShipMax*). All the information in *Ship* is lost when the program is done.

The file *ShipFile* manages a collection of some number of records of type *AShip*. All of those records are kept out on mass storage (floppy disk, hard disk, RAM disk, whatever). *ShipFile* represent a "door" through which you can get or put individual records. However, it does it in a manner somewhat different from Standard Pascal. Standard Pascal uses the file variable (*ShipFile*) as a pointer; you then use the standard procedures *Put* and *Get* to copy records to and from the file. TURBO Pascal simply uses the Read and Write statements, along the same lines as textfile I/O.

The best way, of course, to show you what this all means is to give an example. Let's suppose that you've been playing your game for a while, and you want to quit, but you want to be able to resume your game later. Among other things, you need to save on the disk the information in *Ship*. The variable *ShipCnt* indicates how many of the records in *Ship* you are actually using. You then could write those records out to disk as follows:

```
Assign(ShipFile,'SHIPS.DAT');        { assign to a file on disk }
Rewrite(ShipFile);                   { open the file for writing}
for Indx := 1 to ShipCnt do
   Write(ShipFile,Ship[Indx]);       { put the next record out}
Close(ShipFile);                     { close and save the file}
```

The *Assign* statement connects the file variable *ShipFile* to the name "SHIPS.DAT". No disk file has actually been accessed at this point. The *Rewrite* statement, however, does cause some action. A file named SHIPS.DAT is now created out on the disk; if a file by that name already exists, it is erased. The file variable points at the first (0th) record in the file; the file itself is empty. Each call to *Write* copies the record contained in *Ship[Indx]* out to the disk file and advances to pointer to the next position. *Close* makes sure that any records being held in memory by the operating system are actually written out to the disk. and closes out the file, breaking the connection between *ShipFile* and "SHIPS.DAT".

You've saved the ship information out to disk and have stopped playing. Now, after a while, you want to start the game up again where you left off. Among other things, you need to read the records you saved back into *Ship*. One problem faces you, though: how does the program know how many records there are in the file? Answer: it doesn't need to. Instead, Pascal provides the boolean function EOF. EOF stands for **end of file**. As you read the file, EOF returns a *False* value until you hit the end; then it returns *True*. You can now read in the records as follows:

```
Assign(ShipFile,'SHIPS.DAT');        { assign to a disk file}
Reset(ShipFile);                     { open the file for reading }
ShipCnt := 0;                        { # of ships in array}
while not EOF(ShipFile) do begin
   ShipCnt := ShipCnt + 1;           { yes, there is another ship}
   Read(ShipFile,Ship[ShipCnt]);
end;
Close(ShipFile);                     { close file}
```

The *Assign* statement works as before. This time, though, you use *Reset* to start reading from the file. If SHIPS.DAT doesn't exist, you'll get an I/O error (see the section on Error Handling later on); otherwise, the file variable will point at the first (0th) record. Each call to *Read* copies the record currently pointed at into *Ship[ShipCnt]* and then points at the next record. If there is no next record, then the function EOF starts returning *True*, and you drop out of the end of the loop. Since you didn't know ahead of time how many records there were, you had to count them (by incrementing *ShipCnt*) until you reached the end of the file.

Actually, with TURBO Pascal, you could have found out ahead of time just how many records there were (though it really wouldn't matter). The function *FileSize* will return the number of records in a file, so you could have written:

```
Assign(ShipFile,'SHIPS.DAT');
Reset(ShipFile);
ShipCnt := FileSize(ShipFile);
for Indx := 1 to ShipCnt do
  Read(ShipFile,Ship[Indx]);
Close(ShipFile);
```

Both approaches work equally well; the second one has the advantage of letting you know ahead of time just how many records you have to read in (in case that's important).

In this example, *ShipFile* was a file of records of type *AShip*. You can define a file of anything (except files), so you could have done this instead:

```
type
  .
  .
  .
  AShip =
    record
      SX                   : XRange;
      SY                   : YRange;
      QX,QY                : Byte;
      Energy               : 0..1023;
      Beams,Shields        : 0..255
    end;
  ShipArray                =  array[1..ShipMax] of AShip;
var
  Ship                     : ShipArray;
  ShipFile                 : file of ShipArray;
  ShipCnt                  : 0..ShipMax;
```

Now, to save the information in *Ship*, you could write:

```
Assign(ShipFile,'SHIPS.DAT');
Rewrite(ShipFile);
Write(ShipFile,Ship);
Close(ShipFile);
```

Simple, isn't it? With a single Write statement, you have sent all the ship records out to the disk. To read it back in, you just write:

```
Assign(ShipFile,'SHIPS.DAT');
Reset(ShipFile);
Read(ShipFile,Ship);
Close(ShipFile);
```

Unfortunately, you've now lost track of one piece of information, namely the value of *ShipCnt*. However, there are a number of ways around this. First, you could redefine *ShipArray* to be a record with two fields: the ship count and the array itself. Second, you could save the value of *ShipCnt* in another file, either by itself or as part of another data structure. Third, you could add another field to the definition of *AShip* which would indicate whether or not that record was, indeed, a "real" ship; then, after you've read *Ship* in, you just go through and count all of the records with that value set. Finally, you could rewrite the game so that the value *ShipCnt* is no longer needed.

18.1 RANDOM ACCESS OF FILES

From what you've seen so far, to read or write a particular record in a file, you must go to the start of the file and work from there. If you wish to read the 10th record in a file, you must reset the file and go through the first nine records to get to the tenth. If you want to modify the tenth record, you must read it in as mentioned, reset the file, read in the first nine, then write the 10th back out. In short, a lot of work.

Now let's suppose that you want to sort that file in some way. Your sorting program would have to do a lot of reading from and writing to the file, using the clumsy procedure above. If the file is small enough, you can read the entire thing into memory, sort it there, then write it back out. If it isn't, you're in for a long, tedious program.

Your task would be easier if you could read and write any given record in the file without having to go back to the start and read all the preceding records. Such a capability is known as **random access** of files, as opposed to the sequential access described above. Since this is such a desirable trait, TURBO Pascal offers it to you via the *Seek* statement, taking the form

```
    Seek(FileVar,RecNum);
```

where *FileVar* is the file variable and *RecNum* is an integer variable or expression indicating the record number. After executing such a statement, *Read(FileVar,Rec)* will copy record *#RecNum* into *Rec*, while *Write(FileVar,Rec)* will write the data in*Rec* out to record number *#RecNum* in the file. Since (as mentioned above) the first record is record #0, the statement

```
    Seek(FileVar,0);
```

will point to the first record in the file. By the same token, the statement

```
    Seek(FileVar,FileSize(FileVar)-1);
```

will point at the last record in the file. *Seek* can, in fact, be used to expand a file, that is, to add records onto the end of it. The statements

```
    Seek(FileVar,FileSize(FileVar));
    Write(FileVar,Rec);
```

will append *Rec* onto the end of the disk file that *FileVar* is assigned to. You can continue to do this until you run out of disk space. Note, though, that you cannot seek beyond *FileSize(File Var)*; that, too, will result in an I/O error.

There is one more function that TURBO Pascal provides for random access of disk files: *FilePos*. The expression *FilePos(File Var)* returns the number (0..*FileSize*) of the record that *FileVar* is currently pointing at.

18.2 TEXT FILES

Pascal differs from BASIC, FORTRAN, and several other languages in that it writes data files as **binary** files, rather than **text** files. In other words, Pascal stores a two-byte integer value as a two-byte integer value, not as a character string several bytes long, with spaces and carriage returns as delimiters. This results in smaller data files and faster disk I/O (since there is no conversion from data to ASCII and back). This also allows you to do things like read in or write out an entire array with a single command, rather than having to use intricate loops and heavily formatted I/O statements.

Often, though, you **do** need to read and write text files—for example, when you are reading in user-entered data or writing out reports or other information. Pascal allows for that, too. You can use the predefined data type *Text*, which is equivalent to **file of** *Char*. (The latter is not available in TURBO Pascal.) So, for example, you could define:

```
program FileTest;
var
  InFile,OutFile      : Text;
begin
  Assign(InFile,'INPUT.TXT');
  Reset(InFile);
  Assign(OutFile,'OUTPUT.TXT');
  Rewrite(OutFile);
  .
  .
  .
end. { of program FileTest }
```

You can now read in text from *InFile*, and write it out to *OutFile*, using the following procedures:

Read(F,P1,...,Pn)	reads in the variables P1 through Pn from text file F
Readln(F,P1,...,Pn)	reads as above, plus advances to next line
Write(F,P1,...,Pn)	writes variables P1 through Pn into text file F, all on same line
WriteLn(F,P1,..,Pn)	writes as above and advances to next line

These routines are pretty much what they appear to be. The parameters can be variables (and, for Write and *WriteLn*, constants and expressions) of type Integer, Real, Char, and String (*Write/WriteLn* can also handle Boolean). *ReadLn* and *WriteLn* don't need to have parameters at all; they can simply be used to read and write end-of-line markers. If you leave the file variable (F) out, then *Read* and *ReadLn* default to keyboard input, while *Write* and *WriteLn* default to screen output.

Let's say that you wanted to write out the data for your game in textfile format, and had *InFile* and *OutFile* declared as *Text*. Our game save routine might look like this:

```
Assign(OutFile,'SHIPS.TXT');
Rewrite(OutFile);
WriteLn(OutFile,ShipCnt:4);
for Indx := 1 to shipcnt do
  with Ship[Indx] do begin
    Write(OutFile,SX:4,SY:4,QX:4,QY:4);
    WriteLn(OutFile,Energy:5,Beams:4,Shields:4)
  end;
Close(OutFile);
```

(The formatting notation, **p:n**, means to right-justify the value of **p** within a field of **n** characters. More on this later.)

Your ship information has now been stored out as a text file, one which you could edit or print. If you had three ships, at various locations, with full energy (1023), and with beams and shields set at 0, the file SHIPS.TXT might look like this:

```
3
7     2     50   175    1023      0   0
3    10    199    12    1023      0   0
5     5      0    47    1023      0   0
```

Having saved your game information as text, you can (and must) read it back in the same way. The following code would suffice:

```
Assign(InFile,'SHIPS.TXT');
Reset(InFile);
Readln(InFile,ShipCnt);
for Indx := 1 to ShipCnt do
  with Ship[Indx] do
    ReadLn(InFile,SX,SY,QX,QY,Energy,Beams,Shields);
Close(InFile);
```

18.3 THE EOLN FUNCTION

Earlier, we discussed the Boolean function **eof(f)** which tells you when you've reached the end of a file. However, text files have an additional delimiter: **end of line**. At times, you may want to read in a line item by item and know when you've hit the end of the line. How can you tell? By using the Boolean function **eoln(f)**, which tells you just that.

Let's suppose that you have the following text file:

```
342  3  1123
41  1772  1  2  33  5
65
441  233  23  67
```

You don't know ahead of time how many numbers there are on a line, but you still need to read them in. The following chunk of code will read them into an array:

```
program eolntest;
const
  nummax            =  50;
var
  infile            :  text;
  numlist           :  array[1..nummax] of integer;
  numcnt            :  integer;
```

```
begin
  reset(infile);                              { again, this will vary }
  numcnt := 0;                                { nothing in array yet}
  while not eof(infile) and (numcnt < nummax) do begin
   while not eoln(infile) and (numcnt < nummax) do begin
     numcnt := numcnt + 1;
     read(infile,numlist[numcnt])
   end;
   readln(infile);        { read in end of line}
  end;
  for indx := 1 to numcnt do
   writeln('numlist[',indx:2,'] = ',numlist[indx]:5);    close(infile)
end. { of program eolntest }
```

In the innermost loop, you read numbers off of a line, one by one, until you hit the end of the line (**eoln**). You then advance to the next line (using **readln**) and continue the process until you either hit the end of the file (**eof**) or have filled up the array (**numcnt** = nummax). At the end, for a check, you write all of the values out to the screen.

18.4 FORMATTED OUTPUT

Pascal has been accused of having lousy I/O capabilities, usually by people who are familiar with BASIC, FORTRAN, etc., and who are used to the extensive I/O formatting commands available. Actually, TURBO Pascal has extremely powerful I/O capabilities and is capable of doing things that cannot even be done if you're programming in BASIC or FORTRAN.

TURBO Pascal contains formatting capabilities. You can write out Integer (and Byte), Real, Char, Boolean, and String values (variables, constants, expressions). For all of those, you can specify a field width, using the notation *Parm:N*, where *N* is the width (in characters) of the field. If *N* is greater than the number of characters that *Parm* actually requires, then blanks are added **in front of** *Parm* to expand it to the proper width (this is called **right-justification**, because you're forcing *Parm* to be flush with—or justified against—the right-hand side of the field). For example, the statements

```
Indx := 16421;
WriteLn('This field is too narrow:',Indx:3);
WriteLn('This field is just right:',Indx:6);
WriteLn('This field is too wide    :',Indx:15);
```

would produce

```
This field is too narrow:16421
This field is just right: 16421
This field is too wide    :          16421
```

Note that if you specify a field that is too narrow, the field automatically expands by the minimum amount necessary to write the value out. Specifying no field at all is equivalent to giving one that is too short: just enough space is allocated to write the value out, and no more. This can cause problems if you're writing more than one value per line. For example, if you had written

```
Write(OutFile,SX,SY,QX,QY);
WriteLn(OutFile,Energy,Beams,Shields);
```

in the code to save the ship information out to a text file, the file SHIPS.TXT would look like this instead:

```
3
7250175102300
31019912102300
55047102300
```

Reading the information back in would be impossible, since you would be unable to separate the fields into their different values. So it pays to use a width specifier whenever appropriate.

As mentioned above, you can write out Boolean values, with or without a width parameter. The values translate into the strings 'TRUE' and 'FALSE', so that the statements:

```
Flag1 := True; Flag2 := False;
Writeln('Flag1:   ',Flag1:5,'   Flag2:   ',Flag2:5);
```

would produce the output

```
Flag1:  TRUE  Flag2:   FALSE
```

You **cannot**, however, read in TRUE and FALSE as boolean values. Instead, you would have to read them in as strings and convert accordingly.

Real values have their own special output rules for TURBO Pascal. If no field width is specified, then the value is written out in an 18-character-wide field, with the format

```
bsd.ddddddddddEtdd          where b =  blank (' ')
                                  s =  blank or '-'
                                  d =  digit ('0'..'9')
                                  t =  '+' or '-'
```

If a single field width (*Parm:N*) is given, then the same basic format is used, with some adjustments for field width:

```
 <b>sd.<d>Etdd             where <b>  is 0 or more blanks
                                 <d>  is 1 to 10 digits
```

The minimum value for *N* is 7 (8, if *Parm* < 0.0). As *N* gets large, the number of digits is <d> expands until it hits 10. After that, the number of blanks in <b> starts increasing. The format given above for no width parameter is equivalent to *Parm*:18.

You can specify a second field width for reals (*Parm:N:M*). In that case, *M* represents the number of digits after the decimal point that should be displayed, and fixed-point (rather than floating point) notation is used. *M* must be less than or equal to *N*-3. If *M* equals 0, then no decimal point and no following digits are displayed. If *M* is greater than 24, then floating-point notation is used.

If all of this seems confusing, here are some examples to show you how everything works:

```
program FormatDemo;
const
  Pi      = 3.1415926535;
begin
  WriteLn(Pi);
  WriteLn(Pi:8);
  WriteLn(-Pi:8);
  WriteLn(Pi:12);
  WriteLn(Pi:16);
  WriteLn(Pi:20);
  WriteLn(Pi:8:0);
  WriteLn(Pi:8:4);
  WriteLn(Pi:12:10)
end. { of program FormatDemo }
```

will produce the output

```
  3.1415926535E+00
3.14E+00
-3.1E+00
3.141593E+00
3.1415926535E+00
    3.1415926535E+00
       3
  3.1416
3.1415926535
```

18.5 FILENAMES

TURBO Pascal runs under CP/M, CP/M-86 and MS/DOS (a version is available for each), so it follows the file conventions of each system, which are essentially the same: a file name of up to eight (8) letters and/or digits, with an optional 3-character extension. If the extension exists, it is separated from the file name by a period. For example, the following file names are acceptable:

```
thisfile
thxl138
simple.pas
simple.exe
stars.dat
```

If you use these file names "as-is", then your program will assume that they are on the currently logged drive (as defined either by TURBO Pascal, if it's loaded, or by the operating system prompt "X>", where "X" is the logged drive). You can, of course, explicitly state which drive the file is on by appending the drive name in front of the file name:

```
    a:thisfile
    b:thxl138
    c:stars.dat
```

18.6 UNTYPED FILES

Up until now, all files you've looked at have been textfiles (= Text) or data files (= **file of**<type>). However, there are times when you want to deal with "raw" data, i.e., data that hasn't been formatted for you. For example, in some programs, you need to have many different types of data out on the disk. However, you don't want the overhead of having many files open at once or having to constantly open and close files.

The solution? You put everything into one large disk file and use an **untyped file** to access it. An untyped file is declared as follows:

```
var
  BigFile       : file;
```

It's opened and closed just like any other file. Reading from and writing to it are different, though, from other files. Untyped files can only be read or written a block (128 bytes) at a time. You do this using two procedures, *BlockRead* and *BlockWrite*. Here's their format:

```
    BlockRead(FileVar,Buf,NumBlocks);
    BlockWrite(FileVar,Buf,NumBlocks);
```

FileVar is, of course, the untyped file variable (such as *BigFile* above). Buf can be any type of variable; most often, it's an array of some sort. *NumBlocks* is the number of blocks that you want to read or write. Since each block is 128 bytes, *SizeOf(Buf)* must be greater than or equal to 128**NumBlocks*. This is **critical**. These two routines do not do any range-checking, and if *Buf* is too small, then a *BlockRead* will cheerfully copy the requested blocks to memory, overwriting whatever code and/or variables follow *Buf*. A *BlockWrite* in such a case is not necessarily as disastrous—you merely copy extra garbage out to the file—but even that can catch up with you later.

All of the standard file routines (except for *Read, Write*, and *Flush*) work on an untyped file. This means, among other things, that you can do random access on the file. For an untyped file, *Seek* will assume that each block is one record, with the first block being record #0. You can directly move to a given block, then use *BlockRead* and *BlockWrite* to access it (and those that follow it). A word of caution, though: if you are opening an existing file to update it (with *BlockRead* and/or *BlockWrite*), be sure to use the *Reset* procedure. Just as with data files and text files, a call to *Rewrite* erases any existing file and creates a new one.

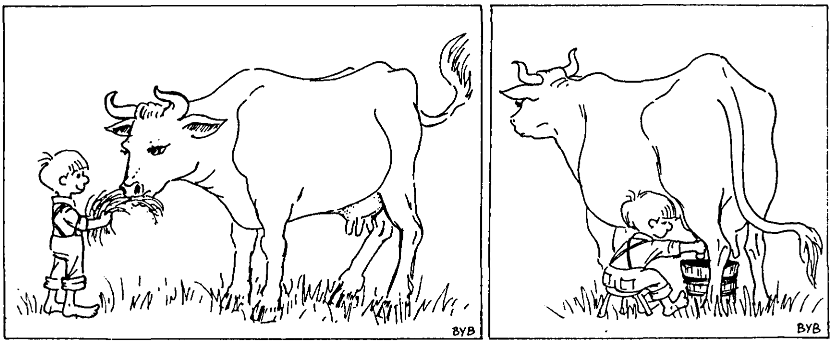
Let's illustrate all this with an example. Suppose that our "Trek" program has one data file to contain all game information: sectors, ships, and so on. Suppose that blocks 10 through 13 of your data file contain records which, for convenience's sake, are 32 bytes long each. Each block then has 4 of these records in it, and there are 16 (4 * 4) records in all. You could then write the following routine to get or put a specific record (numbered 0 through 15) from that chunk of the file. Let's say that record type is *BaseRec* and that your untyped file (*BigFile*) is already open.

```
procedure BaseRecIO(Indx        : Integer;
                    ReadFlag  :  Boolean;
                     var Rec  :  BaseRec);
{
does read/write for Rec[Indx]
if ReadFlag = True, then reads Rec, else writes it
}
var
  Bcnt,IBlk,IRec,IErr       : Integer;
                           : array[0..3] of BaseRec;
```

```
begin
  Indx := Abs(Indx) mod 16;          { force to allowable range }
  IBlk := Indx div 4 + 10;           { calculate block #}
  IRec := Indx mod 4;                { calculate rec w/in block }
  Seek(BigFile,IBlk)
  BlockRead(BigFile,Data,IBlk)       { get the data }
  if ReadFlag
   then Rec := Data[IRec]            { get appropriate record}
  else begin
   Data[IRec] := Rec;                { else save it in 'data'}
   Seek(BigFile,IBlk)
   BlockWrite(BigFile,Data,IBlk)        { & write it }
  end
end; { of proc BaseRecIO }
```

By writing similar routines for the other data types stored in *BigFile*, you can have ready access (and random access, at that) for a wide variety of data types with a minimum of overhead. If the size of the buffer (128 bytes or some multiple thereof) bothers you, you can always use *New* to create it on the heap and *Dispose* (or *Mark* and *Release*) to get rid of it. Actually,since it is a local variable, it is already allocated and deallocated on the fly.

18.7 DEVICE I/O



-Input/Output—

Most applications require I/O involving the computer hardware itself. Reading from the keyboard and writing out to the screen are the two most obvious examples. Standard Pascal (and TURBO Pascal) predefines the textfiles *Input* and *Output* for just those functions. All *Read* and *Write* statements without a file variable use these two files. But there are other times when you might want to read from or write to a specific device. How do you do this?

Simple. You use a set of special filenames that refer to hardware devices rather than to disk files. Consider, for example, the following program:

```
procedure Convert;
const
  OutFileName           = 'STARS.DAT';
type
  StarArray             = array[1..3] of Real;
var                    { Stars[1] = X, Stars[2] = Y, Stars[3] = Z }
  InFileName            : string[30];
  InFile                : Text;
  OutFile               : file of StarArray;
  Star                  : StarArray;
begin
  Write('Enter input file:   '); ReadLn(InFileName);
  Assign(InFile,InFileName);
  Reset(InFile);
  Assign(OutFile,OutFileName);
  Rewrite(OutFile);
  while not EOF(InFile) do begin
    ReadLn(InFile,Star[1],Star[2],Star[3]);
    Write(OutFile,Star)
  end;
  Close(InFile); Close(OutFile)
end. { of program Convert }
```

This program prompts the user for the name of an input file, which it expects to be a text file with three real numbers on each line. It reads those three numbers into *Star*, then writes *Star* out to the data file STARS.DAT. In short, it's converting the star information from a text file to a data file.

Now, suppose you wanted to enter the data manually instead of having it read from a disk file. All you would have to do is give the filename 'CON:' when asked for the input file. You would then enter the data, line by line. When you weredone, you would type ctrl-Z, which tells the program that it's reached the "end of file". By the same token, a program that writes a text file to disk could be redirect to write the output to the screen by also using 'CON:'. CON: is known as a **logical device**, and there are several other logical devices as well, representing keyboard input, screen output, the printer, the serial port, and other such items. Note that because these are character-oriented devices, only files of type *Text* should be connected with them.

Here's a list of the special filenames that TURBO Pascal recognizes:

CON:	console I/O, i.e., read from the keyboard and write to the screen. Echoes input, allows correction with backspace. Expands tabs on output. Echoes CR (carriage return) as CR/LF (carriage return/line feed) for both input and output.
KBD:	keyboard input with no echo or interpretation.
TRM:	console output with no interpretation.
LST:	the line printer. Can only be used for output. No interpretation. Tabs are not expanded.
AUX:	input and output device, usually an RS232 port. Corresponds to PUN: and RDR: in CP/M.
USR:	user I/O device. Advanced programmers can write their own I/O drivers for specific devices.

In *Convert*, you wanted to be able to specify the input file when you actually ran the program. Sometimes, you'll know ahead of time that you want to do I/O with a specific device. In that case, you don't have to define the file variable, assign it, and so on; instead, TURBO Pascal has predefined (and pre-assigned) file variables for each of these devices. In other words, you can use the file variables *Com, Trm, Kbd, Lst, Aux,* and *Usr* without having to do anything.

18.8 "REAL-TIME" KEYBOARD INPUT

In certain applications (such as games), you want the program to continue to run while at the same time checking for user input. In other words, the program doesn't sit and wait for you to enter a command, but instead would periodically check for and handle user commands. For example, the "Trek" game would be considerably more exciting if the attacking ships continued to advance and fire while you were trying to make up your mind what to do.

TURBO Pascal makes this fairly easy, via the *Kbd* device and the Boolean function *KeyPressed.* As you might guess, *KeyPressed* returns *True* if the user has hit any key, and *False* otherwise. It does not sit and wait, but merely checks the console status. You might, then, write a routine like this:

```
procedure CheckCommand;
var
  Cmd          : Char;
begin
  if KeyPressed then begin
    Read(Kbd,Cmd);          { read key w/out echo }
    Cmd := UpCase(Cmd);          { force to upper case }
    case Cmd of
      .
      .
      .      { handle commands}
    else
      Write(Chr(7));        { beep at illegal cmd }
    end
  end
end; { of proc CheckCommand }
```

By scattering calls to *CheckCommand* through your program, you can give the illusion of "real-time" commands, with swift response to any user input at any point. You do need to be careful of one thing, though: make sure that you don't call *Check-Command* at a point where any of the commands could have side-effects on what's currently being done. For example, you shouldn't call it if the program is updating the display and *CheckCommand* can change that display. Instead, call *Check-Command* just before or just after that section of code.

18.9 I/O ERROR HANDLING

Suppose that you ran *Convert* (the program listed above), and that you gave it the name of a file that didn't exist. What would happen? As soon as the program tried to open that file for input (via the *Reset* call), you would get an I/O error, and the program would abort. This, of course, is something of a pain, more so if you were in the middle of a large, complex program rather than just at the start of a small, simple one. Fortunately, TURBO Pascal offers a solution to this problem. Using a **compiler directive**, you can disable the "abort on I/O error" feature for sections of code (or, if you wish, for the entire program). To turn off I/O error trapping, you insert the comment statement { $I-} into your program. When you want to turn it back on, you insert { $I+}. For example, you could rewrite part of *Convert* to read

```
{ $I-}
Write('Enter input file:   '); ReadLn(InFileName);
Assign(InFile,InFileName);
Reset(InFile);
{ $I+}
```

Of course, this by itself doesn't really solve your problem; the program won't bomb, but if the file doesn't exist, you won't be able to read in any stars. TURBO Pascal again comes to the rescue with a built-in function named *IOresult*. *IOresult* returns an error code based on the success or failure of the last I/O operation you tried to perform. If the operation was successful, it returns the value 0; otherwise, it returns a value indicating just what the problem was. For example, we could again rewrite the code above like this:

```
{ $I-}
repeat
   Write('Enter input file:    '); ReadLn(InFileName);
   Assign(InFile,InFileName);
   Reset(InFile)
until IOresult = 0;
{ $I+}
```

The I/O compiler directive, along with *IOresult*, can be used in many places to develop "bulletproof" programs, that is, programs which cannot be bombed through various problems.

There are some important things you need to be aware of in using these techniques. First, any call to *IOresult* returns the current value, then resets the error condition to 0. This means that you can't keep calling *IOresult* and expect it to continue to return the error code for the last problem you ran into. Suppose you had written

```
{ $I-}
repeat
   Write('Enter input file:    '); ReadLn(InFileName);
   Assign(InFile,InFileName);
   Reset(InFile);
   if IOresult > 0
     then WriteLn('File not found; try again')
until IOresult = 0;
{ $I+}
```

If you entered the name of a non-existent file, you would get the message 'File not found; try again'. However, the call to *IOresult* that caused that message would clear the error condition. *IOresult* would then always return 0 at the **until** statement, and the program would never go back and ask you to re-enter the file name. What you need, instead, is something like this:

```
{ $I-}
repeat
   Write('Enter input file:    '); ReadLn(InFileName);
   Assign(InFile,InFileName);
   Reset(InFile);
   IOerr := (IOresult <> 0);
   if IOerr
     then WriteLn('File not found; try again')
until not IOerr;
{ $I+}
```

where *IOerr* is a variable of type Boolean. That way, *IOerr* can "remember" that there was an I/O error, even after *IOresult* has been reset to 0.

There is a second caution in using { $I-} / { $I+} and *IOresult*. If an error does occur, your program must call *IOresult* before attempting addition I/O. In other words, if you attempt to do I/O when the error code is something other than 0, problems can occur (such as your program hanging). Because of this, it is a good idea either to limit your use of { $I- }/{ $I+ } to very specific sections of your program, or to write your own I/O checking routine and then call that anywhere where problems might occur. Assuming that you have set aside line 24 of the screen for error messages, your routine might look like this:

```
program MyProgram;
   .
   .
   .
const
   Bell        : Char = Chr(7);
   IOerr        : Boolean = False;
{ These are both typed constants }
   .
   .
   .
type
   Prompt        : string[80];
   .
   .
   .
procedure Error(Msg : Prompt);
{
      write error Msg out on line 24 of the screen
}
begin
   GoToXY(1,24); ClrEol;
   Write(Bell,Msg)
end; { of proc Error }
```

```
procedure IOcheck;
{
       check for I/O error; print message if needed
}
var
   IOcode        : Integer;
   Ch        : Char;
begin
   IOcode := IOresult;
   IOerr := (IOcode <> 0);
   if IOerr then begin
     case IOcode of
       $01        : Error('File does not exist');
       $02        : Error('File not open for input');
       $03        : Error('File not open for output');
       $04        : Error('File not open');
       $10        : Error('Error in numeric format');
       $20        : Error('Operation not allowed on logical device');
       $21        : Error('Not allowed in direct mode');
       $22        : Error('Assign to standard files not allowed');
       $90        : Error('Record length mismatch');
       $91        : Error('Seek beyond end-of-file');
       $99        : Error('Unexpected end-of-file');
       $F0        : Error('Disk write error');
       $F1        : Error('Directory is full');
       $F2        : Error('File size overflow');
       $FF        : Error('File disappeared')
     else
           Error('Unknown I/O error:    ');
           Write(IOcode:3)
       end; { of case }
     Read(Kbd,Ch)
   end
end; { of proc IOcheck }
```

This routine does a number of things. First, since it always calls *IOresult*, it clears the pending I/O error code. Second, it sets the global flag *IOerr*, so that other parts of the program will know whether or not there has been an error and can act accordingly. Third, it prints out an error message on line 24, pausing until the user hits any key (hence *"Read(Kbd,Ch)"*, which will read a single character without echoing it back to the screen). Fourth, it uses the **else** clause of the **case** statement to handle any undefined I/O errors.

(A programming note: this routine uses a single **case** statement for clarity. However, since the values in the **case** statement are so widely spread—using only 15 values within a range of 255—a combination of **if..then..else** and **case** statements might be more efficient.)

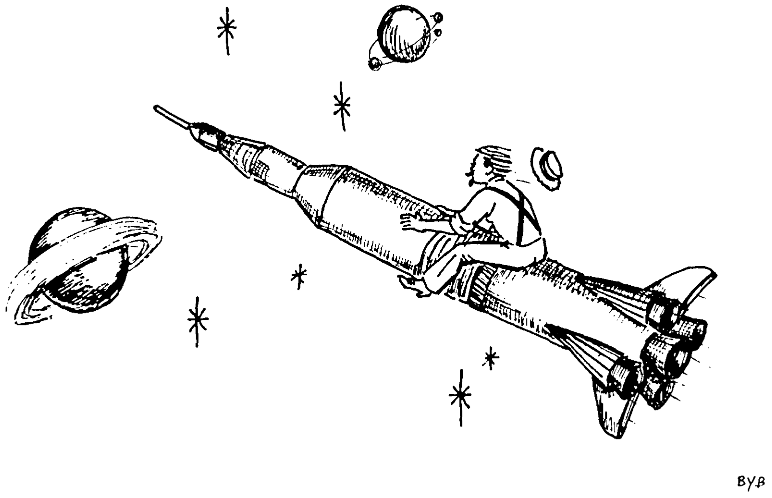
If we insert this code into *Convert*, then we might rewrite our loop to look like this:

```
{ $I-} program Convert;
   .
   .
begin { main body of Convert }
  repeat
    Write('Enter input file:    '); ReadLn(InFileName);
    Assign(InFile,InFileName);
    Reset(InFile); IOCheck
  until not IOerr;
   .
   .
   .
end. { of program Convert }
```

Since you are now using *IOcheck* here (and, presumably, elsewhere throughout your code), you can now turn off I/O error trapping for the entire program with a single { $I-} directive at the start of the program. Well, you and I have journeyed far. We have covered many, many aspects of programming using TURBO Pascal, and with some practice and practical application, you should be well on your way to solving some real problems of your own.

I've included a program called GAME1.PAS that brings many of these concepts together in a practical way, and lets you have fun besides. You should study it, compile it, and play with it for awhile. Then, when (and if) you feel up to it, you might want to tackle the advanced programming concepts presented in Part III.

But now you have the basis for doing just about anything you want to do. Your only limit is your imagination, and the sky is the limit...



—The Sky's the Limit—

PART III

ADVANCED TOPICS IN TURBO PASCAL

19. USEFUL TURBO PASCAL ROUTINES

Welcome to Part III! I am glad you made it this far. Since you are here, I assume that you are interested in some advanced programming techniques. Well, there's no shortage of those here. Happy programming!

Here are some program examples designed for advanced functions under TURBO Pascal. They aren't specific to an operating system (such as MS-DOS or CP/M); instead, they can be used anywhere. They're designed as utilities to help you put your programs together.

All of these programs can be found on your Turbo Tutor diskette. The file name of each program is given in the title for each section. These programs can all be compiled and executed immediately; no special hardware or software is assumed.

19.1 FUNCTION KEY DETECTION (FUNCKEYS.PAS)

This routine allows you to see what character or character sequences are produced by any special keys on your computer's keyboard. This will help you to use those keys for special functions in your programs. It does assume that any two-character sequences have ESC (Chr(27)) as the first character.

```
program GetFunctionKeyData;
{
      This program looks at keyboard input to see if a key was hit that
      generates a two-character sequence. On most keyboards, these two
      characters consist of chr(27) [ESC] plus an alphanumeric char.
}
var
  Ch           : Char;
  Previous     : Boolean;
  Count        : Integer;
```

```
begin
  for Count := 1 to 20 do begin
    Read(Kbd,Ch);              { Read a character, if ESC (chr(27) then }
    if (Ch = chr(27)) and keypressed then begin
                               { keystroke must be either ESC key or one }
      Previous := True;        { that generates a two-digit code }
      Read(Kbd,Ch);
    end
    else Previous := False;
    if Previous
      then Write('previous ')
      else Write('single char ');
    WriteLn('Ord(Ch)= ',Ord(Ch))
  end end. { of program GetFunctionKeyData }
```

19.2 BUFFERED INPUT (TYPEAHED.PAS)

This program shows how to guarantee that any characters typed by the user will be read. In other words, even if the program is off doing something else, characters typed will not be lost (up to the size of the buffer, the size of which depends upon your computer model and operating system).

```
{ $U-,C- }
program Buffered;
{
     The $C-directive is necessary for type-ahead (buffered)
     input; otherwise, characters will be lost. Also, since
     Read(Ch) require an end-of-line before processing, you
     must use the Read(Kbd,Ch). Type "#" to end the program.
}
var
  Ch              : Char;
  Indx,Jndx       : Integer;
begin
  repeat
    for Indx := 1 to 10000 do      { delay loop to show type-ahead }
      Jndx := Indx + Indx;
    Read(Kbd,Ch);                 { get next character from buffer }
    Write(Ch)                    { and echo it back out to the screen }
  until Ch = '#'                  { continue until "#" is entered }
end. { of program Buffered }
```

This program will delay between *each* character. Therefore, it could take a while to exit from if you typed many characters ahead.

19.3 I/O ERROR CHECKING (IOERROR.PAS)

This program contains a subroutine (*IOCheck*) along with two global variables (*IOVal* and *IOErr*) that should be a part of any program which you want to make "bullet-proof". You need to turn off I/O error trapping (using the $I- compiler option) and then makes calls to *IOCheck* after most (if not all) I/O operations. There are two important reasons for this. First, you (and your program) will then know if any errors have occured and can take appropriate steps. The global variables *IOVal* and *IOErr* can be used to test for specific problems. Second, if an error has occured, you need to read *IOresult* (which *IOCheck* does) to clear it before performing another I/O operation; otherwise, a second (and different) error can result.

```
{ $I- }
program TestIOCheck;
{
     The routine IOCheck, along with the global declarations
     IOFlag and IOErr, should be placed in any program where you
     want to handle your own I/O error checking.
}
const
  IOVal        : Integer = 0;
  IOErr        : Boolean = False;
var
  InFile       : Text;
  Line         : string[80];

procedure IOCheck;
{
     This routine sets IOErr equal to IOresult, then sets
     IOFlag accordingly. It also prints out a message on
     the 24th line of the screen, then waits for the user
     to hit any character before proceding.
}
var
  Ch           : Char;
```

```
begin
  IOVal := IOresult;
  IOErr := (IOVal < > 0);
  GotoXY(1,24); ClrEol;          { Clear error line in any case }
  if IOErr then begin
    Write(Chr(7));
    case IOVal of
      $01  :  Write('File does not exist');
      $02  :  Write('File not open for input');
      $03  :  Write('File not open for output');
      $04  :  Write('File not open');
      $05  :  Write('Can''t read from this file');
      $06  :  Write('Can''t write to this file');
      $10  :  Write('Error in numeric format');
      $20  :  Write('Operation not allowed on a logical device');
      $21  :  Write('Not allowed in direct mode');
      $22  :  Write('Assign to standard files not allowed');
      $90  :  Write('Record length mismatch');
      $91  :  Write('Seek beyond end of file');
      $99  :  Write('Unexpected end of file');
      $F0  :  Write('Disk write error');
      $F1  :  Write('Directory is full');
      $F2  :  Write('File size overflow');
      $FF  :  Write('File disappeared')
    else      Write('Unknown I/O error:    ',IOVal:3)
    end;
    Read(Kbd,Ch)
  end
end; { of proc IOCheck }

procedure PutLineNum(LineNum  :  Integer);
{
      This routine tells you which line is being executed,
      so that you can see which statement is causing which
      error.
}
begin
  GotoXY(1,1); ClrEol;
  Write('Executing line #',LineNum)
end; { of proc PutLineNum }

begin
  PutLineNum(1); Assign(InFile,'dummy');          IOCheck;
  PutLineNum(2); Rewrite(InFile);                 IOCheck;
  PutLineNum(3); Read(Infile,Line);               IOCheck;
  PutLineNum(4); Close(Infile);                   IOCheck
end. { of program TestIOCheck }
```

20. MS-DOS ROUTINES

—DOS Function Calls—

Over half of the copies of Turbo Pascal out there are running under MS-DOS, so this advanced section will start with some code designed to aid in interfacing with MS-DOS. These routines are meant to serve as examples for advanced programmers; by their very nature, they can get you into trouble if you're not sure of what you're doing.

All of these programs can be found on your TURBO Tutor disk (assuming that you have the MS-DOS version). The file name of each program is given in the title for each section. These programs can all be compiled and executed immediately; however, you should look at each program before running it, since it may expect certain things, such as a color graphics card or a software driver.

20.1 RANDOMIZE (RANDOM.PAS)

We'll start with fixing a problem in TURBO Pascal. As you may have noticed, the procedure *Randomize* does nothing at all in TURBO. Here is a procedure for MSand PC-DOS that replaces the built in *Randomize* with a working version.

The new Randomize has two Integer parameters. If they are both 0, then the random number seed is set randomly. If either of the parameters is nonzero, then they are both stored directly into the 32 bit seed.

To set the seed randomly (*Randomize*(0,0)), the procedure calls MS-DOS to get the current time. This is a 32 bit value, which is also stored directly into the seed. On some systems, (i.e. the NCR Decision Mate V), the clock does not tick, so the time never changes. *Randomize* checks this, and if the clock hasn't changed after a *Delay*(100), it asks the user to hit a key. While waiting for the key, it continuously increments two counters. These are then stored into the seed.

Please note: This routine is for MS-DOS/PC-DOS TURBO ONLY!

```
program RandomTest;
{
     This program tests out the Randomize procedure. It also
     calculates a chi-square value as a test of Random itself.
     Chi-square values between 3 and 16 are desirable, with
     values close to 8.3 being optimum.
}
var
  S1,S2,Indx,Jndx,Count         : Integer;
  Sum,T,NP                      : Real;
  Tally                         : array[0..9] of Integer;

procedure Randomize(I,J: Integer);
```

```
var
  RSet        : record
                  AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Integer;
                end;
  Ch          : Char;

begin
  if (I=0) and (J=0) then begin    { Generate a random random number
                                                                seed }
    RSet.AX:=$2C00;                           { DOS time of day function }
    MSDos(RSet);
    I:=RSet.CX;                             { Set I and J to the system time }
    J:=RSet.DX;
    Delay(100); { This delay may have to be increased for faster systems }
    MSDos(RSet);
    if (I=RSet.CX) and (J=RSet.DX) then begin      { Clock isn't ticking }
      I := 0;
      J := 0;
      while KeyPressed do
        Read(Kbd,Ch);                              { Clear keyboard buffer }
      Write('Hit any key to set the random number generator:  ');
      repeat
        I := I+13;
        J := J+17
      until Keypressed;
      Read(Kbd,Ch);                                { Absorb the character }
      WriteLn
    end
  end;
  MemW[DSeg:$0129]:=I; { This is the core of the routine: store a 32 bit }
  MemW[DSeg:012B]:=J;       {seed at locations DSeg:$0129..DSeg:$012B}
end;                                          { of procedure Randomize }

begin{ main body of program RandomTest }
  Writeln('Enter count <= 0 to end program');
  repeat
    Write('Enter count:           ');                 { get # of samples }
    ReadLn(Count);
    if Count > 0 then begin                        { do random number test }
      Write('Enter seeds (S1 S2):    ');           { get 2 integers for seed }
      ReadLn(S1,S2);
      Randomize(S1,S2);                           { set random number seed }
      FillChar(Tally,SizeOf(Tally),0);                  { clear tally array }
      for Indx := 1 to Count do begin            { generate Count numbers }
        Jndx := Random(10);                                 { range is 0..9 }
        Tally[Jndx] := Tally[Jndx] + 1           { count how many of each }
      end;
      Sum := 0.0;                                    { clear sum for X^2 }
      NP := Count/10.0;                     { theoretical number for each }
      for Indx := 0 to 9 do begin            { for each possible result do }
        Write(Tally[Indx]:5);                  { write total for that value }
        Sum := Sum + Sqr(Tally[Indx]-NP)/NP        { and calculate X^2 }
      end;
      WriteLn;
      WriteLn('Chi-Square (9 degrees of freedom) = ',Sum:8:3)
    end
  until Count <= 0
end. { of program RandomTest }
```

20.2 READ DIRECTORY (DIRECTRY.PAS)

TURBO Pascal gives you a lot of control over disk files, including the ability to erase files as well as to rename them. But nothing quite beats being able to read the directory off the disk.

Here's a simple program that will read the names of all the files off a given drive. You can also specify a mask, so that you only get files matching a given pattern.

```
program DirList;
{
      This is a simple program to list out the directory of the
      current (logged) drive.
}
type
  Char12arr        = array [ 1..12 ] of Char;
  String20         = string[ 20 ];
  RegRec =
    record
      AX, BX, CX, DX, BP, SI, DI, DS, ES, Flags : Integer;
    end;

var
  Regs             : RegRec;
  DTA              : array [ 1..43 ] of Byte;
  Mask             : Char12arr;
  NamR             : string20;
  Error, I         : Integer;

begin { main body of program DirList }
```

```
  FillChar(DTA,SizeOf(DTA),0);           { Initialize the DTA buffer }
  FillChar(Mask,SizeOf(Mask),0);         { Initialize the mask }
  FillChar(NamR,SizeOf(NamR),0);         { Initialize the file name }

  WriteLn( 'Directory list program for MS-Dos.' );
  WriteLn;
  RegsAX := $1A00;              { Function used to set the DTA }
  Regs.DS := Seg(DTA);          { store the parameter segment in DS }
  Regs.DX := Ofs(DTA);          { "      "    "offset in DX }
  MSDos(Regs);                  { Set DTA location }
  Mask := '????????.???';         { Use global search }
  Regs.AX := $4E00;               { Get first directory entry }
  Regs.DS := Seg(Mask);           { Point to the file Mask }
  Regs.DX := Ofs(Mask);
  Regs.CX := 22;                  { Store the option }
  MSDos(Regs);                    { Execute MSDos call }
  Error := Regs.AX and $FF;       { Get Error return }
  I := 1;                         { initialize 'I' to the first element }
  if (Error = 0) then
    repeat
      NamR[I] := Chr(Mem[Seg(DTA):Ofs(DTA)+29+I]);
      I := I + 1;
    until not (NamR[I-1] in [' '..'~']) or (I>20);

  NamR[0] := Chr(I-1);              { set string length because assigning }
                                    { by element does not set length }
  while (Error = 0) do begin
    Error := 0;
    Regs.AX := $4F00;               { Function used to get the next }
                                    { directory entry }
    Regs.CX := 22;                  { Set the file option }
    MSDos( Regs );                  { Call MSDos }
    Error := Regs.AX and $FF;       { get the Error return }
    I := 1;
    repeat
      NamR[I] := Chr(Mem[Seg(DTA):Ofs(DTA)+29+I]);
      I := I + 1;
    until not (NamR[I-1] in [' '..'~'] ) or (I > 20);
    NamR[0] := Chr(I-1);
    if (Error = 0)
      then WriteLn(NamR)
  end
end.                              { of program DirList}
```

20.3 READ DIRECTORY II (QDL.PAS)

Here's a more extensive directory-reading program.

```
program QDL;

{------------------------------------------------------------------------
      program QDL Version 2.00A                           09/01/84

      QDL uses MSDos to get a listing of an IBM formated diskette.
   The function calls used can be found in the DOS Technical Reference
   Manual. This program saves the current Data Transfer Area ( DTA ) in
   the variables DTAseg and DTAofs. The DTA is then reset to the Segment
   and Offset of a Buffer variable 'DTA'.

 ------------------------------------------------------------------------}
{ $I-,U-,C-}

type                                    { TYPE declarations }
  Registers =
    record                              { register pack used in MSDos call }
      AX, BX, CX, DX, BP, SI, DI, DS, ES, Flags : Integer;
    end;
  Char80arr       = array [ 1..80 ] of Char;
  String80        = string[ 80 ];

var                                     { VARIABLE declarations }
  DTA : array [ 1..43 ] of Byte;        { Data Transfer Area Buffer }
  DTAseg,                               { DTA Segment before execution }
  DTAofs,                               { DTA Offset       "        "    }
  SetDTAseg,                            { DTA Segment and Offset set after }
  SetDTAofs,                            { start of program }
  Error,                                { Error return }
  I, J,                                 { used as counters }
  Option : Integer;                     { used to specify file types }
  Regs : registers;                     { register pack for the DOS call }
  Buffer,                               { generic Buffer }
  NamR : string80;                      { file name }
  Mask : Char80arr;                     { file Mask }
```

```
{------------------------------------------------------------------------
SetDTA resets the current DTA to the new address specified in the
parameters 'SEGMENT' and 'OFFSET'.
------------------------------------------------------------------------}

procedure SetDTA( Segment, Offset : Integer; var Error : Integer );
begin
   Regs.AX := $1A00;              { Function used to set the DTA }
   Regs.DS := Segment;            { store the parameter Segment in DS }
   Regs.DX := Offset;             {   "     "      "     Offset in DX }
   MSDos( Regs );                 { Set DTA location }
   Error := Regs.AX and $FF;      { get Error return }
end; { of proc SetDTA }

{------------------------------------------------------------------------
GetCurrentDTA is used to get the current Disk Transfer Area ( DTA )
address. A function code of $2F is stored in the high Byte of the AX register
and a call to the predefined procedure MSDos is made. This can also be
accomplished by using the "Intr" procedure with the same register record
and a $21 specification for the interrupt.
------------------------------------------------------------------------}

procedure GetCurrentDTA( var Segment, Offset : Integer;
                         var Error : Integer );
begin
   Regs.AX := $2F00;         { Function used to get current DTA address }
                             { $2F00 is used instead of $2F shl 8 to save
                             three assembly instructions. An idea for
                             optimization. }
   MSDos( Regs );            { Execute MSDos function request }
   Segment := Regs.ES;       { Segment of DTA returned by DOS }
   Offset := Regs.BX;        { Offset of DTA returned }
   Error := Regs.AX and $FF;
end; { of proc GetCurrentDTA
{------------------------------------------------------------------------

GetOption returns the code used to find the file names on the current
directory ( ie. hidden, standard, or directory ).
------------------------------------------------------------------------}

procedure GetOption( var Option : Integer );
var
   Ch : Char;
begin
   Ch := '?';
   Option := 1;
   while ( Ch = '?' ) do begin
```

```
    Write( 'File Option to use, [ ? ] for list : ' );
    ReadLn( Ch );
    WriteLn;
    case ( Ch ) of
      '1' : Option := 1;        { -\                                    }
      '2' : Option := 7;        {    \                                  }
      '3' : Option := 8;        {      - These are the options.         }
      '4' : Option := 16;       {      - Look below for an expla-       }
      '5' : Option := 22;       {    /   of each.                       }
      '6' : Option := 31;       { - /                                   }
      '?' : begin               {  gives list of possible options       }
              WriteLn( 'File options are : ' );
              WriteLn;
              WriteLn( ' [ 1 ] for standard files [ default ].' );
              Write( ' [ 2 ] for system or hidden files ' );
              WriteLn( 'and standard files.' );
              WriteLn( ' [ 3 ] for volume label.' );
              Write( ' [ 4 ] for directories and ' );
              WriteLn( 'standard files.' );
              WriteLn( ' [ 5 ] for directories, hidden or ' );
              Write( '         system files, and standard' );
              WriteLn( ' files.' );
              Write( ' [ 6 ] same as 5, but with volume' );
              WriteLn( ' label included.' );
              WriteLn;
            end;
    else Option := 1;           { if nothing is typed or an }
    end;                        { incorrect entry is made Option 1 is used}
  end
end; { of proc GetOption }
```

{——
GetFirst gets the first directory entry of a particular file Mask. The Mask is passed as a parameter 'Mask' and,the Option was previously specified in the SpecifyOption procedure.
——}

```
procedure GetFirst( Mask : Char80arr; var NamR : String80;
                    Segment, Offset : Integer; Option : Integer;
                    var Error : Integer );
var
  I : Integer;
begin
  Error := 0;
  Regs.AX := $4E00;            { Get first directory entry }
  Regs.DS := Seg( Mask );      { Point to the file Mask }
  Regs.DX := Ofs( Mask );
```

```
   Regs.CX := Option;           { Store the Option }
   MSDos( Regs );               { Execute MSDos call }
   Error := Regs.AX and $FF;    { Get Error return }
   I := 1;                      { initialize 'I' to the first element }
   repeat                       { Enter the loop that reads in the }
                                { first file entry }
     NamR[ I ] := Chr( mem[ Segment : Offset + 29 + I ] );
     I := I + 1;
   until ( not ( NamR[ I - 1 ] in [ ' '..'~' ] ));
   NamR[ 0 ] := Chr( I - 1 );       { set string length because assigning }
                                    { by element does not set length }
end; { of proc GetFirst }

{______________________________________________________________________
GetNextEntry uses the first bytes of the DTA for the file Mask, and returns
the next file entry on disk corresponding to the file Mask.
 ______________________________________________________________________}

procedure GetNextEntry( var NamR : String80; Segment, Offset : Integer;
                        Option : Integer; var Error : Integer );
var
   I : Integer;
begin
   Error := 0;
   Regs.AX := $4F00;            { Function used to get the next }
                                { directory entry }
   Regs.CX := Option;           { Set the file option }
   MSDos( Regs );               { Call MSDos }
   Error := Regs.AX and $FF;    { get the Error return }
   I := 1;
   repeat
     NamR[ I ] := Chr( mem[ Segment : Offset + 29 + I ] );
     I := I + 1;
   until ( not ( NamR[ I - 1 ] in [ ' '..'~' ] ));
   NamR[ 0 ] := Chr( I - 1 );
end; { of proc GetNextEntry }

{
               main body of program QDL
}
```

```
begin
   for I := 1 to 21 do DTA[ I ] := 0;       { Initialize the DTA Buffer }
     for I := 1 to 80 do begin               { Initialize the Mask and }
        Mask[ I ] := Chr( 0 );               { file name buffers }
        NamR[ I ] := Chr( 0 );
     end;
   NamR[ 0 ] := Chr( 0 );                    { Set the file name length to 0 }
   WriteLn( 'QDL version 2.00A' );
   WriteLn;
   GetCurrentDTA( DTAseg, DTAofs,
   Error );                                  { Get the current DTA address }
   if ( Error <> 0 ) then begin              { Check for errors }
     WriteLn( 'Unable to get current DTA' );
     WriteLn( 'Program aborting.' );         { and abort. }
     Halt;                                   { end program now }
   end;
   SetDTAseg := Seg( DTA );
   SetDTAofs := Ofs( DTA );
   SetDTA( SetDTAseg, SetDTAofs,
   Error );                                  { Reset DTA addresses }
   if ( Error <> 0 ) then begin              { Check for errors }
     WriteLn( 'Cannot reset DTA' );          { Error message }
     WriteLn( 'Program aborting.' );
     Halt;                                   { end program }
   end;
   Error := 0;
   Buffer[ 0 ] := Chr( 0 );                  { Set Buffer length to 0 }
   GetOption( Option );                      { Get file Option }
   if ( Option <> 8 ) then begin
     Write( 'File Mask :' );                 { prompt }
     ReadLn( Buffer );
     WriteLn;
   end;
   if ( length( Buffer ) = 0 ) then          { if nothing was entered }
     Buffer := '????????.???';               { then use global search }
   for I := 1 to length( Buffer ) do         { Assign Buffer to Mask }
     Mask[ I ] := Buffer[ I ];
   GetFirst( Mask, NamR, SetDTAseg, SetDTAofs, Option, Error );
   if ( Error = 0 ) then begin               { Get the first directory entry }
     if ( Option <> 8 ) then begin           { if not volume label }
      WriteLn( 'Directory of : ', Buffer );{ Write directory message }
      WriteLn;
     end;
     WriteLn( NamR )
   end
   else if ( Option = 8 ) then
     WriteLn( 'Volume label not found.' )
```

```
  else WriteLn( 'File ''', Buffer, ''' not found.' );
  while ( Error = 0 ) do begin
    GetNextEntry( NamR, SetDTAseg, SetDTAofs, Option, Error );
    if ( Error = 0 ) then WriteLn( NamR );
  end;
  SetDTA( DTAseg, DTAofs, Error );
end. { end Main }
```

20.4 DISK STATUS (DISKSTUS.PAS)

This program reads information about free space on a given drive. Note that the function *DefaultDrive* can be used to determine the current logged drive.

```
program ShowDiskStatus;

{
     ShowDiskStatus uses MSDos and the functions therein to get
     Drive information on either the current Drive or the Drive
     specified on the command line.
}
{ $I-,U-,C-}

type                          { TYPE declarations }
  RegRec =
    record                    { register pack Used in MSDos call }
      AX, BX, CX, DX, BP, SI, DI, DS, ES, Flags : Integer;
    end;

var
  Tracks,                     { number of available Tracks }
  TotalTracks,                { number of total Tracks }
  Drive,                      { Drive number }
  Bytes,                      { number of Bytes in one sector }
  Sectors       : Integer;    { number of total Sectors }
  Used,TotalBytes : Real;
  Regs          : RegRec;

procedure DiskStatus( Drive : Integer;var Tracks, TotalTracks,
                      Bytes, Sectors : integer );
{
     makes MSDos call to read status of Drive; returns data
     in Tracks, TotalTracks, Bytes, and Sectors
}
```

```
begin
   Regs.AX := $3600;            { Get Disk free space }
   Regs.DX := Drive;            { Store Drive number }
   MSDos( Regs );               { Call MSDos to get disk info }
   Tracks := Regs.BX;           { Get number of Tracks Used }
   TotalTracks := Regs.DX;      {  "    "     "  of total Tracks }
   Bytes := Regs.CX;            {  "    "     "  of Bytes per sector }
   Sectors := Regs.AX           {  "    "     "  of Sectors per cluster }
end; { of proc DiskStatus }

function DefaultDrive : Integer;
{
      makes MSDos call to find out what current default drive is
}
var
   Regs : RegRec;
begin
   Regs.AX := $1900;            { Get current Drive number }
   MSDos( Regs );               { Call MSDos }
   DefaultDrive := (Regs.AX and $FF)  +  1     { Return value via function }
end; { of func DefaultDrive }

begin { main body of program ShowDiskStatus }
   Drive := 0;                  { Initialize Drive }
   if (Mem[Cseg:$80]) > 0       { Get command line }
     then Drive := Mem[Cseg:$82] and $1F;
   if not ( Drive in [  1..6  ] )  { If nothing on command line }
     then Drive := 0;           { or bad drive specified,}
                                { then logged drive }
   DiskStatus( Drive, Tracks, TotalTracks, Bytes, Sectors );
   WriteLn;
   WriteLn;
   Write( '          ' );
   if ( Drive = 0 ) then
     Drive := DefaultDrive;
   WriteLn( 'DSCST on Drive ', chr( Drive + $40 ), ':');
   WriteLn;
   { Write disk information }
   WriteLn( Tracks:7, 'available tracks.' );
   WriteLn( TotalTracks:7, 'total tracks.' );
   Used :=(( TotalTracks - Tracks ) / TotalTracks ) * 100;
   WriteLn( Used:7:2, '% used.' );
   WriteLn( Sectors:7, '    sectors per cluster.' );
   WriteLn( Bytes:7, '    bytes per sector.' );
   TotalBytes := (( Sectors * Bytes * 1.0 ) * Tracks );
   WriteLn( TotalBytes:7:0, '    total bytes available on disk.' );
   WriteLn
end. { of program ShowDiskStatus }
```

20.5 DOS VERSION NUMBER (VERSION.PAS)

Here's a much simpler routine, designed to allow you to read the DOS version number:

```
program DosVersion;

function DosVer : Real;
var
   Regs        :  record
                     AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags : Integer;
                  end;
   AL,AH        : Byte;

begin
   Regs.AX := $3000;
  MSDos( Regs );
  AL := Regs.AX and $FF;
  AH := ( Regs.AX and $FF00 ) shr 8;
  DosVer := AL + AH/100;
end; { of func DosVer }

begin { main body of program DosVersion }
   WriteLn;
   WriteLn;
   WriteLn( DosVer:4:2 );
end. { of program DosVersion }
```

20.6 DIRECT VIDEO OUTPUT (IBMINT10.PAS)

This routine uses Interrupt 10 to write directly to the screen. This allows you to do fast, unfiltered screen updates. It is designed to work only on the IBM PC. It is **not** generic.

```
program IBMpcScreen;
     { This program shows the use of direct Video on the IBM-PC ( only!!!)). }
     { Interrupt 10 is used for all Video I/O                    }
```

```
const
  Video             =$  10;  { Set Video I/O Interrupt }
  SetVideo          =    0;  { Set Video mode }
  SetCurPosition    = $200;  { Set cursor position }
  ReadCursor        = $300;  { Read cursor position }
  WriteChar         =$E00;   { Write character to sceen }
  VideoBW80x25A     =    2;  { Mode 80x25 B/W, Alpha }
  VideoColor80x25A  =    3;  { Mode 80x25 Color, Alpha }
type
  Result =                   { Register pack }
    record
      AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Integer;
    end;
var
  Rec                        : Result;
  Row,Col                    : Integer;
begin
  Rec.AX := SetVideo + VideoBW80x25A;   { assumes BW 80x25 display }
  Rec.BX := 15;
  Intr(Video,Rec);                                  { Set Video Mode }
  Rec.AX := WriteChar + Ord('A');
  Intr(Video,Rec);                                      { Output 'A'}
  Rec.AX := ReadCursor;
  Intr(Video,Rec);                        { Read the cursor position }
  Row := Rec.DX and $FF00 shr 8;
  Col := Rec.DX and $FF;
  Write('Row =  ',Row,' Column =  ',Col);  { Show the Row and column }
  Rec.AX := SetCurPosition;
  Rec.DX := $0A0A;
  Intr(Video,Rec);          { Set the cursor to Row 10 and column 10 }
  Rec.AX := WriteChar + Ord('#');
  Intr(Video,Rec);                                      { Output '#'}
  Rec.AX := ReadCursor;
  Intr(Video,Rec);                        { Read the cursor position }
  Row := Rec.DX and $FF00 shr 8;
  Col := Rec.DX and $FF;
  Write('Row =  ',Row,' Column =  ',Col);  { Show the Row and column }
  Rec.AX := SetCurPosition;
  Rec.DX := $1414;
  Intr(Video,Rec);          { Set the cursor to Row 20 and column 20 }
end. { of program IBMpcScreen }
```

—Interrupts—

20.7 DIRECT MEMORY OUTPUT (MEMSCREN.PAS)

Here's a second way to write directly to the screen: direct memory addressing. The following program shows how. It's set up for the Color Graphics display, but can be easily changed to work with the Monochrome by changing the *Segment* and *Offset* addresses used in the subroutine X. Again, this program is for the IBM PC only; it is **not** generic.

```
program ScreenMap;
  {program to Write to the screen map}
const
    ColorSeg                  = $B000;
    ColorOfs                  = $8000;

var
    I,J                       : Integer;
    C                         : Char;
    RowNumber,ColNumber,Length,Dir
                              : Integer;
    Direction                 : Char;

procedure X(J,K : Integer);
begin
  Mem[ColorSeg:ColorOfs + J] := K              { Write to screen memory }
end; { of proc X }

procedure Y(R,C,N,Ch,D : Integer); {r=row C=column n=length ch=char}
var
  I,J : Integer;
begin
  J :=((R-1)*160) + ((C-1)*2);               { compute starting location }
  for I := 1 to N do begin
    X(J,Ch);                                            { loop n times }
    if D=0
      then J := J + 160
      else J:= J+2;
  end
end; { of proc Y }

begin
  ClrScr;
  GotoXY(20,10);
  Write('Do you want to play <Y/N> ? ');
  Read(kbd,C);
  if (C = 'y') or (C = 'Y') then begin
    repeat
      ClrScr;
      GotoXY(20,10);
      Write('Would you like to draw a line <Y/N> ? ');
      Read(kbd,C);
    until (C = 'y') or (C = 'Y') or (C = 'n') or (C = 'N');
    if (C = 'y') or (C = 'Y') then begin
      repeat
        ClrScr;
        Y(1,1,80,31,1);
        Y(2,1,24,16,0);
        Y(2,80,24,17,0);
        Y(25,2,78,30,1);
        GotoXY(20,10);
        Write('The line starts at what row? ');
        Read(rownumber);
        GotoXY(20,12);
        Write('                 what column? ');
        Read(colnumber);
        GotoXY(20,14);
        Write('       Line length in number? ');
        Read(length);
        GotoXY(20,16);
        Write('       What character to use? ');
        Read(Kbd,C); Write(C);
        repeat
          GotoXY(10,18);Write('which direction<d>own or<a>cross? ');
          Read(Direction)
        until (Direction='a') or (Direction='d');
        if Direction='a'
          then Dir:=1
          else Dir:=0;
        ClrScr;
        Y(rownumber,colnumber,length,ord(C),dir);
        GotoXY(40,24);
        Write('Your line. Try again? ');
        Read(Trm,C);
        until (C<>'Y') and (C<>'y');
      ClrScr
    end
  end
  else begin
    GotoXY(23,13);
    WriteLn('******** Bye ********')
  end
end. { of program ScreenMap }
```

20.8 READING THE COMMAND LINE (CMDLINE.PAS)

This next routine can come in very handy. It reads the command line after the name of your program. This allows you to pass parameters for your program to use. For example, you could write

```
A>convert stars.txt stars.dat
```

and pick up the string 'starts.txt starts.dat' to determine what files to use.

```
program CommandLine;

{ This program demonstrates how to get information off the command
  line. One thing you must remember— 32 characters are always
  there for you to use— if you want to use the full 127, the first
  statement in your program must parse the command line and retrieve
  the information as any subsequent reads or writes will shorten the
  command line to 32 characters. }

type
  CommandString= string[127];

var
  Buffer        : CommandString;
  CL            : CommandString absolute cseg:$80;

begin
  Buffer := CL;          { read the command line immediately }
  Gotoxy(20,12);
  WriteLn('|',Buffer, '|');
end. { of program CommandLine }
```

20.9 SERIAL PORT LIBRARY (COMLIB.PAS)

While the standard file *Aux* lets you talk through the serial port, you may want something more powerful or sophisticated. Here are some routines designed to let you select which port to use, set the baud rate, set the number of stop bits, set the number of data bits, and select odd, even, or no parity. This is **not** a generic program and is intended only for the IBM PC.

```
{ $U+}
program ComLibTest;
var
  Port,Baud,StopBits,DataBits,Par: Integer;
  Message: string[80];

type
  string19=string[19];
```

```
{ A set of routines to enable COM1 and COM2 to be accessed from Turbo
Pascal. The following procedures are meant to be called by your programs:

AssignAux(PortNumber in [1,2]) assigns Aux to COM1 or COM2
AssignUsr(PortNumber in [1,2]) assigns Usr to COM1 or COM2
SetSerial(PortNumber in [1,2],
          BaudRate in [110,150,300,600,1200,2400,4800,9600],
          StopBits in [1,2],
          DataBits in [7,8],
          Parity in [None,Even,Odd]) sets the baud rate, stop bits, data
                bits, and parity of one of the serial ports.

The arrays InError and OutError may be examined to detect errors. The
bits are as follows:
     Bit 7 (128)        Time out (no device connected)
     Bit 3 (8)          Framing error
     Bit 2 (4)          Parity error
     Bit 1 (2)          Overrun error

Function SerialStatus(PortNumber in [1,2]) returns a more complete
status:
     Bit 15 (negative)  Time out (no device connected)
     Bit 14 (16384)     Transmission shift register empty
     Bit 13 (8192)      Transmission holding register empty
     Bit 12 (4096)      Break detect
     Bit 11 (2048)      Framing error
     Bit 10 (1024)      Parity error
     Bit  9 (512)       Overrun error
     Bit  8 (256)       Data ready
     Bit  7 (128)       Received line signal detect
     Bit  6 (64)        Ring indicator
     Bit  5 (32)        Data set ready
     Bit  4 (16)        Clear to send
     Bit  3 (8)         Delta receive line signal detect
     Bit  2 (4)         Trailing edge ring detector
     Bit  1 (2)         Delta data set ready
     Bit  0 (1)         Delta clear to send

   Identifiers starting with "__" are not meant to be used by the user
program.
}

  type
   __RegisterSet=Record case Integer of
                   1: (AX,BX,CX,DX,BP,DI,SE,DS,ES,Flags: Integer);
                   2: (AL,AH,BL,BH,CL,CH,DL,DH: Byte);
                 end;
   __ParityType=(None,Even,Odd);
   var
     __Regs: __RegisterSet;
     InError,OutError: array [1..2] of Byte;

   procedure __Int14(PortNumber,Command,Parameter: Integer);
   { do a BIOS COM driver interrupt }

  begin
    with __Regs do
      begin
      DX:=PortNumber-1;
      AH:=Command;
      AL:=Parameter;
      Flags:=0;
      Intr($14,__Regs);
      end;
    end;

procedure SetSerial(PortNumber,BaudRate,StopBits,DataBits: Integer;
                              Parity: __ParityType);
{ Set serial parameters on a COM port }

  var
    Parameter: Integer;

    begin
      case BaudRate of
        110  : BaudRate:=0;
        150  : BaudRate:=1;
        300  : BaudRate:=2;
        600  : BaudRate:=3;
        1200: BaudRate:=4;
        2400: BaudRate:=5;
        4800: BaudRate:=6;
        else BaudRate:=7; { Default to 9600 baud }
      end;
      if StopBits=2 then StopBits:=1
      else StopBits:=0; { Default to 1 stop bit }
      if DataBits=7 then DataBits:=2
      else DataBits:=3; { Default to 8 data bits }
      Parameter:=(BaudRate shl 5)+(StopBits shl 2) + DataBits;
      case Parity of
        Odd: Parameter:=Parameter+8;
        Even: Parameter:=Parameter+24;
        else; { Default to no parity }
      end;
    __Int14(PortNumber,0,Parameter);
  end;
```

```
Function SerialStatus (PortNumber: Integer): Integer;
{ Return the status of a COM port }

begin
    __Int14 (PortNumber,3,0);
    SerialStatus := __Regs.AX;
end;

procedure __OutPort1(C: Byte);
{ Called by Write to Aux or Usr when assigned to COM1 }

  begin
    while (SerialStatus(1) and $30)=0 do ;
    __Int14(1,1,C);
    OutError[1]:=OutError[1] Or (__Regs.AH and $8E);
  end;

procedure __OutPort2(C: Byte);
{ Called by Write to Aux or Usr when assigned to COM2 }

  begin
    while (SerialStatuS(2) and $30)=0 do ;
    __Int14(2,1,C);
    OutError[2]:=OutError[2] or (__Regs.AH and $8E);
  end;

Function __InPort1: Char;
{ Called by Read from Aux or Usr when assigned to COM1 }

  begin
    __Int14(1,2,0);
    __InPort1:=Chr(__Regs.AL);
    InError[1]:=InError[1] Or (__Regs.AH and $8E);
  end;

Function __InPort2: Char;
{ Called by Read from Aux or Usr when assigned to COM2 }

  begin
    __Int14(2,2,0);
    __InPort2:=Chr(__Regs.AL);
    InError[2]:=InError[2] or (__Regs.AH and $8E);
  end;
```

```
procedure __AssignPort(PortNumber: Integer; var InPtr,OutPtr: Integer);
{ Assign either Aux or Usr to either COM1 or COM2 }

  begin
    if PortNumber=2 then
      begin
      OutPtr:=Ofs(__OutPort2);
      InPtr:=Ofs(__InPort2);
      end
    else { Default to port 1 }
      begin
      OutPtr:=Ofs(__OutPort1);
      InPtr:=Ofs(__InPort1);
      end;
    InError[PortNumber]:=0;
    OutError[PortNumber]:=0;
  end;

procedure AssignAux(PortNumber: Integer);
{ Assign Aux to either COM1 or COM2 }

  begin
    __AssignPort(PortNumber,AuxInPtr,AuxOutPtr);
  end;

procedure AssignUsr(PortNumber: Integer);
{ Assign Usr to either COM1 or COM2 }

  begin
    __AssignPort(PortNumber,UsrInPtr,UsrOutPtr);
  end;

Function Binary(V: Integer): String19;

  var
    I : Integer;      B: array [0..3] of string[4];
  begin
    for I:=0 to 15 do
      if (V and (1 shl (15-I)))<>0 then B[I div 4][(I mod 4)+1]:='1'
      else B[I div 4][(I mod 4)+1] := '0';
    for I := 0 to 3 do B[I][0] := Chr(4);
    Binary:=B[0]+' '+B[1]+' '+B[2]+' '+B[3];
  end;

begin
  Write('Enter port number:                              ');
  ReadLn(Port);
  AssignUsr(Port);
  Write('Enter baud rate:                                ');
  ReadLn(Baud);
  Write('Enter stop bits:                                ');
  ReadLn(StopBits);
  Write('Enter data bits:                                ');
  ReadLn(DataBits);
  Write('Enter parity (0=none, 1=even, 2=odd):           ');
  ReadLn(Par);
  Write('Enter message to print:                         ');
  ReadLn(Message);
  SetSerial(1,Baud,StopBits,DataBits,ParityType(Par));
  WriteLn(Usr,Message);
  WriteLn('OutError[',Port,']: ',Binary(OutError[Port]));
  WriteLn('SerialStatus(',Port,'): ',Binary(SerialStatus(Port)));
end. { of program ComLibTest }
```

20.10 MICROSOFT MOUSE INTERFACE (TBOMOUSE.PAS)

The Microsoft Mouse runs off of a hardware/software combination: a card which plugs into a slot, and a software driver (MOUSE.COM) which (when executed) installs itself in RAM. The following program shows how to read the Mouse from Turbo Pascal. This is **not** a generic program; it works only on the IBM PC.

```
program MouseSketch;
{
          This program shows how to read the Microsoft Mouse.
                          ***WARNING***
               Be sure that you have loaded the mouse driver
}         (by running MOUSE.COM) before executing this program.
```

```
type
   RegPack =
     record
       AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags : Integer;
     end;

var
   OldX,OldY,X,Y              : Integer;
   M1,M2,M3,M4                : Integer;
   RegPak                     : RegPack;

procedure Mouse(var M1,M2,M3,M4 : Integer);
var
   Regs                       : RegPack;
begin
   with Regs do begin
     AX := M1;     { Set up ax,bx,cx,dx for interrupt }
     BX := M2;
     CX := M3;
     DX := M4
   end;
   Intr(51,Regs);  { Trip interrupt 51 }
   with Regs do begin
     M1 := AX;
     M2 := BX;
     M3 := CX;
     M4 := DX
   end
end; { of proc Mouse }
```

```
begin { main body of program MouseSketch }
  M1  := 0;
  M2  := 0;
  M3  := 0;
  M4  := 0;
  HiRes;        { Choose graphics mode and color }
  HiResColor(Yellow);
  M1  := 0;       { Initialize mouse driver }
  Mouse(M1,M2,M3,M4);
  M1  := 1;       { Turn on Mouse cursor }
  Mouse(M1,M2,M3,M4);
  M1  := 3;
  OldX := 0;
  OldY := 0;
  while not KeyPressed do begin      { Exit mouse when any key pressed}
    Mouse(M1,M2,M3,M4);
    if M2 <> 0
      then Draw(OldX,OldY,M3,M4,1); { Draw if button pushed }
    OldX := M3;
    OldY := M4
  end
end. { of program MouseSketch }
```

20.11 FILLCHAR DEMO (FILLCHAR.PAS)

The **FillChar** routine is a handy means of intializing arrays and other data structures. This program gives an example of how it works and shows how it fills memory with a given value. It also shows how to examine (dump) a portion of memory from within a program.

```
program DemoFillChar;
{
  This program demonstrates how the built-in procedure FillChar works
}
var
  Seg1:          Integer;
  Ofs1:          Integer;
  Count:         Integer;
  OutWord:       Integer;
  Num:           Integer;
  Var1:          Integer;
  Value:         Char;
```

```
begin
  ClrScr;
  GotoXY(20,5);
  Write('enter Value for starting address');
  ReadLn(Var1);
  Seg1 := Seg(Var1);
  Ofs1 := Ofs(Var1);
  WriteLn;
  WriteLn('This variable is at segment: ',Seg1,' with an offset of: ',Ofs1);
  WriteLn;
  Write('Now put in a Value(single char) that you wish memory loaded with: ');
  ReadLn(Value);
  Write('Put in how many words you want filled: ');
  ReadLn(Num);
  FillChar(Var1,Num,Value);
  ClrScr;
  WriteLn('Now we print our memory starting with ',Seg1,':',Ofs1);
  for Count := 1 to Num do begin
    OutWord := Mem[Seg1:Ofs1];
    WriteLn(Seg1,':',Ofs1,' has value ',OutWord);
    Ofs1 := Ofs1 + 1;
  end
end. { of program DemoFillChar }
```

21. CP/M ROUTINES

Here are some program examples designed for advanced functions under CP/M. These routines are meant to serve as examples for advanced programmers; by their very nature, they can get you into trouble if you're not sure of what you're doing.

All of these programs can be found on your Turbo Tutor disk (assuming that you have the CP/M version). The file name of each program is given in the title for each section. These programs can all be compiled and executed immediately; no special hardware or software is assumed.

21.1 READ DIRECTORY (CPMDIR.PAS)

This program allows you to read the directory of the current logged drive. It makes the appropriate BDos calls, pulling the file names out of the memory location where CP/M leaves them.

```
program CPM80Dir;
  { This program will give a directory of the logged drive. }
const
  Search_First          : Integer = $11;
  Search_Next           : Integer = $12;
  Set_DMA               : Integer = $1A;
var
  Error, Loop, Start    : Integer;
  FCB                   : array[0..25] of Byte absolute $005C;
  DMA                   : array[0..255] of Byte;
```

```
begin
  Error := BDos(Set_DMA,Addr(DMA));
  FCB[0] := 0;
  for Loop := 1 to 11 do
    FCB[Loop] := ord('?');
  Error := BDos(Search_First,Addr(FCB));
  if Error <> 255 then begin
    Start := Error * 32;
    for Loop:= Start to start+8 do
      Write(Char(Mem[Addr(DMA)+Loop]));
    Write(' ');
    for Loop:= Start+9 to Start+11 do
      Write(Char(Mem[Addr(DMA)+Loop]));
    WriteLn
  end;
  repeat
    Error := BDos(search_Next);
    Start := Error * 32;
    if Error <> 255 then begin
      for Loop:= Start to start+8 do
        Write(Char(Mem[Addr(DMA)+Loop]));
      Write(' ');
      for Loop:= Start+9 to Start+11 do
        Write(Char(Mem[Addr(DMA)+Loop]));
      WriteLn
    end
  until Error=255
end. { of program CPM80Dir }
```

21.2 SYSTEM STATUS (CPMSTAT.DIR)

This program does a more exhaustive display of system status. It shows information about each disk drive, as well as the version of CP/M currently being used. This program is intended for use on CP/M versions 2.0 or higher.

```
program CPMstatus;
{
    Reads and displays CP/M status information
}
const
  CPMversion      =  12;
  CurDisk         =  25;
  AllocVector     =  27;
  DiskParam       =  31;
  GetUser         =  32;
```

```
type
  Word              = Integer;
  HexStr            = string[4];

  DPBREC =
    record
      SPT           : Integer; { SECTORS PER TRACK }
      BSH           : Byte;    { DATA ALLOCATION BLOCK SHIFT FACTOR }
      BLM           : Byte;
      EXM           : Byte;
(*    DSM           : Integer; { TOTAL STORAGE CAPACITY }   *)
      DSMlo         : Byte;
      DSMhi         : Byte;
      DRM           : Integer; { NO of DIRECTORY ENTRIES }
      AL0,AL1       : Byte;
      CKS           : Integer;
      OFF           : Integer
    end; { DPBREC }

var
  DPB               : ^DPBREC;
  RecsPrBlock       : Integer;
  RecsPrDrive       : Real;
  TrksPrDrive       : Real;

  BIOSaddr          : Integer absolute 1;
  BDOSaddr          : Integer absolute 6;

  TPA               : Real;
  Version           : Integer;
  Result            : Integer;

function Hex(Number: Integer;Bytes: Integer): HexStr;
const
  T                 : array[0..15] of Char = '0123456789ABCDEF';
var
  D                 : Integer;
  H                 : HexStr;
begin   H[0]:=Chr(Bytes+Bytes);
  for D:=Bytes+Bytes downto 1 do begin
    H[D]:=T[Number and 15];
    Number:=Number shr 4
  end;
  Hex:=H
end; { of proc Hex }
```

```
begin { main body of program CPMStatus }
  ClrScr;
  Writeln(
' —Logged—    ————Records————     —Tracks—    ——Capacity——         ——TPA—— ')
 ; writeln (
' Drive User    Block  Track  Drive     Sys.  Drive    Directory  Drive          Bytes    K');
{    x:   xxx      xxxxx xxxxx  xxxxx     xxx   xxxxx    xxxx/xxxx xxxxxK         xxxxx  xx.x }

  Write('    ', Chr(Bdos(CurDisk) + Ord('A')), ':     ',
      Bdos(GetUser,$FFFF):3, '    ');
  DPB:=Ptr(BdosHL(DiskParam));
  with DPB^ do begin
    RecsPrBlock:=BLM+1;
    Write(RecsPrBlock:5);
    Write(SPT:6);
    RecsPrDrive:=(DSMhi*256.0 + DSMlo + 1.0)*RecsPrBlock;
    Write(RecsPrDrive:6:0,'    ');
    Write(OFF:3);
    TrksPrDrive:=RecsPrDrive/SPT + OFF;
    if TrksPrDrive < > Trunc(TrksPrDrive)
      then TrksPrDrive:=TrksPrDrive+1;
    Write(Trunc(TrksPrDrive):7,'    ');
    Write(DRM+1:4,'/',CKS*4:4,Trunc(RecsPrDrive/8):6,'K');
  end;
  TPA:=2.0*(BDOSaddr shr 1) -$100;
  WriteLn(TPA:8:0,' ',TPA/1024:6:1);
  WriteLn;
  Writeln('- Operating System -');
  Writeln(' Version   BDOS   BIOS');
        {   xxxx  x.x xxxx    xxxx}
  Result:=BdosHL(CPMversion);
  Version:=Hi(Result);
  if Version = 0
    then Write('CP/M ')
  else if Version = 1
    then Write('MP/M ')
    else Write('???? ');
  Version:=Lo(Result);
  if Version = 0
    then Write('1.x')
    else Write(Version div $10,'.',Version mod $10);
  WriteLn(Hex(BDOSaddr, 2):6, Hex(BIOSaddr-3, 2):6);
end. { of program CPMStatus }
```

22. ASSEMBLY LANGUAGE PROGRAMMING

—Assembler—

There are usually three reasons for calling assembly language from a high-level language. The first is to execute some system operation; the second, to perform some sort of data manipulation; the third, to carry out some set of instructions more quickly. In all three cases, the action is performed in assembly language because of limitations in the high-level language. Ironically, these problems don't crop up all that often in TURBO Pascal. First, TURBO provides broad access to system functions and resources. You can make calls directly to the operating system (via BDos, BIOS, MSDos, Intr), you can directly access any location in memory, and you can reference all the I/O ports. Second, Turbo Pascal provides low-level operations (such as **shr** and **shl**) and also extends logical operations (**and**, **or**, etc.) to integer values. Retyping capabilities bypass the need for assembly language data conversion functions. Finally, TURBO Pascal produces machine code which is often fast enough to satisfy your needs.

22.1 IN-LINE CODE (INLINE.PAS)

The above statement is often true, but not always...so, yes, there may be time when you want to write assembly language routines. In that case, you have two routes to go. The first (and simplest) is to use **in-line machine code**. Turbo Pascal allows you to insert (anywhere!) machine code, which will then be executed everytime the **inline** statement is encountered.

The **inline** statement takes the form:

```
inline(vall/vall/.../valn);
```

where each value (val1, etc) is a constant (either literal or named, and of type integer), a variable identifier, or a location counter. More information can be found in the **TURBO Pascal Reference Manual**, in the appropriate appendices for your operating system.

Here's a sample program, written to run on an 8086-based system (under either MS-DOS or CP/M-86). The routine does a simple divide-by-two of the value passed to it; note how the parameter's name (a variable identifier) can be put right into the **inline** statement.

```
program InLineSample;
{
    The following program example divides even integers by two.
    For odd integers this program returns -32768 + the value
    divided by two (integer division)
}
var
  Value   : Integer;

procedure VInLine(var Value:Integer);
{
    a simple use of inline code. Note that some constants
    have been defined and used, while other values are left
    as literal hexadecimal constants. This is done just for
    illustration.
}
const
  CLC       = $F8;
  IN_CDI      = $47;
begin
  inline
    ($C4/$BE/VALUE/       { LES     DI,VALUE[BP] }
    CLC/                  { CLC     }
    $26/$D0/$1D/          { RCR     ES:BYTE PTR [DI] }
    IN_CDI/               { INC     DI       }
    $26/$D0/$1D);         { RCR     ES:BYTE PTR [DI] }
end; { of proc VInLine }

begin { main body of program InLineSample }
  ClrScr;
  repeat
    Write('Enter a number, <0> to quit:  ');
    ReadLn(Value);
    VInLine(Value);
    WriteLn('Return Value is:                ',Value);
  until Value = 0;
  ClrScr;
end. { of program InLineSample }
```

22.2 ASSEMBLY LANGUAGE ROUTINES (PASSFUNC.PAS,PASS.ASM)

TURBO Pascal allows you to call assembly language procedures and functions that you have assembled separately. These are known as **external** subprograms. If you are running under CP/M-80, you have to declare these routines to be at specific locations in memory:

```
    procedure LowToUp(var Str : BigStr); external $2E00;
```

When your program starts executing, it must go out and itself load the routine at the appropriate memory location, with code looking something like this:

```
procedure LoadRoutine;
var
  CodeFile        : File;
  Buffer          : array[0..1] of Byte absolute $2E00;
  Index,Rec       : Integer;
begin
  Assign(CodeFile,'LOWTOUP.COM');
  Reset(CodeFile);
  Index := 0; Rec := 0;
  { $R-}
  while not EOF(CodeFile) do begin
    BlockRead(CodeFile,Buffer[Index],Rec);
    Rec := Rec + 1;
    Index := Index + 128
  end;
  { $R+}
  Close(CodeFile)
end; { of proc LoadRoutine }
```

If instead you are using MS-DOS, then you must write a true assembly language routine which is assembled separately and then linked in at compile time. The advantage is that you can write directly in assembly language (rather than "hand-assembling" your routines into machine code for the **inline** statement). The disadvantages are that you have to own an assembler (and one that's compatible with TURBO Pascal), and that you have to make sure that your assembly routines and your Pascal code can correctly communicate with each other. Again, the particulars are specific to each operating system, so you are again referred to the appropriate appendices in the **TURBO Pascal Reference Manual**.

For those you you with MS-DOS systems, here is a sample assembly language routine:

```
; * * WARNING * WARNING * WARNING * WARNING * WARNING *
; Please do not try to use external functions
; unless you are familiar with assembly language.
;
; IMPORTANT: Externals must be written in assembly language.
;
; The following example addes two integer numbers.
;

code        segment
            assume      cs:code
pass        proc        near

            push        bp          ; SAVE ENVIRONMENT
            mov         bp,sp

            mov         ax,[bp+4]   ; GET PARAMETER 1
            add         ax,[bp+6]   ; GET PARAMETER 2
                                    ; GIVES THE RESULT
            mov         sp,bp       ; RESTORE ENVIRONMENT
            pop         bp
            ret         4

pass        endp
code        ends
            end

; Now exit to PC-DOS and type:
;    >ASM PASS
;    >LINK PASS
;    >EXE2BIN PASS.EXE PASS.COM
;
; Ignore minor errors from ASM and LINK.
```

Having carried out the steps above, you can now compile the following TURBO Pascal program (making sure that PASS.COM is on the default (logged) drive):

```
program PassFunc;
{
   This routine expects the assembly language routine 'Pass'
   to reside in the file 'PASS.COM'
}
var
  Var1, Var2, Var3: Integer;
```

```
function Pass(VarX, VarY: Integer): Integer; external 'PASS.COM';

begin { main body of program PassFunc }
  repeat
    ReadLn(Var1);
    if Var1< >0 then begin
      ReadLn(Var2);
      Var3 := Pass(Var1,Var2);
      WriteLn(Var1,' + ',Var2,' = ',Var3);
      WriteLn
    end
  until Var1 = 0
end. { of program PassFunc }
```

When this program compiles, TURBO Pascal will look for the file PASS.COM and link the machine code into the executable code being produced.

22.3 CONCLUSION

Well, this is the end of my little book. I have enjoyed telling you what I know about TURBO Pascal. I hope you've enjoyed learning about it. All of the tools are now in your hands. It is up to you to use those tools like any skilled craftsman would. Practice what you have learned until you master it, then move on to something else.

Since you have read the entire manual, I would like to reward you with a few bits of wit I thought you might enjoy:

1. If it were not for the last minute, nothing would ever get done.
2. The efficency of a programming team is inversely proportional to the number of members in that team, whenever that number is greater than three!
3. Even the simplest things are more complicated than you thought.
4. The first 90 percent of a job takes 90 percent of the available time; the last 10 percent takes the other 90 percent.
5. Always plan for the project to take twice as long as you expected, after first doubling your original estimates.

Thank you for your attention to the matters of programming in TURBO Pascal and for buying this book. If you are programming for your own entertainment, your newly-learned programming skills are probably sufficient for anything you will want to do. If you intend to write serious Pascal programs, there are a couple of other books I strongly recommend. These are:

—Wirth, Niklaus: *Algorithms + Data Structures = Programs*. Prentice Hall (1976).

—Wirth, Niklaus: *Pascal*. Prentice Hall.

—Knuth, Donald E.: *The Art of Computer Programming*. (Vols. 1,2, and 3). Addison Wesley (1973).

—Horowitz, E. et al: *Fundamentals of Data Structures*. Pitman (1976).

NOTES

INDEX

CATALOG OF ALL BORLAND PRODUCTS

BORLAND
INTERNATIONAL

NOTES

SuperKey

SUPERKEY TURNS 1,000 KEYSTROKES INTO 1 AND KEEPS YOUR 'CONFIDENTIAL FILES' . . . CONFIDENTIAL.

Borland International, the folks who brought you SIDEKICK™, now offer you a keyboard enhancer that can truly make your life easier and your files safer.

When you've got a lot to do, SUPERKEY™ lets you dramatically cut the number of times you touch the keyboard. Even your most sophisticated programs becomes one-stroke easy.

SUPERKEY also performs as a high-level security encryptor/decryptor to protect your sensitive files.

Just look at what SUPERKEY'S got!

- **HIGH-LEVEL RESIDENT ENCRYPTION: SUPERKEY** includes a two-mode encryption system that uses your secret password to encrypt and decrypt files —even while running other programs.

Mode 1: DIRECT-OVERWRITE ENCRYPTION: Just give SUPERKEY your password and the files of your choice will be encrypted or decrypted — *without* changing file size or creating backup files for even greater document security. You can even encrypt public bulletin board messages. No more curious reading of your files unless you provide the secret password.

Mode 2: EXE OR COM FILE ENCRYPTION: Using your personal code, SUPERKEY can encrypt your files for transmission (over a phone line, for instance). On the other end, only your secret code can reconstruct the file.

- **PRIVACY MODE: SUPERKEY** lets you turn off your screen and lock your keyboard against intruders.
- **SCREEN BURN-IN PROTECTION: SUPERKEY** will switch your monitor off if there's no input or screen I/O. As soon as a key is touched, the screen image returns.
- **RAM RESIDENCE AT ALL TIMES:** Create, edit, save and even recall new or existing macro files anytime, even while running another program.
- **PULL-DOWN MACRO EDITING:** A single keystroke opens a full-screen WordStar™-command edit window. You can perform most word processing functions including file naming. SUPERKEY even has our 'clip & tape' function for moving blocks of data from the screen for later retrieval.
- **COMMAND KEY REDEFINITION:** With a few easy strokes, you can redefine key commands 'on the fly'.
- **ENTRY AND FORMAT CONTROL IN DATA FIELDS: SUPERKEY** tests for digits or letters, converts lower case to upper case and visa-versa, and justifies files right, left, or center.
- **TOTAL PROKEY COMPATIBILITY: SUPERKEY** uses every ProKey™ macro file without any con version! No time is wasted and you're that far ahead.
- **DISPLAY-ONLY MACRO WINDOW:** Create your own customized on-line help system for other users. An ideal feature to set-up interactive tutorials on your computer.
- **CONTEXT-SENSITIVE ON-LINE HELP SYSTEM:** You can request just the help you want—no more wading through the whole operating manual.

SUPERKEY is now available for an unbelievable $69.95 (not copy-protected).

Minimum System Configuration: SUPERKEY is compatible with your IBM PC, XT, AT, jr. and 100% compatible microcomputers. You computer must have at least 128K RAM, one disk drive and PC-DOS 1.0 or greater.

BORLAND INTERNATIONAL

Available at Better Dealers Nationwide.
4585 Scotts Valley Drive
Scotts Valley, CA 95066

Telephone: (408) 438-840
Telex: 172373
Telecopier: (408) 438-869
For Credit Card Order Ca
(800) 255-8008,
(800) 742-1133 (in CA.)

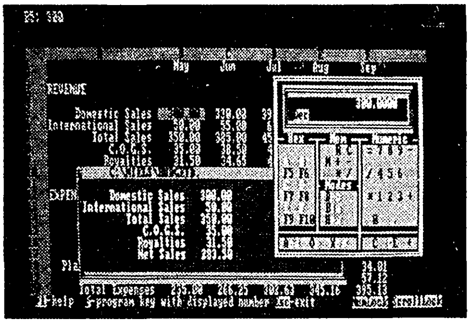
NOTES

SIDEKICK

INFOWORLD'S SOFTWARE PRODUCT OF THE YEAR

Just listen to what Dan Robinson has to say!

"SIDEKICK™ stands in the shadows behind whatever program you are using, ready to jump forward when you need it. The program's various functions use windows that overlay the display you are working with and restore the screen when you are through. The program contains a respectable word processor for note taking, a dialer that your smart modem can use with your phone list, a calculator for hexidecimal/binary/decimal arithmetic, an appointment calendar and a ASCII table for programmers . . . SIDEKICK IS A TIME-SAVING, WORK-SAVING, FRUSTRATION-SAVING BARGAIN." **–Dan Robinson of INFOWORLD**

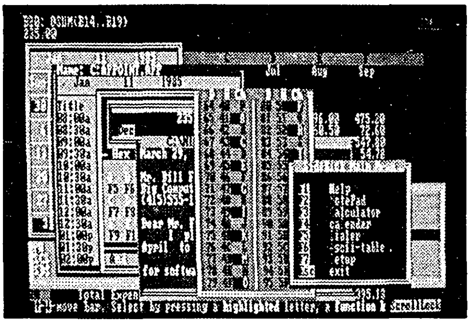


```
              May    Jun    Jul    Aug    Sep
REVENUE
      Domestic Sales         330.00
 International Sales   50.00  55.00
         Total Sales  350.00  385.00
            C.O.G.S.   35.00   38.50
           Royalties   31.50   34.65

       Domestic Sales   300.00
  International Sales    50.00
          Total Sales   350.00
             C.O.G.S.    35.00
            Royalties    31.50
            Net Sales   283.50

  Total Expenses   235.00  266.25  302.63  345.16  395.13
```

Here's SIDEKICK running over Lotus 1,2,3™. In the SIDEKICK Notepad you'll notice data that's been imported directly from the Lotus screen. In the upper right you can see the Calculator.



```
Help
NotePad
Calculator
Calendar
Dialer
Ascii-table
Setup
exit
```

All the SIDEKICK windows stacked up over Lotus 1,2,3. From bottom to top: SIDEKICK's "Menu Window", ASCII table, Notepad, Calculator, Appointment Scheduler and Calendar, and Phone Dialer. Whether you're running WordStar™, Lotus™, dBase™, or any other program, SIDEKICK puts all these desktop accessories instantly at your fingertips.

THE CRITIC'S CHOICE

— **Charles Petzold, PC MAGAZINE:** "In a simple, beautiful implementation of WordStar's™ block copy commands, SIDEKICK can transport all or any part of the display screen (even an area overlaid by the notepad display) to the notepad."

— **Garry Ray, PC WEEK:** "SIDEKICK deserves a place in every PC."

— **Jerry Pournelle, BYTE:** "If you use a PC, get SIDEKICK. You'll soon become dependent on it."

SIDEKICK is an unparalleled bargain at only $54.95 (copy-protected)*
$84.95 (not copy-protected)*

*Minimum System Configuration: SIDEKICK is available now for your IBM PC, XT, AT, jr. and 100% compatible microcomputers. The IBM PCjr. will only accept the SIDEKICK not copy-protected version. Your computer must have at least 128K RAM, one disk drive and PC-DOS 1.0 or greater.

BORLAND
INTERNATIONAL

Available at Better Dealers Nationwide.
4585 Scotts Valley Drive
Scotts Valley, CA 95066

Telephone: (408) 438-8400
Telex: 172373
Telecopier: (408) 438-8696
For Credit Card Order Call:
(800) 255-8008,
(800) 742-1133 (in CA.)

NOTES

TURBO PASCAL 3.0:

BUSTS THE COMPILER SPEED BARRIER!

Borland International, the creators of SIDEKICK™ and SUPERKEY™, now launch TURBO PASCAL™ 3.0—The world's fastest compiler!

Look at the Power:		MS Pascal	TURBO 3.0
	Compilation Speed:(*) (And twice as fast as TURBO 2.0!)	206 sec	A quick 8 sec
	Execution Speed:(*) (Resulting programs run faster. A real plus for TURBO PASCAL programs over other compiled programs.)	20 sec	A speedy 9 sec
	Code Size: (The same size as TURBO 2.0. How'd we do it?)	35K	A small 12K
	Compiler Size: (It hasn't changed from TURBO 2.0.)	300K+	A mere 35K

(*) Benchmark run on an IBM PC using MS Pascal version 3.2 and the DOS linker version 2.6. The 179 line program used is the "Gauss-Seidel" program out of Alan R. Miller's book, *Pascal Programs for Scientists and Engineers* (Sybex, page 128) with a 3-dimensiona non-singular matrix and a relaxation coefficient of 1.0.

TURBO PASCAL 2.0 had a lot of great features and they're still there in TURBO PASCAL 3.0:
- BUILT-IN INTERACTIVE EDITOR: WordStar™-easy editing lets you debug quickly.
- ONE-STEP COMPILE: No more 'hunting & fishing' expeditions. TURBO PASCAL does it all in one simple step.
- MICROCALC: A full-blown spreadsheet on your disk ready-to-run with complete annotated source code.

The best just gets better with a host of additional features:
- We rewrote the I/O system and we now support I/O redirection.
- For IBM PC versions we've added TURTLE GRAPHICS and full tree directory support.
- For all 16-bit versions, we now offer two options: 8087 math coprocessor support for intensive calculations and Binary Coded Decimals (BCD) for business applications.

Debugging is So Easy! When you run your program, TURBO 3.0 locates any error, takes you to it, and tells you what it is. Your program is up in record time.

Specifications:

Compile Speed	Approximately150 lines per second (IBM PC)	**Code size:**	Object code plus 9K for run time library(complete)
Number of passes:	1	**Max code size:**	64K
		Max data size:	64K
Compiled to:	Native machine code (.COM files [.CMD for CP/M-86])	**Max stack size:**	64K
		Max heap:	Unlimited

The Critics Unite:

Jeff Duntemann, PC MAGAZINE: "Language deal of the century . . . TURBO PASCAL. It introduces a new programming environmen and runs like magic."

Dave Garland, POPULAR COMPUTING: "Most Pascal compilers barely fit on a disk, but TURBO PASCAL packs an editor, compiler linker, and run-time library into just 29K bytes of RAM."

Jerry Pournelle, BYTE: "What I think the computer industry is headed for: well-documented, standard, plenty of good features, anc a reasonable price."

The unbelievable TURBO PASCAL 3.0 is available right away for $69.95 (not copy-protected)*.

* Options: TURBO PASCAL with 8087 or BCD at a low $109.90. TURBO PASCAL with both options (8087 and BCD) priced at $124.95.

Minimum System Configuration: TURBO PASCAL 3.0 is available today for your Z-80, 8088/86, 80186 or 80286 microprocessor computer running either CP/M-80 2.2 or greater, CP/M-86 1.1 or greater, MS-DOS 1.0 or greater or PC-DOS 1.0 or greater. Your computei must have at least 64K RAM and one disk drive. A XENIX version of TURBO PASCAL will soon be announced, and before the enc of 1985, TURBO PASCAL will be running on most 68000-based microcomputers.

Integrated full screen editor with WordStar-like functions. Supports BIOS/BDOS calls for CP/M-80 and 86. Supports Interrupt and MS-DOS function calls for MS DOS, PC-DOS. Inline machine code. External subroutines (in relocatable binary format). Direct access to CPU and data ports.

BORLAND INTERNATIONAL

Available at Better Dealers Nationwide.

4585 Scotts Valley Drive
Scotts Valley, CA 95066

Telephone: (408) 438-8400
Telex: 172373
Telecopier: (408) 438-8696
For Credit Card Order Call:
(800) 255-8008,
(800) 742-1133 (in CA.)

NOTES

NOTES

NOTES

NOTES