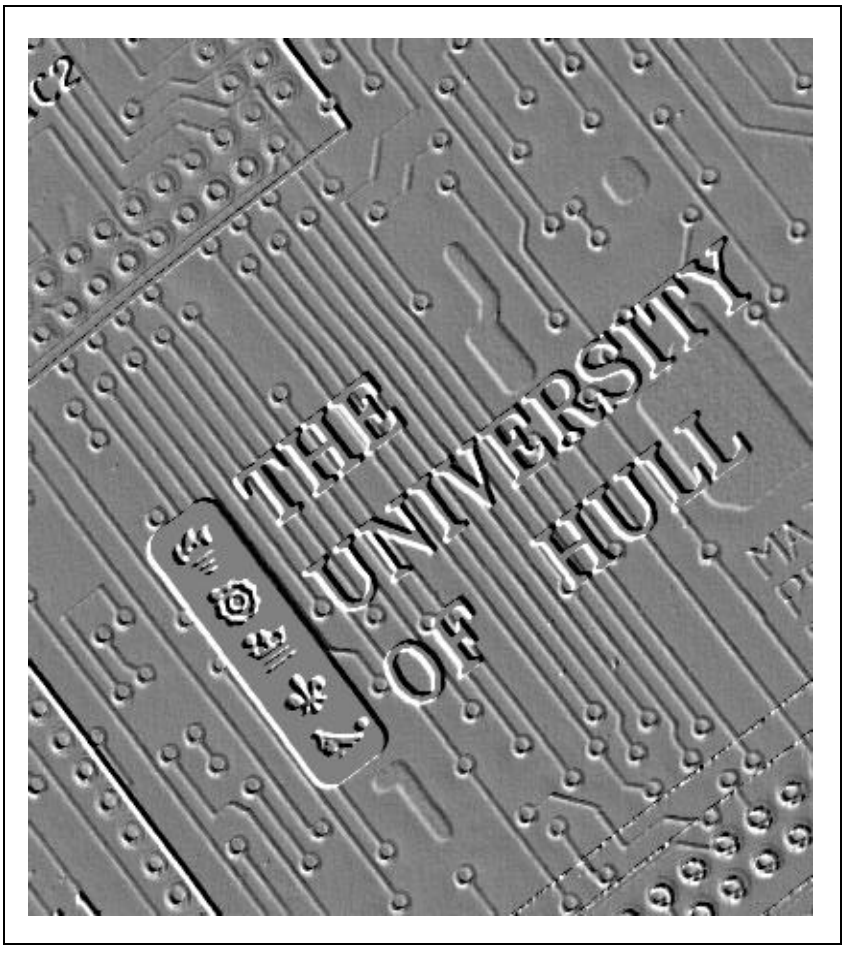
The University of Hull

An Introduction to PASCAL

Rob Miles



Department of Computer Science

Contents

 The author can be contacted at:

The Department of Computer Science

The University of Hull

HULL

HU6 7RX

r.s.miles@dcs.hull.ac.uk

Introduction to Pascal

10/01/2001 10:30

The IBM Personal Computer

Before we can start programming in PASCAL we are going to take a quick look at the IBM PC compatible, what it is and how you use the command prompt.

Hardware

The IBM Personal Computer (PC) is a microcomputer based upon the INTEL microprocessors series. It is the most popular microcomputer standard in the world.

When the IBM PC was launched it was made an "open architecture" machine in that IBM released comprehensive details of how it worked. This made it easier for other manufacturers to produce hardware and software to be used with it and was a major reason why the machine became so popular. Unfortunately for IBM this also means that it is comparatively easy for another manufacturer to duplicate the design and make what is known as an IBM "clone". In fact, these days machines actually made by IBM are the exception rather than the rule.

In fact, the PC is now best considered as just a vehicle for the software which it runs.

Software

These notes are going to focus on using a PC from the Windows operating system, but controlling programs from the MS-DOS prompt.

The MS-DOS Operating System

When IBM launched the PC they did not develop their own operating system, instead they made an agreement with Microsoft, a software house which writes exclusively for microcomputer systems, to license their MS-DOS operating system for use on the PC.

MS-DOS went through several releases after the PC was first launched. It is basically a single user operating system although a version is available

MS-DOS was originally very similar to the CP/M operating system popular on 8 bit microcomputers. It is the underlying platform on which successive versions of the Windows environment have been placed. Only with the Windows NT and Windows 2000 systems are Microsoft finally getting rid of all the MS-DOS code in their operating systems.

The MS-DOS Prompt in Windows

In the old days the MS-DOS prompt was all you had. It provided a set of commands which you used to perform housekeeping on your computer, run programs and the like. When your computer started up you were presented with the MS-DOS prompt and you did everything from there.

Nowadays you can use the Windows environment to control your computer, however, the MS-DOS prompt has been retained. In Windows NT and Windows 2000, although the MS-DOS operating system is no longer present, the commands that you use and the way that they work has been faithfully preserved in the command prompt program.

It is worth knowing a bit about the command prompt, since it makes it possible to do things which would be very difficult just using a mouse and the windows.

You start the command prompt by selecting it from the Programs menu from the Start button. In Windows 2000 the command prompt has been moved into the Accessories program group.

MS-DOS Commands

There are two types of MS-DOS commands, *internal* and *external*. Internal commands are part of the MS-DOS command prompt program. These tend to be the commands you use most often.

External are programs which are loaded and run each time you want to use them. You can use the command PATH to tell the MS-DOS command program where to look for an external commands, initially the PATH will point to your Windows system directory, where the external programs are stored.

MS-DOS commands are single words, for example FORMAT to format a disk, DEL to delete a file. Many commands have two forms, a long form and an abbreviation. Within this document we will give the long form followed by the abbreviation in brackets, for example CHDIR (CD). Most commands operate on something, for example the internal command TYPE displays the contents of a file. The item the command works on follows the command, separated with a single space, for example to type the contents off a file called FRED.TXT you would issue the command:

```
TYPE FRED.TXT
```

An item a command works on is called a parameter. Some commands need more than one parameter, for example the COPY command needs two - the input file and the output file.

You can change the way a command operates by adding additional information in the form of *switches*. Some commands have a range of switches available, for example the V switch for the copy command tells it to verify the copy once it is complete. Switches are sometimes called *options* .You specify a switch by preceding it with a / character. Switches and parameters are separated with spaces, for example to copy FRED.TXT to JIM.TXT and verify the copy you would use:

```
COPY FRED.TXT JIM.TXT /V
```

You issue MS-DOS commands by typing them at the keyboard and then pressing the ENTER key when you have finished the command and want MS-DOS to do something.

Some commands, for example TYPE, can be interrupted using CTRL+BREAK . This can be useful if you type what you think is a 4 line file and it turns out to contain 4,000 lines! To interrupt a command hold down the CTRL key and press the key marked BREAK which is located to the left of the key on the top right of the keyboard. Note that some commands cannot be interrupted in this way.

You can give commands in upper or lower case (CAPITALS or small letters).

The Default Drive

MS-DOS identifies each disk drive connected to a system by a particular letter. Whilst using MS-DOS one drive is nominated as the "default" drive, i.e. the place where MS-DOS will look for files and commands unless told otherwise. The default drive is usually displayed in the command prompt; When we start an MS-DOS command prompt we are usually using drive C and so the prompt "C:\" is displayed. You can change the default drive by typing the name of the new default, followed by a ":", for example:

```
D:
```

- would change the default drive to drive D.

Drives A and B are usually floppy disk drives, drive C is a hard disk and drive D is usually your CD-ROM drive. If you have additional hard disks you may find that drive D is also a hard disk, and your CD-ROM ends up as drive E.

You can have up to 26 drive letters, one for each letter of the alphabet. Sometime a drive letter can be assigned to a network connection, for example drive G: may actually be stored on a server connected to your computer.

Formatting a Floppy Disk

If you want to take files off a machine you can put them onto a floppy disk. Before Windows and MS-DOS can use a floppy disk a structure must be set up on the disk into which files can be placed. This procedure, known as formatting, is usually performed once for a disk, since formatting removes all information on it and sets up an empty structure. It is accomplished by the command FORMAT, which is followed by the name of the drive you wish to format.

Note that if the FORMAT command is not followed by a drive name it will format the default drive, which could be embarrassing if this is your system disk! If you try to do this you will get a warning, which you should take very seriously!

To format a blank disk in drive A simply type:

FORMAT A:

- and press ENTER.

Files and Directories

Files are identified within MS-DOS by name. Whilst Windows lets you have files with very long names, we are going to stick with the "traditional" MS-DOS file name conventions for now.

A filename is a string of up to 8 characters, optionally followed by an extension which is a 3 letter key to identify the files' contents. Examples are .PAS for a PASCAL program file and the filename by the "." character. There are rules about what constitute valid filenames, but if you just stick to letters in your filenames and meaningful extensions you will not go far wrong.

MS-DOS marks the end of the file with the special character CTRL-Z, which you can type on the keyboard by holding down CTRL and pressing Z. Normally this character will be inserted for you by the program you use to create the file but it can be useful when using the console device (see later).

MS-DOS supports a hierarchical filing system very similar to that available within UNIX. Files are organised within directories, a directory being a special kind of file which contains the names of the files in it and pointers to where the files are stored on the particular device. This means that directories can contain files which are directories and so on, allowing the build up of various levels of the filestore.

This facility is very useful, it allows you to easily group together related files, for example you may hold all your PASCAL programs in a directory called PASCAL, all your documents in a directory called DOCS and so on.

An empty disk has no directory structure at all, initially all files are stored at the "top level" of the disk which is known as the root directory. You impose the structure on the disk by creating your own directories using the MKDIR (MD) command. This command is followed by the name of the directory you wish to create, for example to create a directory called DOCS you could use the command:

MD DOCS

The directory is created at the current level, if I wanted to create a directory called LETTERS within the directory DOCS I would issue the MKDIR command from within DOCS.

The CHDIR (CD) command moves me to a directory specified by the parameter, which is a PATH. A path is a route to a directory, it can be relative to where you currently are, or it can be an absolute reference to a particular directory or file, for example:

CHDIR LETTERS

- would take me to the directory LETTERS beneath the current level (if such a directory exists). However:

CHDIR \DOCS\LETTERS\HOME

- would take me to the directory HOME which was contained in the directory LETTERS which was within the directory DOCS. Particular directories in a path are separated with the "\" character. The first " \ " in a path indicates the root directory, for example you can get back to the root directory from anywhere by simply typing:

**CD **

The string " . " means the current directory, ".." means the directory above the current one.

You can place a path in front of a filename at any time, to identify a file in a particular place, for example:

TYPE \DOCS\LETTERS\HOME\MUM.TXT

Unless you specify otherwise paths are searched for on the default device, however you can precede a path with a device identification if you wish, for example:

A:\progs\pascal\prog1.pas

- refers to a particular file on the disk in drive A:.

You can use the command DIR to find out the contents of a directory, it can be followed by a path which identifies the directory to be displayed, otherwise the current directory is shown.

Your Current Directory

The command prompt will tell you the current drive and the directory where you are presently located:

d:\notes\pascal\lesson1>

This indicates that I am using drive D, and that I am "in" the directory lesson1, which is held in the directory pascal, which is held in the directory notes.

Long Filenames

You can use filenames longer than 8 characters if you wish, although some old MS-DOS programs will not like this. You can even do stupid things like put spaces into filesnames (although I do not advise it). If you want to enter a filename which contains spaces you must enclose the filename in double quote characters:

type "a silly file which contains spaces in the name.txt"

- would type the filename given.

Devices

As well as named files, MS-DOS recognises a number of device names. These can be used within commands to identify a particular device where data is to be sent or got from. Some MS-DOS devices are:

COM1	the serial port
PRN	the printer (if connected)
CON	the console (i.e. the keyboard or the screen)

You can use device names where you would normally use a filename, for example consider:

COPY CON CON

- you may find the above notes on end of file markers useful at this point!. MS-DOS also supports the routing of output to specified devices. I will leave this as an exercise for you to find out about.

Exercises

You should be able to complete the following with reference to these notes

Turn on your system, boot it and start an MS-DOS command prompt.

Insert a blank floppy disk into drive A: and format it.

Create a directory on your new disk (in drive A) called NOTES.

Use the command "COPY CON a:\notes.txt" to create a file on your disk called notes.txt

Copy the file NOTES.TXT into a file called PAGE1.TXT in your new directory, specifying the full path of the output file and verifying the copy.

Describe the difference between the paths:

NOTES\LETTERS\LETT1.TXT

\NOTES\LETTERS\LETT1.TXT

The PATH command

MS-DOS allows you to tell it where to look for all external commands by use of the PATH command.

PATH is followed by a number of different paths, separated by semicolons. MS-DOS will search these paths when a command is issued which is not an internal one. Only if none of the paths contains a matching command will an error be produced.

Note that MS-DOS does not distinguish between built in commands, programs and batch files when looking for an external command - this makes it very easy to effectively add extra commands to MS-DOS. Note that the paths are searched in the order they are given in the PATH command, so you are advised to put directories containing the most popular commands early in the list.

Wildcard Characters

When you specify a filename MS-DOS allows you to include " wildcards " in the filename. A wildcard replaces a character or string of characters and allows you to select a range of files with a single command:

?	a question mark matches an single character in a filename, for example FR?D would match FRED, FROD or FRAD but would not match FRIED.
*	an asterisk matches an character at that position and any subsequent characters, for example FR* would match any name beginning with FR. An asterisk on its own matches any number of characters, for example *.BAT would match all the batch files in a directory.

You can use the wildcards in filenames and they will be matched in the current directory only.

Deleting Files and Directories

MS-DOS provides commands to delete files and directories, these are: DEL - delete the filename specified and RMDIR (RD) - remove the directory specified. Note that MS-DOS will refuse to remove a directory which is not empty, in order to avoid leaving files lying around with no pointers to them.

The parameters to RMDIR and DEL can contain wildcard characters but you are advised to be careful - there is no way to bring a deleted file back again.

It is possible to protect precious files against accidental erasure on a disk by sticking a tag across the Write Protect notch on one side. This protects all the files on a disk.

Pausing the Output

Sometimes an MS-DOS command may produce output faster than you can view it on the screen. You can get MS-DOS to pause when displaying to the screen by using the character CTRL+S, i.e. hold down the key marked CTRL and press the S key. To restart the output use CTRL+S again. Many commands, for example DIR, have switches which cause them to pause at the end of each screen full of output and wait for you to press any key to continue.

Programming in PASCAL

Now that we know a bit about driving MS-DOS we can start to think about writing our own programs.

Programs and Data

Computers work on information in a manner roughly similar to the way sausage machines work on sausage meat; something is fed in one end and, after processing, something else comes out the other end. In the case of the sausage machine what it does with the sausage meat is always the same, in the case of the computer different things happen to what goes in depending on what you tell the computer to do with it.

This does not mean that everything the computer does with the information is correct, if you give the computer the wrong instructions it will still try and do something and maybe produce some output, in the same way that if you got the design of your sausage machine wrong you might still get something out of the end. Neither does it mean that a computer will refuse to do something with incorrect information, in the same way that a sausage machine would try to make sausages out of a bicycle if one was placed in the feed hopper!

When we talk about data, we are talking about the information which the computer is working on. All computer systems take in data in some way and then produce data as output. A program is the sequence of instructions which you give the computer telling it what to do with the data. To change what happens to the data you just have to change the program.

Because computers are machines which operate essentially on numbers a computer program usually contains a lot of numeric manipulation. However the art of telling the computer what to do does not require a great deal of numeric ability, any more than you need to known how to strip down a gearbox in order to drive a Ford Mondeo.

A program is obeyed one step at the time by a computer system, starting at the first instruction and working on until the end of the program is reached. Some constructions enable the direction of the execution of the program to change, depending on the data which is being processed.

PASCAL and Programs

PASCAL was developed as a teaching language by Professor Niklaus Wirth at the Eidgenossische Technische Hochschule in Zurich. It has however become popular as a mainstream programming language, and is now available on most computer systems. In common with other programming languages, its grammar and syntax are rigidly defined and a PASCAL compiler will reject any program not adhering to the standard.

Structured Programs

PASCAL is a structured language, in that it allows you to build up a program as a series of separate parts which fitted together to construct a solution to the problem in hand. This leads to a programming approach in which you divide the problem to be solved into a number of sub-tasks and then write a series of smaller "programs" to solve each of them. You may find at this point that each subprogram can be broken down into a number of even smaller subprograms and so on.

This form of programming methodology (!) is called Top Down and leads to a more manageable way of solving problems, for example once the overall structure of the complete system has been decided upon different people could work on each of the subprograms. It also means that you can build up a "library" of useful subprograms which can be incorporated into later programs.

PASCAL also allows you to build structures to hold the data you are processing with your programs. Early programming languages only provided mechanisms for storing very simple forms of data, of mainly numeric form. PASCAL allows you to build up and manipulate a structure which holds items of data specific to your application, of which more later.

A PASCAL program is broken up into several segments, broadly speaking first you define the form of the data you are going to use and then you write the program which works on it: for example:

```
program simple (input, output);
var
i : integer ;
begin
      Readln (i) ;
      i := i * 2 ;
      Writeln (i) ;
 end .
```

This is a complete PASCAL program. There are no prizes for guessing what it does! Note that some of the words are printed in **bold**. These are reserved words, i.e. they are used by PASCAL to mean something. When you type your program you do not need to mark these words as anything special as they will be recognised automatically. They have been printed in bold in the above for clarity only.

PASCAL recognises particular reserved words in certain positions in the program, according to the syntax of the language. You have to be careful when writing your program that you do not use reserved words in the wrong context, for example consider the implications of a program with the name begin!

The Program Heading

The program heading gives the program a name and tells the PASCAL system what communication it will have with the world outside. You can pick whatever name you want, obviously you will choose one which reflects what the program does. The communication information in the example indicates that this program will take input from the surroundings and will also produce some output.

When we look at Turbo PASCAL we will find that this portion of the heading is less important as it is more efficient to deal with MS-DOS directly when we wish to transfer information.

The var Segment

The reserved word var marks the start of the segment which defines all the variables which will be used within the program. A variable is a place where PASCAL will put a piece of information we are interested in. This information may be changed as the program runs, depending on the information which is being processed at the time. We do not need to worry how the variable is stored, PASCAL looks after all that.

In this example we have one variable, in this case it has the name i. When writing your own programs it is best to choose variable names which reflect the purpose of the information stored in them, for example count, NoOfCars etc have much more meaning than X1, Y132 etc. Within PASCAL the case of letters, i.e. whether they are capitalised or not, is irrelevant; which means that NOOFCARS,

noofcars and NoOfCars are regarded as identical. You can use this facility to good advantage when specifying variable names.

Variable Names

The name of a variable can contain digits, letters or the "_' character and it must begin with a letter, i.e. X28 is a valid variable but 28X and NoOf\ are not. There are no penalties for choosing long and meaningful variable names, in fact this is one of the first steps to writing "useful" programs.

Variables can contain different types of information, depending on what your program needs. In the above example the variable i has the type integer which is one of the standard types which PASCAL provides. The integer type contains the set of numbers with no fractional part. Later on we will look at ways of designing our own types to store particular information. The process of telling PASCAL about a variable is called the declaration of the variable. When a variable is declared its type must also be indicated, that is the purpose of the ": integer " after the name i above.

Real Data

Another type you may find useful when writing programs for the exercises is real. The real type contains the set of all numbers. It is sometimes called *floating point*.

Note that the nature of computers means that the range of integers and reals will be limited and the reals will represent quantities to a finite accuracy. This should not affect any of the programs you will write during the course, but may become significant when you start writing larger programs.

The Instructions Segment

The instructions segment tells PASCAL what to do with the data structures. It starts with the reserved word begin. The first thing that the program does is call Readln. Readln is a standard procedure which is built into PASCAL. Note that Readln is not a reserved word, it simply shows that the writers of the PASCAL system have recognised that users need a way of getting information into and out of programs. Instead of building this into the PASCAL standard a number of procedures have been defined which do the donkey work.

A procedure is a piece of program which has been given a name and may be referred to within a program. Readln takes a variable name and goes and fetches the information which it requires from the input device, in the case of Turbo PASCAL this will be the keyboard. How Readln actually gets the information will vary from machine to machine, Turbo PASCAL makes a call to MS-DOS. When a Turbo PASCAL program is waiting for some input the cursor will appear and the system will wait until you type the information and, in the case of Readln, press the ENTER key.

The next line of the program performs an assignment to the variable i, i.e. a new value is assigned to the contents of i. The string ":= " separates the name of the variable to be assigned from an expression. You can regard ":=" as "becomes equal to". In this case the expression is very simple, we will look at expressions in more detail later.

The final line of the program uses the standard procedure Writeln to produce some output; in much the same way as Readln was called to obtain input.

The " end ." marks the end of the program instructions.

The Borland PASCAL Environment

The implementation of PASCAL which will be used for these notes was written by a company called Borland. It provides a complete "environment" in which to write, compile and run PASCAL programs. The implementation of PASCAL is quite close to standard PASCAL, although a number of extensions have been provided to make life easier when working on the IBM PC.

It is perhaps not the best way to develop very large scale programs on the IBM, mainly due to limitations on the maximum size of a program, although it does allow programs of unlimited size to be assembled from a number of modules and some enormous programs have been written using PASCAL.

From the point of view of learning PASCAL this version is very good because it takes very little time to correct mistakes and re-compile a program you are working on.

A version of PASCAL, called DELPHI has been produced by Borland to allow you to write programs which work within the Windows operating system.

The Borland PASCAL Environment

Borland PASCAL provides an environment within which you develop your programs. Your program source, the compiler, editor and the runnable version of your program are all held in memory at the same time.

Different versions of the language are started in different ways, and the environments may be slightly different along with the ways in which the programs are started. You should find out how to start your version.

Once you start it up, until you explicitly exit, all commands you issue will be acted on by TURBO PASCAL, not MS-DOS or Windows.

Borland PASCAL is a menu driven, windowed system. Along the top of the display you will see the options FILE, EDIT, RUN, COMPILE and OPTIONS. You select a menu by holding down the key marked ALT and pressing the first letter of the option you want. This may cause a menu to appear, from which you select the required command.

Your Work File

When you are working in TURBO PASCAL you will always be manipulating a program file of some kind. A program file just contains the text of your program. The TURBO system converts it into a runnable form and then executes it. Because you have not chosen a filename, the initial name is NONAME.PAS, however you will be asked if you want to give another name when you save the file. If you do not supply a language extension Turbo PASCAL adds the extension .PAS on automatically, otherwise it uses the one you give.

The Edit Command

Pressing ALT+E selects the Turbo PASCAL editor and moves you into your program text so that you can begin making changes to it. The editor is a screen based one very similar to a word processing program called WORDSTAR, in fact a great number of commands are identical. If you have used WORDSTAR you will find that all the cursor movement control characters work in exactly the same way, however in addition some of the IBM PC keys can be used instead.

The very top line of the display shows you the position you are within the program you are editing in terms of line and column position, it also shows the name of the work file you are editing and what modes, if any, are selected.

The editor works in either insert or overwrite mode. If you are in overwrite mode whatever you type is placed on top of the character underneath the cursor. If you are in insert mode the line is moved to the right to make room for the additional character. You can change between insert and overwrite mode by pressing the key marked Ins.

You can use the arrow keys on the right hand side of the keyboard to move the cursor about the screen, the PgUp and PgDn move the cursor up and down a screen at a time, holding down Ctrl and pressing PgUp or PgDn moves you to the top or bottom of your file respectively.

The Home key moves the cursor to the beginning of a line of text, the End key moves the cursor to the end of the line.

The left arrow key on the top right of the main keyboard area deletes the character to the left of the cursor and moves the cursor one space to the left.

When you enter the editor it is in "auto indent" mode. This means that when you press Enter to mark the end of a line and move down to the next line the editor will also move the cursor across to line it up

with the first character of the line above. This is very useful when writing heavily nested programs, however sometimes it can become a pain. You toggle auto indent mode on and off by pressing CTRL+Q and then I.

The editor also provides commands allowing you to move blocks of text around and repeatedly change strings of text.

To leave the editor hold down ALT and press the key to identify the option you want to move into.

The Compile Command

Compilation is the process whereby your program text is converted by Borland PASCAL into a sequence of instructions that the IBM PC can understand. If your program does not conform to the PASCAL standard the compiler will detect this and produce an error which you will have to correct.

You start the compilation process by pressing ALT+C . Borland PASCAL will show a menu, from which you select the Compile option. You now should see a window containing a rapidly moving counter showing the progress through your program. If an error is detected Borland PASCAL will stop at that point, display what it thinks is wrong and move you into the editor at the place the error was detected so that you can correct it. Remember however that this is not always where the problem is, although the error message given should be some help.

The Run Command

Having successfully compiled your program you can then run it with the ALT+R command. After you have pressed R Turbo PASCAL steps to one side and your program takes over until it reaches the end. Once your program completes you must then press any key to be put back into the main menu. Turbo PASCAL will not allow you to run a program if it has not compiled correctly, and if you have not compiled the file the Turbo PASCAL will compile it before running.

You can interrupt a running program with CTRL+BREAK, although you may have to press this several times to attract the attention of Turbo PASCAL. You can also page output from a Turbo PASCAL program using CTRL+S as from MS-DOS.

A word of warning; surprising though this may sound sometimes your program may go insane and effectively "break" the Turbo PASCAL system. If this happens you will have to reset the computer and start again. This will mean clearing out the memory and your program will be totally lost. In order to safeguard your work it is best to issue a Save command just before you run the program. Note also that if you are doing something highly fiddly with the system you may cause MS-DOS to have a breakdown and write all over any disks connected at the time, which may make them impossible to use. It is very reassuring to have a copy of your work on a disk in its envelope, i.e. not loaded in the machine, before you run the program.

The Quit Command

The Quit command, ALT+X takes you out of the Turbo PASCAL environment. If you have not already done so Turbo PASCAL asks you if you want to save the workfile.

Exercises

Type in the example program above and run it. Experiment with the input of real values and the effects of invalid number formats.

Modify the program so that it performs an illegal operation, for example tries to divide one by zero. Run it and see what happens.

Write a program which reads two real numbers and writes out their sum, difference and product. Investigate within the Turbo PASCAL help system to see how to write strings of text as well as the contents of variables.

Investigate the effects of compiler error messages (assuming you have not produced any already!).

The PASCAL Language

Variable Declaration within PASCAL

Variables are defined within the var segment of the program. All variables must be declared before they are used. If, during a program, you refer to a variable which does not exist - perhaps because you misspell it - the PASCAL compiler will detect this and flag an error.

A variable is declared by specifying the name of the variable or a list of variable names separated by commas; followed by a full colon and then the name of the type that the variable or variables will have:

```
var
fred, jim, ethel, nigel : integer ;
simon : real ;
i, j, k : integer ;
```

The ";" at the end of each declaration marks the end of it, and separates it from the next declaration (if any).

Note that you do not have to declare all the variables of a particular type in the same declaration statement.

Standard Data Types Within PASCAL

Until now we have considered only two types of date, integer and real. You will however want to deal with data which is not numeric, and you may also have structures of information which you will wish to manipulate.

A variable within a PASCAL program is assigned a type, which determines what can be stored in it, i.e. if a variable name identifies a particular box in memory the type gives the kind of item that the box can hold.

When operating on variables with different types it is important to take care when moving information from one type to another, for example it is meaningful to move the contents of an integer variable into a real one, since the set of real numbers also contains all integers but it is not reasonable to move data the other way. We will see later how to move data between types in a useful way.

PASCAL contains a number of standard types which are integer, real, boolean, and char. We have already met the first two.

Boolean Type

The type boolean can hold a value which is True or False. You would use such a variable in a program when you wished to hold a value which could have one of two possible values, typical variable names might be; EndOfFileHit, ProgramAborted, StandingOnMyFoot etc.

Boolean variables can be combined using the standard logical operators to produce logical expressions which return a boolean result, e.g.

```
(EndOfFile AND FinishedProcessing)
```

You can use such constructions to take decisions on the basis of a number of different factors.

Character Type

The type char holds a single text character, i.e. the letter which would be represented by a single keypress on the keyboard or by a single letter on the screen.

Borland PASCAL Types

TURBO PASCAL makes available extra data types. Note that if you wish to write programs to run on other PASCAL implementations you must not use these types, but if you are lucky enough to be working in TURBO PASCAL they are very useful. The TURBO PASCAL manual gives details of how much memory each type of variable will take up.

String Type

The first additional type is string. A string holds a number of characters, which varies according to the length of the string you want to store. You set an upper limit on the size of the string when you declare it in the var segment of your program. An error is given if when your program is run it tries to exceed the declared length. The length of a string is defined when it is declared:

```
LongLine : string [80] ;
ShortName : string [5] ;
```

When a string variable is defined an area of memory is set aside which is able to hold a string of the length specified in the square brackets. TURBO PASCAL also keeps a note of the length of the string, which can therefore be shorter than its maximum length.

This additional type is very useful, particularly as TURBO PASCAL also provides a number of standard procedures to allow sub-strings to be extracted and strings to be combined etc. You can use string variables in any situation where a piece of text is to be stored. The maximum length which a string can have is 255 characters.

The Byte Type

The byte type is used to store integers in the range 0 to 255. It is provided by TURBO PASCAL so that you can write programs which have smaller data areas. A byte takes up half the space in memory that an Integer does, and so is useful for storing values which you know will not be very large.

The Word Type

The word type is used to store integers in the range 0 to 32768. It is very similar to the integer type, except that negative values are not available.

Data Within the Program Text

Your PASCAL program will contain reserved words, variable names and data which you assign to variables in expressions.

Reserved words are simply stated and PASCAL will recognise them. When PASCAL examines your program during the compilation phase it sets up a list of all the variable names you have defined in the var segment and checks all the variables used against this list, flagging any it does not recognise. It also checks for type consistency within the program e.g. an attempt to assign a real value to a character variable would be rejected.

Data within the program is also checked to ensure that it conforms to the correct type for the context.

Within a PASCAL programs you represent numbers as strings of digits, with an optional decimal point and fractional part if they are of type real. Strings and characters are delimited by the ' character (I will leave how to get a ' character into a string as an exercise for you to find out). Boolean values within the program are either "True" or "False", for example the following entirely useless program is however entirely legal:

```
program NotALott (input, output) ;
 var
     Count, Items : integer ;
     Size : real ;
     Key : char ;
     Name : string [6] ;
```

```
    InputLine : string [80] ;
    IdCode : byte ;
    EndHit : boolean ;
begin
    Count := 0 ;
    Items := Count + 1 ;
    Name := `Fred' ;
    Readln (InputLine) ;
    Size := 1.232 ;
    IdCode := 99 ;
    EndHit := (IdCode = 99) ;
end .
```

However the following contains a number of errors - see how many you can spot!

```
program ThisIsIt (input, output)  ;
var
    i, j, k : integer ;
    Width : real ;
    Title : string [4] ;
    WidthMessage : string [5] ;
    KeyPress : char ;
    EndHit : boolean ;
begin
    i := i + 1 ;
    Readln (KeyPress) ;
    Width := KeyPress * i ;
    i := width/i ;
    title := `Professor' ;
    Widthmessage := Width ;
    Width := 99 ;
    EndHit := (Width=99) ;
end .
```

TURBO PASCAL supports additional ways of writing data: A number which begins with a $ character is regarded as a hexadecimal value. If you do not know what hex is don't worry - you will when you need to!

It is also possible to express real numbers in a exponential format, I will leave you to find out how!

When assigning a value to a character variable you can use # followed by a value in the range 0 to 255 to denote the character with that ASCII code. You can also represent control characters (i.e. ones which are not displayed but have a non-printing effect such as clear the screen or move the cursor) by use of a ^ character followed by the character in question, You can string together a number of characters represented in this way, producing a string of control characters. Useful control characters are:

^M return the cursor to the left hand margin
^J move the cursor down one line
^L clear the screen

Note that these character representations are not enclosed in ' characters, i.e. ^M means return but `^M' means the character ^ followed by the character M.

Comments

Within the program text you will find it useful to leave some notes which are purely for your own use, i.e. the do not form part of the program text and are ignored when the program is compiled. You do this in PASCAL by enclosing the comment in the characters { and } or the strings (* and *). e.g.

```
start := 0 ;     { set the initial val ue to 0 }
```

When the compiler sees { or (* it ignores any further text until it encounters a matching } or *). Please make copious use of the comment facility to keep yourself informed of what the program is doing at any particular stage. You will find this information invaluable if you ever have to come back to a program after an interval. Another use for comments is to keep track of version numbers, authors and changes to software that you write:

```
program editor (input, output ) ;
{     Text editing program }
{     written by Rob Miles, University of Hull, 20/7/00 }
{     Version 1.11 }
{ Updates: }
{     30/7/00   removed end of file bug }
{     1/8/01    added print option }
```

Assignments

We mentioned assignment statements briefly above, they provide a way of altering the contents of a variable by assigning a new value to it, i.e.

```
variable := new value ;
```

The new value can be a constant, i.e. a value which will never change from one run to the next, another variable or an expression. An expression is composed of operators which work on operands. When an assignment is performed the expression on the right hand side of the ":=" is evaluated and then placed in the variable specified on the left, i.e.

```
x := 1 ;
x := x + 1 ;
```

- would result in x containing the value 2. Assignments must be consistent, in that the variable on the left of the ":=" must be of the same type as the expression on the right.

Operands

Operands are the items which are combined using operators within expressions. An operand may be either a constant of the appropriate type, given as shown above, or the name of a variable, in which case the contents of the variable will be used at that point.

Operators

The range of operators which is available varies from type to type, for example the + operator cannot be used between operands of type boolean.

Real Operators

When considering real data the standard range of numeric operators is available, and they work with the same precedence as in mathematics, e.g. within an expression all the operations involving multiplication will be performed before all additions. Again, as in mathematics it is possible to enforce a particular order of evaluation by the use of parenthesis. Note however that unlike mathematics implied operations are not supported, e.g. if you put xy when you mean x*y PASCAL will look for a variable called xy and complain when it does not find it. The symbols for arithmetic operations are as follows:

Operation	Symbol
multiplication	*
division	/
addition	+
subtraction	-

Note the use of " * " instead of the more mathematical "x". It is also possible to use the "-" character before a constant to represent a negative value, in this special case the negation is performed before anything else.

Integer Operators

Integers share a similar range of operators to reals, although care should be taken because the "/" operator may produce a real result. In addition to "/" the two operators " DIV " and " MOD " are provided. The first performs an integer division, removing any fractional part of the result. The second returns the remainder of such a division:

```
program vtest (input, output) ;
 var
      i, j, k : integer ;
      x : real ;
 begin
      i := 2 ;
      j := 3 ;
      x := i/j ;
      Writeln ( x ) ;
      k := i MOD j ;
      Writeln ( k ) ;
      k := i DIV j ;
      Writeln ( k ) ;
 end .
```

The program shown would print out .6666666, 2 and 0. Note that in the above program the line:

```
k := i/j ;
```

will not be compiled as PASCAL does not allow you to try to put a real value into an integer variable.

Logical Operators

The three logical operators are AND, OR and NOT . The operator NOT is applied first, followed by AND and then NOT - again the order of application can be modified by the use of parenthesis.

Character Operators

There are no operators which work on the character type. A character assignment statement will therefore only assign a character constant or another character variable.

String Operators

TURBO PASCAL allows the addition operator to be applied to strings, for example:

```
var
      FullName : string [20] ;
 begin
      FullName := 'Rob'+' '+'Miles' ;
 end .
```

Standard Procedures

We have already looked at two standard procedures in our first programming example: Readln and Writeln . These enable communication with the outside world from within a program. PASCAL contains a number of standard procedures, TURBO PASCAL several more. Some of the procedures work on parameters, which are broadly similar to the parameters used for MS-DOS comands, with the difference that sometimes the parameter gives the name of a variable which is to be changed by the procedure.

We will look at further standard procedures as we go on, in order to complete the exercises you will have to investigate those provided for manipulating strings.

Defining Your Own Types - the type segment

PASCAL will also allow you to design your own types of data which can be used within your program. The advantages of this facility are twofold, firstly the program is made more readable because all assignments etc. refer to exactly what the variable is supposed to contain and secondly the program is made more robust, in that an attempt to put an invalid value into a variable will be detected by the compiler.

Types are defined in the type segment, which comes before var. For each new type all possible values it can have are given, for example:

```
type
    DaysOfWeek = (Mon, Tue, Wed,  Thu, Fri, Sat, Sun) ;
 var
    CurrentDay : DaysOfWeek ;
```

The above program snippet defines a new type, called DaysOfWeek. This can have any of the supplied values, meaning that your program could contain:

```
CurrentDay := Fri ;
```

It is also possible to define a type in terms of a subrange of a type which already exists, for example:

```
type
    DaysOfWeek = (Mon, Tue, Wed, Thu, Fri, Sat, Sun) ;
    WeekDay = Mon .. Fri ;
```

Note the use of ".." to mean all the ones between the two end points. The order in which the values a type can have is significant, i.e. LongWeekend = Fri .. Mon would not be permissible. It is also possible to have subranges of the integer type, e.g.

```
LowValues = 0 .. 10 ;
```

PASCAL provides the standard functions pred and succ which can operate on variables of a particular type. Pred returns the value preceding the one the variable currently has, Succ returns the value following the current variable, for example

```
pred (Tue) = Mon
succ (Tue) = Wed
```

Note that the "lowest", i.e. the first item, of a type has no predecessor and the "highest" value has no successor, for example succ(Sun) would cause an error.

It will come as no surprise to find that when the program is running PASCAL actually manipulates integer values which replace the actual ones.

Exercises

Write a program which reads a christian name and a surname and prints out the full name, which is held in the variable FullName.

Expand the above program to print out the surname followed by the initial of the christian name. (use the PASCAL help system to find details of standard procedures to manipulate strings).

Investigate the effects on compilation of mixing types in expressions. See if you can catch the compiler out.

Write a type definition for the standard type logical, which can have the settings True or False. How would a type similar to this (e.g coin = (heads, tails)) differ from the logical type?

Syntax

Unlike English, the PASCAL language is rigidly defined. The syntax of the language, i.e. how you may assemble reserved words, variable names etc. to produce a valid program, was carefully designed by N. Wirth and he chose a very neat way of expressing it. PASCAL can be described using a series of syntax diagrams, which map out the constructions which are valid.

The whole of PASCAL can be defined in this way, and at the end of these notes you will find a copy of the entire language syntax. Note how some parts of the language are defined recursively, i.e. in terms of themselves.

The syntax maps provided are for the standard PASCAL lauguage, not Turbo PASCAL.

Statements

A PASCAL program is composed of a number of statements, each separated by the ubiquotous semi-colon (;). Sometimes it is useful to group several statements together into a compound statement; this is done by enclosing them in the reserved words begin and end .

e.g.

```
Count := Count + 1 ;
```

- this is a single statement as opposed to :

```
begin
temp := FirstValue ;
FirstValue := SecondValue ;
SecondValue := temp ;
 end ;
```

- which is a compound statement which swaps the values in the variables FirstValue and SecondValue.

The Flow of Control Within a Program

Up until now all the program we have written have run in a very simple way, i.e. they start at the first line and then obey every single line up until the end. However the real power of a computer program is that it can change the way it executes according to the data which it processes. PASCAL provides a number of constructions which allow you to modify the way your program runs.

The IF - THEN - ELSE construction

The first we shall look at is the if - then - else construction

The condition in a conditional statement returns True or False . If it is true the statement after then is executed. If False the statement after else is performed. Note that the else portion is optional, because sometimes you will only want do something if a condition is true.

An example of the use of this construction could be as follows:

```
if Size = MaxSize
 then
```

```
        Writeln (`Overflow')
    else
        Size := Size + 1 ;
```

This piece of code would add one to the value in Size, unless it was equal to the value in MaxSize, in which case a warning would be output.

Note that the statement which follows then or else can be a compound statement, allowing you to do as many things as you like as a result of a particular condition.

PASCAL allows if - then - else constructions to be nested, i.e. put inside one another, as in the following:

```
if FileError
 then
      if EndOfFile
      then
           begin
           writeln (`End of File hit') ;
           NewFile := True ;
            end
       else
            begin
           writeln (`Filing system error') ;
           abort := True ;
            end
else
      begin
      Writeln (`Entry Filed OK') ;
      EntryMade := True ;
       end ;
```

Note that there is scope for ambiguity because the else clause is optional, consider the following:

```
if condition 1
 then
      if condition 2
      then
           statement 1
      else
           statement 2 ;
```

This is legal PASCAL, with only one of the if constructions containing an else clause, but it is ambiguous, in that the syntax diagrams do not explain which then clause the else belongs to. We can make a guess based on the layout above, but we cannot expect the compiler to do this.

Many other programming language share the " dangling else problem", in the case of PASCAL it is solved by matching each else with the nearest then, i.e. statement 2 would be performed if condition 1 is True and condition 2 is False.

Note that this is a situation where good layout makes it very simple for us to understand what is going on.

The REPEAT - UNTIL construction

You will often come across situations when writing a program where you wish to repeat a particular piece of the program until a particular condition is True. In order to allow you to do this PASCAL provides the repeat - until construction allowing a number of statements to be repeated until a statement is True.

The statements are repeated in turn and then the condition is tested. If it is true the statement following the repeat - until construction is obeyed. If not the statement following repeat is returned to and so on, for example:

```
Count := 1 ;
repeat
      Writeln (Count) ;
      Count := Count + 1 ;
until Count = 11;
```

- would write out the numbers 1 to 10.

Note that if the condition is False to start with the statements will still be obeyed once because the test is performed at the end of each pass through the sequence, i.e the output from the program.

The sequence of instructions must include one which at some time will cause the end condition to become True, otherwise the loop will never stop.

The WHILE - DO Construction

The repeat - until construction allows statements to be repeated until a particular condition becomes true. However the statements to be repeated are always obeyed once, before the condition is tested.

Sometimes we will want to test for the end point of the repetition first, so that if the construction is entered with the end condition True nothing happens. You could get this effect by putting a conditional statement before a repeat - untl construction as follows:

```
if condition
then
      repeat
            statements
until condition ;
```

However, this is clumsy and so PASCAL provides another construction to handle repetition of this kind. This is called the while - do construction:

When the program runs the condition after while is tested. If it is True the statement following do is then obeyed. The condition is then tested again. The statement are repeated until the condition becomes False, when the program moves on to the statement following the while - do construction.

Note that, unlike the repeat - until construction, only a single statement is repeated after the do. It is of course simple to make this a compound statement to repeat more than one statement:

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

Note that the effect of this program is exactly the same as the first example of the repeat - until construction, i.e. it prints out the numbers 1 to 10. This illustrates a very important point; it is perfectly possible to get the same results from programs written in different ways using different constructions. Both program segments can be considered correct solutions to the task "write a program which prints out the numbers 1 to 10", as would a program which simply contains 10 calls of Writeln!

In fact the actual number of different language constructions needed to write programs is very small and we already know most of them. The additional constructions allow you to write programs more efficiently, as there will usually be a construction fitting the particular situation.

If we were being critical however we would see that the above solution to the problem "write a program which prints out the numbers 1 to 10" is marginally less efficient than one using the repeat - until construction because it evaluates the condition one more time than is necessary. In a simple example

like the one above this is not important, but if the loop was to be used many times as part of a larger construction this could result in a program which ran noticeably slower.

Knowing which construction is most applicable in a particular case is part of the skill of programming.

The FOR - DO Construction

Sometimes a program needs to repeat some statements a particular number of times; you could do this with either the repeat or while constructions but it is useful to have an additional construction to perform this - it enables clearer programs to be written. The for - do construction is defined as follows:

The variable is known as the control variable because it "controls" the execution of the loop. It must be a simple scalar type (i.e. not real) and must have been declared. When the construction is entered the control variable is set to the value of expression1 and the statement executed. When the statement has been performed the control variable is incremented if the to keyword is present or decremented if the downto keyword has been used. The loop continues until the control variable exceeds the value of expression2 if to was used or if the control variable becomes less than expression2:

```
for i := 1 to 10 do
    Writeln (i) ;
```

- is the definitive solution to the problem "write a program which prints out the numbers 1 to 10". I promise I will never mention this problem again!

Note that PASCAL takes care of the initialisation, updating and testing of the control variable, so you should never do anything which affects the value of a control variable from within a FOR - DO. If you find yourself in a situation when you need to do such things you should be using a repeat -until or while - do construction instead.

The definition of PASCAL does not say what the value of the control variable is on exit from the loop. You may find that Turbo PASCAL leaves a value in it, probably the value one further on than the terminating value, but you should never make use of this value as this is not standard PASCAL. Some versions of PASCAL explicitly "undefined" the control variable on exit from a for - do loop and any attempt to use this value will cause an error.

The start and termination values, expression1 and expression2, are evaluated once by PASCAL and then stored in special temporary variables, i.e.

```
j := 2 ;
for i := 1 to j do
begin
    j := j+1 ;
    Writeln (i) ;
end ;
```

- would print out:

```
1
2
```

- and then stop. This makes the program run faster, because the expressions are not evaluated every time around. It also stops any possible ambiguities.

Provided that you use a simple scalar type for the control variable and that this is consistent with expressions 1 and 2 you can use other types, this can be useful:

```
var
letter : char ;
begin
    for letter := `A' to `Z' do
    Writeln (letter) ;
end .
```

Exercises

Decide whether or not the following statement construction is valid within the definition of the PASCAL language:

```
begin
begin
begin
x := 0 ;
y := 0
end
end
end ;
```

The standard function sqrt takes a real parameter and returns the square root of it, e.g.

```
x := sqrt(2) ;
```

- would place the square root of 2 in the variable x. Write a program which asks a user for real value and then prints out its square root. Your program must handle the input of negative values in a reasonable way, i.e. it should produce a warning if one is input.

Expand the program you have written above to repeatedly ask for real numbers until the terminating value 99 is given.

Use the repeat - until construction to write a program which requests a value and then prints out a times table for that value, from one times to twelve times. The output should be in the form:

```
1 times 2 is 2
2 times 2 is 4
3 times 2 is 6 .....
```

Make use of the for statement to re-write the program above.

The factorial of an integer is the product of all integers less than and including the integer, e.g.

```
factorial 6 = 6 * 5 * 4 * 3 * 2 * 1
```

Write a program which inputs a number and prints out its' factorial.

Write a program which reads in two integers, x and y, and then prints a rectangle of "*" characters on the screen x wide by y high.

Hint : Write (`*') writes a * but does not take a new line. Writeln with no parameters simply takes a new line. You can "repeat repetition" by putting one FOR - DO loop inside another.....

Investigate what happens if you set up an "impossible" FOR -DO construction, e.g. i := 10 to 1.

Procedures and Functions

PASCAL is a language which allows you to write structured programs in which a task is broken down in to a number of smaller parts, each of which may contain other tasks and so on. Until now we have done nothing other than say that this is so, now it is time to see how this is done within the language.

PASCAL allows you to assign a name to a particular program statement. Once the name has been defined merely specifying it within the program will cause the statement associated with it to be obeyed. A piece of program which is linked to a name in this way is called a procedure . We have been making use of procedures since we started writing programs, the ones we have been using are the so called standard ones, for example Writeln.

You use procedures for two reasons, they allow you to break up a large program into a number of more easily manageable chunks, and they also make it possible to use the same code in many different parts of a program.

Consider the following statement:

```
for i:= 1 to 10000 do ;
```

This apparently stupid piece of code can be quite useful; it pauses the execution of a program for a small time. Note that before you resort to such desperate measures to halt a program you should first find out if there is not a way of pausing the program which is built into the system you are using - the above is very wasteful of computing resources!

You may be writing a program which needs to pause in many different places, for example it may display several screens full of information at regular intervals. You could write the statement above each time, but this is tedious. A much better solution would be to define a procedure, perhaps called pause, to do the waiting for us:

```
procedure pause ;
begin
    for i:= 10000 do ;
end ;
```

Now, when we want the program to pause we simply write:

```
pause ;
```

PASCAL requires that a procedure be defined before you attempt to use it; procedures are in fact defined at the start of a program block, after the const, type and var definitions.

When the PASCAL compiler sees the keyword procedure it next expects the name the procedure is to have, then a semi-colon, and then the body of the procedure. Note that procedures are not executed when they are defined, PASCAL merely makes a note of the name of the procedure and puts the procedure body to one side ready for any references to the name.

When the name of the procedure is encountered within the program the compiler actually produces a call to the procedure body which it previously stored. A procedure call causes the procedure body to be obeyed and execution to return to the statement following the call, i.e.

```
Writeln (`Hello') ;
pause ;
Writeln (`Mum') ;
```

- would produce a line with "Hello" on it and then, after a short pause the word "Mum" would appear.

Although a procedure may be called many times from within the program text the actual procedure body is only compiled and stored once. This means that you can use procedures to make programs smaller and faster to compile as well as easier to write and understand.

Local and Global Variables

A procedure can be more than just a single statement, it can be a "mini-program" with its own types, variables and even procedures. This makes sense, because there may well be variables which only the procedure makes use of, and these are best declared just where they are used. Variables which are declared within a procedure, in exactly the same way as they are declared within a program, are local to it; i.e. they can only be used within the procedure body. They are referred to as local.

Variables defined within the main program are global, i.e. they can be used anywhere within the program - including within procedures. The range of the program in which a variable can be legally used is called its scope.

At the end of a procedure the values in all the variables local to it are discarded. You cannot expect the values in local variables to be the same on entry to a procedure as they were when the procedure was last exited: when PASCAL enters a procedure which makes use of local variables it simply puts aside some space to hold the local variables, in the same way as it reserves memory for variables in a program. When the procedure finishes this memory is returned to the free space.

Consider the following:

```
program paradox (input, output) ;
var
```

```
    x, y : integer ;
procedure changexy ;
var
    y : integer ;
begin
    x := -1 ;
    y := x + 3 ;
end ;
begin
    x := 1 ;
    y := 1 ;
    changexy ;
    Writeln (x, y) ;
end .
```

This program illustrates a paradox in that you may wonder what it will display. However, it is really not that complicated:

x and y have global scope, i.e. they are defined at the start of the program and can be legally used anywhere within it, including within procedures.

The procedure changexy declares a local variable called y. Within the procedure this variable will be used instead of the global variable y. When the procedure changexy finishes the contents of the local variable y will be "forgotten".

Within the body of changexy the global variable y is said to be "scoped out" by the local variable y. Note that the entire program contains three different variables: global x and y and local y. The fact that two variables have the same name does not mean that they refer to the same memory location, any more than a particular street name refers to the same street in two different towns. The program would therefore print out:

```
-11
```

Global variables hold their value even when " scoped out " by identically named local variables: they just become temporally inaccessable.

Note that this is useful, if several people are writing different parts of a large program they can each write a particular procedure. If they all use local variables within their code there is no possibility of problems due to " clashes " of variable names.

Sometimes a procedure may affect the value of a global variable. Note that this can cause confusion, in that a person reading the code and seeing a procedure call may not realise that certain variables have been changed. A procedure which changes the value of global variables is said to have side effects . You should make use of side effects with care.

Simple Parameters

Sometimes you may wish your procedure to work on something, e.g. in the example above we may wish to select the duration of the pause. Many standard procedures can also be supplied with something to work on, for example the standard procedure Writeln can accept things to be written out. Items you supply to procedure are called parameters, just as in MS-DOS we supplied parameters to commands.

When you define a procedure you specify what parameters it is to have and their type in the following way:

```
procedure pause ( duration : integer ) ;
var
timer : integer ;
begin
    for timer := 1 to duration do ;
end ;
```

The call of pause would now be:

```
pause (1000) ;
```

A list of parameters, together with their types is supplied after the name of the procedure. When the procedure is called a corresponding list of items must be provided, otherwise PASCAL will refuse to compile the program.

When PASCAL calls this procedure it sets aside a special variable with the name duration. Before the body of the procedure is performed duration is given the value which was supplied as a parameter, in the case of the above example it would be set to 1000. Within the procedure the parameter can be used in exactly the same way as a local variable and its value is also destroyed on exit - ready to hold the value supplied in the next call.

Exercises

Improve your factorial program so that the following would cause the factorial of 7 to be printed:

```
factorial (7) ;
```

Write a list of all the variables in this program and their scope:

```
program taxing ( input, output ) ;
    var
    i, j, k : integer ;
    x, y, z : real ;
    Up, Down : boolean ;
    procedure proc1 ( x : integer ) ;
        var
        a, b, Up : real ;
        begin
        Up := x/7 ;
        Down := (x=1) and Up ;
        j := 1 ;
        end ;

    procedure proc2 ( Up : real ) ;
    var
        i, j, k : real ;
    begin
        proc1 (1) ;
        Down := (1/Up) > 1 ;
    end ;

    begin
        x := 1 ;
        y := 2 ;
        i := 1 ;
        proc1 (i) ;
        if up then proc1 (1) else proc2 (1.22) ;
        proc2 ( y ) ;
        z := x + y ;
    end .
```

Work out, without running the program, what the final values of each variable would be.

The standard function ln (real) returns the natural logarithm of a number. The standard function exp (real) returns ex of a number. Write a procedure which sets the value of the global real value result to the cube root of its parameter. By repeatedly calling the function and then multiplying back up again investigate the effects of errors.

Returning Results via Parameters

We have seen how parameters work, in that as far as the procedure is concerned you can regard a parameter as a local variable which is pre-set to the value which was supplied when the procedure was called, i.e.

```
program sample (input, output) ;
var
    i : integer ;
procedure example ( param : integer ) ;
begin
    Writeln ( param ) ;
    param := 99 ;
end ;
begin
    i := 1 ;
    example (i) ;
    Writeln (i) ;
end .
```

The procedure example has a single parameter, with the name param and the type integer. Once the body of the procedure is entered it is as if there is a local variable called param which is given the initial value supplied when the procedure is called. In the example above the value in I, 1, is placed into param, which is then used in the procedure. When the value in param is changed later in the procedure this does not affect the value in i, i.e. the program above will print out:

```
1
1
```

Sometimes you will want to write procedures which change the value of their parameters, i.e. they affect the contents a variable specified as a parameter. Up until now the only way that a procedure can return a value is by the use of side effects.

To allow the contents of a parameter variable to be changed by the procedure to which is a parameter we give additional information in the procedure heading by use of the keyword var which is given before a list of parameters which will be changed by the procedure:

```
program sample (input, output) ;
var
    i : integer ;
procedure example ( var param : integer ) ;
    begin
    Writeln ( param ) ;
    param := 99 ;
    end ;

begin
    i := 1 ;
    example (i) ;
    Writeln (i) ;
end .
```

The var above, before the name of the parameter, tells PASCAL to use the actual variable within the call of the procedure, rather than a copy. So in the call "example (i)" whenever the parameter "param" is referred to this uses the contents of i, i.e.

```
1
99
```

- would be printed out.

When the procedure is called the address of the parameter being used is passed, rather than the contents of the variable itself. This means that in the above program it is not legal to have a call of example as follows:

```
example (2) ;
```

The reason is that the constant value 2 does not have an address because it is not a variable. In fact it might be interpreted as an invitation to change the contents of memory location number 2, something which would have catastrophic results!

Note then that if you specify a procedure parameter as being variable by use of var you must not use constant parameters in calls of that procedure. If you try to do this at best the program will fail to compile, at worst it will crash the computer when it runs.

You can mix parameters you wish to change and those you do not in the same procedure heading:

```
procedure mixed ( integer : input1, input2 ;
        var integer : output3, output4 ) ;
```

Functions

Functions operate in an identical manner to procedures, with the difference that a function is given a type, and it returns a value of that type. An example of a built-in standard function is sin (x), which returns the sine of the numeric parameter x. Functions can only return a single value, so if you need to return more than one piece of information you will have to use variable parameters (see above).

The type of a function is defined after its heading, for example:

```
function factorial ( x : integer) : integer ;
```

- defines a function called factorial with one parameter which returns the factorial of the parameter supplied, so that your main program could contain:

```
Writeln (`The factorial of ',n,' is ', factorial (n) ) ;
```

When the PASCAL compiler sees the name of a function it first makes sure that the type of the function as declared is correct in that context, and then it makes a call to the function.

You return the result from a function by assigning to the name of the function, which can be used within the function body as a variable with the same type as the result, for example:

```
function factorial ( x  : integer ) : integer ;
begin
    factorial := 1 ;
    for x := 1 to x do
    factorial := factorial * x ;
end ;
```

If you do not assign a result to the function strange things happen, i.e. what will happen is not defined.

Recursion

Recursion can be very useful on that once in a blue moon occasion when recursion is very useful. You will not write programs which require recursive code very often, but the technique does have applications.

The old joke definition of recursion is:

> Recursion : see Recursion

- and illustrates quite well what recursion actually is.

Recursion is a technique whereby a procedure or function calls itself. This may sound a very silly thing to do, akin to an infinite loop, but certain features of PASCAL make it possible and certain kinds of problem are amenable to such an approach.

You may wonder how a procedure can usefully call itself without losing its mind, this is done by the use of local variables. Each called copy of the procedure is given its own local variable space on a stack (a stack is a storage area which is allocated on the basis that most recently allocated areas are those which are the first to be handed back - i.e. last in first out). If you think about a series of procedures calls this is the only way to make the memory available. When a call of a procedure terminates, which it must do at sometime if the program is ever to stop, the memory which was allocated for it is handed back and the variable space for the calling routine is re-activated.

The usual example of a recursive program is one to calculate a factorial. (a factorial of a positive integer is that integer multiplied by all the integers between it and 1, e.g. factorial 4 = 4*3*2*1) A recursive solution to this problem goes like this:

```
program recursion ( input, output ) ;
var
     start integer ;
function factorial ( n : integer ) : integer ;
begin
     if n>1
     then
          factorial := n * factorial ( n  - 1 )
     else
          factorial := 1 ;
end ;

begin
     write ( `What number do you want the factorial of : ' ) ;
     readln ( start ) ;
     writeln ( factorial ( start ) ) ;
end .
```

Using the fact that for any positive integer n, the factorial of n is n times the factorial of n-1, the factorial function calls itself to work out the factorial of the number one less until the terminating value of 1 is reached. We know that the factorial of 1 is 1, so at that point a "proper" answer is returned, which is then multiplied back up by the previous calls.

Note that in the above there is always a situation when factorial returns a value and does not make a recursive call. If this was not the case the program would disappear down a bottomless pit. (in fact it would run out of memory first because each incarnation of the function would use up an amount of memory and memory is a finite resource)

You can get the effect of recursion indirectly in a situation where a procedure of function calls another which then calls the first one, and so on - as in the following:

indirect recursion : see "recursion (indirect)"
recursion (indirect) : see "indirect recursion"

Note that there could be many procedures in an "indirect loop", however the philosophy of "top down" programming does not tend to produce programs which contain this problem.

The above example of recursion involves using it to solve a mathematical problem which could just as easily have been solved using a for - to - do loop, i.e. the problem did not require a recursive solution. The kind of problem which requires a recursive solution is:

> If you are searching down a data structure and you come to a fork, i.e. two branches down which you wish to search. A recursive call of the search routine for each branch is a very good way of doing this, particularly as each branch may contain other branches.

My experience has been that once a programmer knows about recursion he or she tends to start looking at each problem and trying to find a way that recursion can be "fitted in" to a solution so that they can say "This program uses recursive techniques". Do not do this. Recursion is rather heavy on machine resources, particularly memory, and should only be used when it is obviously applicable. I prefer the original solution.

Constants

PASCAL allows you to build constant values into a program text. An example of this is when you wish to make use of "special" values within your program which have some meaning, for example you could use the value of -99 in a particular value to indicate that an error has occurred. There would be nothing wrong in implementing your error tests in the form:

```
if condition = -99 then writeln (`Error') ;
```

- but this would not be meaningful.

You could set up a variable called "error" which held the value and then test against it but this is wasteful of space and results in needless accesses of what is in fact a constant value. PASCAL allows you, in the CONST section of the program to declare the error constant as follows:

```
const
error = -99 ;
```

What happens thereafter is that PASCAL behaves as if the string error was the string -99.

You can also use this technique if you wish to dimension areas of memory and then use program constructions to search the memory. If you use a constant to dimension the memory, rather than using a particular constant value, when you need to hold more or less information you simply change the constant at the start of the program and re-compile it.

You can only specify constants for scalar values, not reals.

Arrays

PASCAL allows you to create as many variables as you need, and to give each one a name. Furthermore it allows you to use different "types" of variables and even create "custom" types if the ones available are not suitable. You might think that these facilities are sufficient, however there are situations where other data structures are required.

Consider the problem of reading 10 integers and then printing them out in ascending order. Computers are good at this kind of thing, so you start writing your program knowing that it can be done (a great advantage!). You decide to declare the variables first:

```
var
no1, no2, no3, no4, no5, no6, no7, no8, no9, no10 : integer ;
```

- then you read the values in:

```
Readln ( no1, no2, no3, no4, no5, no6, no7, no8, no9, no10 ) ;
```

- then you wonder about sorting them. Now we can compare two values, and perhaps swap them if they are in the wrong order, but the IF - THEN construction that we would need just to find the largest number in the set would be enormous. Even assuming that we could come up with a program which could handle 10 numbers, if we ever needed it to handle 10000 values the program would need a complete re-write.

It is clear that the standard way of declaring variables is not adequate. PASCAL allows you to declare an "array" of variable locations which are identified by a single name, with each individual item in the array being specified by a numeric " subscript ", for example:

```
var
numbers : array [ 1 .. 10 ] of integer ;
```

This declares an array called numbers which can hold 10 individual values. You refer to each element as follows:

the first element : numbers [1]
the second element : numbers [2]

The value in the square brackets is an integer which identifies the element of the array you refer to.

This in itself is not the breakthrough that we have been looking for. We are now able to declare our area of memory more easily, but it is not clear how this will help us sort the values. The power of arrays comes when you decide to use variables to give the subscript values as in the following:

```
var
     numbers : array [ 1..10] of integer ;
     i : integer ;
begin
     for i := 1 to 10 do
     readln ( numbers [i] ) ;
end .
```

This piece of code declares an array and then reads 10 numbers into it, much more simply than the original code. Also, if I wanted to work on 1000 numbers I would only have to change two values in the code. In fact, when I use values like the size of arrays I would tend to declare them as constants so that changing a single constant updates the entire program:

```
const
     arraysize = 10000 ;
var
     numbers : array [ 1..arraysize] of integer ;
     i : integer ;
begin
     for i := 1 to arraysize do
     readln ( numbers [i] ) ;
end .
```

Exercises

Write a program which reads in 10 numbers and then prints them out in the reverse order to that which they were entered in.

Sorting with arrays

The array makes it possible for us to allocate storage for large amounts of data and the use of subscripts to identify elements of the array allows us to more easily scan through arrays of information. However we have still not hit upon a technique for sorting the values held in an array.

There are many different techniques which have been developed for use when sorting arrays of numbers. Like all different means of performing a certain task they each have their disadvantages and advantages. A great many tomes have been written about sorting, when you have a specific sorting requirement I would advise you to go and investigate the literature, if you want a fast sort which will work quickly on very disarrayed data I would reccommend the " Shell Sort ".

The technique which we are going to look at is a very simple sorting system which is not very efficient when the data is badly disordered but is sometimes used when only a few elements are in the wrong position. It is also very easy to code, and is known as the " bubble sort ".

In english the bubble sort method goes like this:

1. Compare the values in the top element and the one immediately below it. If they are in the wrong order swap them round.
2. Now compare the values in the one immediately below the top one and the one below that. If they are in the wrong order swap them as well.
3. Repeat this process until you have compared the element just above the bottom one with the bottom value and swapped if necessary.
4. Go back to step one and repeat the process.

You know when you have completed the sort because you will make a pass through the data without making any changes to it. At this point you can stop and announce the data as sorted.

This technique breaks down the apparently complex problem of sorting, where it appears that the whole of the data must be considered at one time, into a series of steps which are repeated many times, until the data is put into the correct order. With each successive pass through the data values "bubble" towards their final position, which is how the sorting process got its name.

If you consider further how the sorting process works you can also improve the technique slightly. The first pass from the top to the bottom of the data must put the bottom value into its correct position, because it will have been swapped with all the values which are in the wrong place. Therefore after the first pass you do not need to go all the way to the end of the data, and with each successive pass you can reduce the distance you have to move down the data.

Note that for the bubble sort the worst possible case, when the data is in exactly the wrong order, would required one pass through the data for each value in the list. If only a few values are out of position the number of passes will be much fewer.

You may be interested to know that some solutions which are faster than the bubble sort get their extra speed by moving the out of order values more than just one position when they adjust where they should be, which is fine when the data is in a random order but can result in much poorer performance it the data is already partially sorted.

The following program reads in 10 values and then sorts them using the bubble sort method:

```
program sort (input, output ) ;
{ Reads in values and sorts them }
const
     arraysize = 10 ;
var
     i, limit, temp : integer ;
     values : array [ 1..arraysize ] of integer ;
     doneswap : boolean ;
begin
     { First read the values }
     Writeln ( `Input ', arraysize,
          ' values. Press Enter after each one.' ) ;
     for i := 1 to arraysize do
          Readln ( values [i] ) ;
     { Set the endpoint }
     limit := arraysize - 1 ;
     { Now begin sorting }
     repeat
          { Set our swap marker }
          doneswap := false ;
          { Make a pass through the data }
          for i := 1 to limit do
          if values [i] < values [i+1]
          then
            begin
            { Need to do a swap }
            temp := values [i] ;
            values [i] := values [i+1] ;
            values [i+1] := temp ;
            doneswap := true ;
            end ;
          limit := limit - 1 ;
     until doneswap = false ;
     { Now print out the results }
     Writeln (`Sorted output :') ;
     Writeln ;
```

```
for i := 1 to arraysize do
    Writeln ( values [i] ) ;
end .
```

Note the use of a repeat - until construction to repeat the scan through the data until a pass is completed which does not involve a swap. Note also that in order to swap two elements in an array it is necessary to make use of a third, temporary value. I have also made use of the CONST facility to define the size of the array so that this program could be equally well adapted to sorting 10000 values as 10 (although these would be rather tiresome to type in!

More than one dimension arrays

A single dimensional array can be represented as a row of boxes, each of which is identified by its subscript number, with the whole row being known by the name of the array. This is a perfect construction for storing a list of items but sometimes you may need to store tabular information, for example a representation of heights on a two dimensional grid, or perhaps a table of cricket scores for a number of matches.

You tell PASCAL that an array has "two dimensions" by simply specifying it as such, e.g.

```
heights : array [ 1..maxx, 1..maxy ] of integer ;
scores : array [ 1..11, 1..NoOfGames ] of integer ;
```

You then specify which particular element you want by giving two subscripts, e.g.

heights [1, 1]- would specify the first element in the array
scores [3, 2] - would specify the score by the third batsman in the second match.

How you choose to make use of the array structure is up to you when you design the program.

More than two dimensions...

You can add as many dimensions to an array as you think your problem will require, sometimes you can make use of three dimensional arrays which could be regarded as representing a series of points in space, or a number of tables. If you need it, you could even go as far as to use four dimensions, although you will probably never need to do this and such techniques can lead to serious brain strain if you do!

Note that each time you add a dimension you multiply the amount of memory you require by the range of the new subscript. This means that if you move off into the higher dimensions you will quite soon find yourself out of memory. If you are using strange, many dimensioned, arrays to solve your problem you are probably not using a very good approach and it would perhaps be best to re-consider things.

Note also that because arrays are very voracious of memory it is worth paying more attention to the range of values you are going to store in them. If you are using a single variable to store a value you do not waste much memory using an integer where a byte would suffice, but it you have a very large array it could make a big difference. If the range of values you have to store is very limited consider using a type as this allows the compiler to make use of even smaller memory locations which might not be available any other way.

If the compiler you are using does not support the byte type allowed by Turbo PASCAL you can simulate this by declaring an array of type char and then making use of the ord function to store numeric information in it, although this may impose a slight performance overhead (see Packed Arrays) and you will have to check in the guide for the particular Pascal implementation you are using to find out if characters are actually stored in smaller memory locations.

Packed Arrays

Some PASCAL systems (but not Turbo PASCAL) allow you to specify an array as *packed*. This tells the compiler to squeeze as much into the memory as possible, sometimes going as far as to try to put

several elements in a single memory location. You specify an array as packed by making use of the packed keyword:

compactnos : packed array [1..compactsize] of integer ;

If the compiler is able to, it will squeeze the array compactnos into the smallest amount of memory possible (Turbo PASCAL just ignores the packed keyword).

However: Many months ago we discussed the trade off between size and speed, i.e. rather like cars big programs tend to run faster than little ones. Packed arrays are an example of this law in action. Although packing allows you to reduce the amount of memory you need to use it also imposes an overhead on the access of elements, i.e. in order to read or store an array element the program has to do more work splitting it off from other elements which may be packed in the same memory location. This can have drastic effects on the speed of your program, particularly if it does many array accesses.

Therefore, unless packing is the only way that you can fit the program into the memory you have, you should not normally mark arrays in this way. It is not good housekeeping to have large amounts of memory holding nothing and a program which has to grind just to preserve these empty locations.

If you are forced to pack your data into memory you can sometimes improve the situation by making use of pointers to the packed data which you do not pack, e.g. you may have a large number of customer records which you need to sort into alphabetic order and then into order based on customer reference number and so on. The mass of data is such that it has to be held packed, but you can greatly improve things by keeping a list of pointers to the data and then moving them, rather than the data itself. The use of pointers in this way has two major advantages:

1. It reduces the amount of information moved at each swap.
2. The pointers may not be packed and so can be moved more easily.

Exercises

Write a program which would accept signals from an altimeter and display the height of the aircraft on the screen.

?? The altimeter can be regarded as a device which sends out pulses of different types when a change of height is detected. It is simulated at the keyboard in the following way:

?? The receipt of a + character means an increase of height of 500 feet.

?? The receipt of a - character means a decrease of height of 500 feet.

?? The program should request an initial altitude which can be regarded as tht height of the ground above sea level. It will then accept input from the "altimeter" and display the height as a numeric value and also in some analogue way, e.g. the position of a charactyer up and down the screen. (use the Turbo PASCAL function GotoXY to move to a particular screen position).

?? The program should output a warning message if the height is within 1,000 feet of the ground or more than 20,000 feet.

?? You can use the facility:

Read (kbd, charvar) ;

- to read a single character from the keyboard.

Improve your altimeter program so that it gives the pilot warnings if he climbs continuously for more than 200 feet or looses height continuously for more than 400. Add a third sensor input 0, which means no height change in this time period.

Add X and Y information which is supplied from a directional sensor at the same time as the altimeter information, i.e. for each time period your program will receive two characters, altimeter information first according to:

+, - or 0	height change in this period
L, R or 0	heading change in this period

A L means a turn to the left of 10 degrees, a R means a turn to the right of 10 degrees and a 0 means no turn in this time period.

Improve your display to give a heading readout in degrees, assuming that the aircraft starts of pointing north (0 degrees), south is 180 degrees, east is 90 degrees and west is 270 degrees.

File Handling in Pascal

Up until now the devices our programs have interacted with have been the keyboard and screen (along with imaginary altimeters etc). We now have, by the use of arrays, the capacity to store large volumes of data within the program but at present we have to type the data in each time!

We have been using the filing capabilities of the computer for some time, each program that we write has been stored on the disk for us, in this case the Turbo Pascal system does the file handling for us. However, it is perfectly possible for our programs to perform file manipulation so that, for example, our aeroplane control program can read information about the terrain about to be flown over.

On most computers it is the operating system which actually provides access to files, on the IBM PC it is MS-DOS or Windows which looks after the filestore.

When you wish to access a file you must first set up a link between your program and the file in question. You may have several of these links set up at once, as your program may be in communication with more than one file at a time. Within PASCAL such a link is called a file . A file has a type, which is associated with the data you wish to store in it, for example:

```
DataFile : file of integer ;
```

- would define a variable called DataFile which specifies a link to a file which could be used to store integers in the same format as PASCAL holds them within the program. Note that this would not mean that you could look at such a file and easily spot the numbers in it, because they are written in the internal format they only make sense when the file is read in to a PASCAL program.

If you wish to produce a file which contains readable characters a standard type called text is available which specifies a link to a file which contains readable characters.

The matter of what type of file you set up is very important, and we shall return to this later.

Assigning a File

The standard procedure assign assigns a file variable to a particular file. The name of the file is specified as a string of text, for example:

```
var
ListOutput : text ;
begin
assign ( ListOutput, `outputfile' ) ;
```

- defines a file variable of type text and then the program assigns the variable to a file called `outputfile'. Note that at this point the PASCAL system does not care whether or not the file `outputfile' exists because it has not gone and looked for it yet.

Note that the filename in the assign statement could just as well have been a string variable, allowing the name of the file to be selected at run time:

```
var
    ListOutput : text ;
    ListOutputName : string [20] ;
begin
    Write ( `Give name for output file : ' ) ;
    Readln ( ListOutputName ) ;
    assign ( ListOutput, ListOutputName ) ;
```

- would ask the user for the name of the output file and then assign it.

Beginning Data Transfer

Once the file has been assigned the next stage in making it available for use is to tell PASCAL what use you are going to make of it. Broadly speaking files are used in three ways, for output (Write), for input (Read) and for update (Read and Write).

If you are going to use the file for output it does not matter whether the file exists or not (unless you are concerned about overwriting an existing file).

If you are going to use a file for input or update it must already exist.

PASCAL provides two standard procedures which make a file available for use, reset and rewrite .

Reset

Reset has one parameter, the name of the file variable which has been assigned (not the name of the file). It is used to set up an already existing file and place the file pointer, i.e. the device used by the PASCAL system to keep track of where you are in the file, to the start of the file. You can use Reset to set up a link or to move the pointer back to the start of a file. Note that if the file does not exist your program will generate an error and stop. We will look at a way of trapping this error and producing an appropriate response later.

You use Reset when you wish to access a file which already exists, e.g.

```
Write ( `Give the name of your data file : ' ) ;
Readln ( DataFileName ) ;
Assign ( DataFile, DataFileName ) ;
Reset ( DataFile ) ;
```

Rewrite

Rewrite also has one parameter, again the name of the file variable which has been assigned. It is used to set up a file which will be used for output. If the file already exists its contents are erased and the file pointer is placed at its beginning. If no file exists an empty one is created and the file pointer set, e.g.

```
Write ( `Give name for output file : ' ) ;
Readln ( ListOutputName ) ;
assign ( ListOutput, ListOutputName ) ;
Rewrite ( ListOutput ) ;
```

Note that although a file will be created if one does not exist this is no guarantee that the rewrite call will never fail, for example if the disk you are trying to write to is full an error will be generated. Again, we will look at ways of trapping errors of this kind later on.

Using Files with Read and Write

The standard PASCAL procedures Read, Readln, Write and Writeln can be told to use a particular file by giving a file variable as the first parameter of the items to be written, for example:

```
Readln ( DataFile, count ) ;
```

- would read from the file connected via DataFile rather than from the keyboard. Write output is re-directed in exactly the same way. Note that this means that if you want to output something to both the screen and a file you will need two write statements.

When using the read and write commands with files it is important to note that the difference between the "ln" versions of the procedures, which handle line ends, still applies.

Closing the File

As I mentioned above Turbo PASCAL makes use of MS-DOS to perform all file handling, i.e. when you do a Write statement Turbo PASCAL passes the text on to MS-DOS which actually stores the information on the disk. In order to reduce the number of times the disks are accessed MS-DOS waits until a number of characters need to be written before starting the drive. When your program finishes writing to a file is is therefore quite likely that a number of characters remain in the buffer. It is therefore best to use the standard procedure close to explicitly shut the file after you have finished with it. Close has one parameter, the file variable of the file you wish to close:

```
close ( ListOutput ) ;
```

- would write all buffers to the file associated with the file variable ListOutput and make this variable available for use with other files.

It is also advisable to use close with data files, as you may want to use the same file variabe with another file.

Exercises

Write a Turbo PASCAL version of the TYPE command in MS-DOS. The program should ask for the name of the file to be displayed and then print it on the screen. The program should pause at the end of each page of 25 lines and wait for the user to press Enter, before continuing with the next page.

Improve your Altimeter program to take a "flight log". This will hold details of the changes in altitude and heading of the plane during a flight. Provide a "replay" facility so that the flight can be displayed once the plane had landed again.

The Case Construction

A very useful PASCAL construction allows you to easily select a particular piece of code depending on the value in a scalar variable. Without it you would have to use a complex IF - THEN - ELSE construction.

For example consider a program which could be used to drive a hotel reservation system. The user gives a single character command, A for add, D for delete, L for list etc. The procedures AddGuest, DeleteGuest, ListGuests need to be called. You could program this with:

```
if selection = `A'
then
    AddGuest
else
    if selection = `D'
    then
        DeleteGuest
    else
        if selection = `L'
        then
            ListGuests
        else
            Writeln (`Invalid Command') ;
```

However this is hard work to code and adding additional commands would be hard work. The CASE - OF construction allows you to code up the above in the following way:

```
case selection of
    `A' : AddGuest ;
    `D' : DeleteGuest ;
    `L' : ListGuests ;
    else
```

```
            Writeln (`Invalid Command') ;
    end ; {of CASE construction }
```

Note that the else part, which identifies a statement to be performed if the selection does not match any of the options, is not available in all PASCAL implementations. If you do not have this construction you will have to check that the selection is in the valid list before you obey the CASE - OF construction. You could use the SET operations (see later) to do this.

The statement which is performed for each option can be a compound statement, which allows large chunks of code to be selected in a CASE construction.

The end of the CASE - OF construction is marked by an END, which tells the compiler that there are no more options to be performed. Note that this can be confusing if you use compound statements in some of the options, I add a comment to document the function of the end in this context. Several values can be given to mark each option, an improved version of the above could be:

```
case selection of
    `A', `a' : AddGuest ;
    `D', `d' : DeleteGuest ;
    `L', `l' : ListGuests ;
    else
        Writeln (`Invalid Command') ;
end ; { of CASE construction }
```

- which would allow lower case command characters to be recognized.

It is also possible to give a range of values to select an option, for example:

```
case height of
    0..20 : writeln(`My God we are all going to die') ;
    21..50 : writeln (`Put Undercarriage down') ;
    51..100 : writeln (`Height Low') ;
end ;
```

You can therefore use the CASE statement in two situations:

1. To select a piece of code depending on a command value.
2. To select a piece of code depending on ranges of a value.

Records

PASCAL allows us to define our own "variables" and the values that they may have by the use of the type section of our program. However, sometimes we want to hold structures of information in one easily defined unit, for example: suppose we were holding a file of information about an account holder in a bank. We would store his or her name, usually as a surname and two christian names, their address, current balance and overdraft limit etc. We already know how to do this using data structures available now, so that we could for example use:

```
var
    surnames : array [ 1..NoOfCustomers ] of string [30] ;
    firstnames : array [ 1..NoOfCusto mers ] of string [10] ;
    secondnames : array [ 1..NoOfCustomers ] of string [10] ;
    balance : array [ 1..NoOfCustomers ] of real ;
    overdraftlimit : array [ 1..NoOfCustomers ] of real ;
```

A set of declarations of the type above would allow us to store information for a number of customers. However we would have to remember that the information in element number 1 of each array was part of a complete record about a particular customer and if we have to move customer records about, for example if we need to perform some sorting, each item in the record needs to be moved separately, which makes the program more complex.

To make designing data structures of this type easier Pascal allows you to design a record structure which you can use to define a structured type which contains the various parts of the record, for example:

```
type
    customer = record
        surname : string [20] ;
        firstname : string [10] ;
        secondname : string [10] ;
        balance : real ;
        overdraftlimit : real ;
    end ;
var
    customers : array [ 1..NoOfCu stomers ] of customer ;
```

The record definition defines a type which consists of the fields specified. This means that the type customer contains all the fields specified, and the array customers is an array with an entry for each item in it. It is possible to treat a variable of type customer as any other variable in that they may be assigned to other variable of the same type, i.e. you can say things like:

```
customers [1] := customers [2] ;
```

- to copy the customer record in element 2 of the array into element 1. PASCAL takes care of copying each field in the record from one location to another. Note that it is not meaningful to do things like:

```
customers [1] := customers [2] + customers [3] ;
```

- because operators are not defined to work between structured types. You can access individual fields within the record by giving them as follows:

```
writeln ( `The name is : ', customers [i].name ) ;
```

The name of the field as defined in the definition of the type is given, separated from the variable name by a full stop ".". You can use a particular field in a record in exactly the same context as a variable of that type, for example:

```
{ Make a withdrawal }
customers [i].balance := customer [i].balance  -amount ;
```

Sometimes you will want to do a large number of operations on the fields in one particular structured variable. You could use the above "." notation to specify each of the fields you want but this can be tedious, and may also result in less efficient code being produced by the PASCAL compiler as it works out a reference to each component individually. An alternative way of working on all the fields in a particular variable is:

```
{ Read a new customer in }
with customers [i] do
    begin
    readln (surname) ;
    readln (firstname) ;
    readln (secondname) ;
    balance := 0;
    overdraft := 100 ;
    end ;
```

The statement following the do (in this case a compound statement) can refer to each of the fields by the names given when the record was defined.

Files of Structured Variables

A particularly powerful feature is the ability to set up a file which has the type of a structured variable, for example:

```
var
    customerfile : file of customer ;
```

You can then use this file to hold objects of the type customer, and write and read entire customer entries with a single statement:

```
write ( customerfile, cust omers [i] ) ;
```

You should use record types whenever you wish to keep a number of pieces of information about a particular item. Note that some versions of PASCAL allow you to pack a record, in the same way as elements in an array can be packed, although there is a loss of speed as the items in the record need to be packed and unpacked for access. In the case of packed records it is very advantageous to use the with - do construction as this tells PASCAL to unpack a particular record and make it available for use.

Exercises

We have looked at how our mini control program would handle input from an accelerometer. In order for use to be able to use information about the rate of change of the speed of the aircraft our program has to have some idea of time passing.

In a real situation you could expect the program to be continually running, getting information from the sensors when there is a change in the situation. In the case of our mythical aircraft if we do not get any input from the sensors we must presume that things are proceeding at the same rate. At the moment we have no way of running the program if no key has been pressed, i.e. we spend most of the programs time just waiting for input from the keyboard.

Turbo PASCAL provides us with an extension which allows us to find out if a key has been pressed, and react if it has. Within Turbo PASCAL the boolean function KeyPressed returns true if a key has been pressed and is waiting to be read, and false if no key is available. If our program has been given no input it must assume that the aircraft is proceeding in the same manner as previously, i.e. for a given time period it must update all the parameters relating to the speed and position of the aircraft and re-display them, then going back to see if any more information is available.

Try to amend our program so that it works like this, and is only informed when a change in the current situation takes place.

Pointers

So far all the variables that we have used have been static, i.e. they are defined at the time the program is compiled and space is allocated for them at that time. This means that you have to make allowance for the largest possible amount of data your program will want to process and then dimension the memory space accordingly.

We can make it easier to change the amount of space the program asks for by the use of constant values in array definitions but we still have to allow the space when we compile the program, not when it runs. This raises the possibility that our program will be unable to process the data not because there is insufficient room in the computer for the data but because the correct amount of storage was not allocated when it was compiled. What we want to do is allocate space for information dynamically, i.e. as the program runs. PASCAL provides us with a way of doing this, and also ways of finding out how much memory is available for use in this way.

A question: if you do not declare the item that you want to use when the program is compiled how to you refer to it?

The answer: when you compile the program simply compile a pointer which can be used to point to the thing that you later require. A pointer is a special variable which does not hold a value, instead it holds a pointer to another variable. Within PASCAL you define a variable as a pointer by placing the character ^ before the name of the type you wish to point to. Note that pointers are typed, in that a pointer to a character can only point to variables of type character, and not to variables of other types. e.g.

```
var
count : integer ;
    { reserves space for an integer value }
cpoint : ^integer ;
    { reserves space for a pointer to an integer value }
```

After the above variable definition you could use count to hold an integer value and cpoint to point to an integer variable. The use of the ^ in the program tells PASCAL to reference the contents of the variable pointed at by a pointer, rather than the actual pointer itself, e.g.

```
count := cpoint^ ;
```

- would put the contents of the variable pointed at by cpoint into the variable count. Note that it is perfectly OK to assign pointers to point to the same places, e.g.

```
oldpoint := cpoint ;
```

- is quite acceptable, provided that oldpoint and cpoint are both pointers and point to variables of the same type. Remember that such things as

```
oldpoint := cpoint+1 ;
```

- are potentially very naughty. PASCAL should not let you do this, what it implies is set oldpoint to point to the memory location after the location containing cpoint. This is meaningless, since we do not know what that location contains. It is however meaningful to compare the contents of pointers for equality, e.g.

```
if oldpoint = cpoint then Writeln ( `Same item' ) ;
```

- would display the message if both pointers were pointing at the same variable.

Pointers can be used to point at existing variables, which may make sorting large datasets easier because you can just sort the pointers, or they can be used to point at new variable locations which are created when your program is running. The standard procedure new creates a new variable of the supplied type and points a pointer to it. Consider the following program snippets:

```
type
    customer = record
        surname : string [20] ;
        firstname : string [10] ;
        secondname : string [10] ;
        balance : real ;
        overdraftlimit : real ;
    end ;
    CustomerPointer = ^customer ;
var
    NewCustomer : CustomerPointer ;
```

The bank customer of the last examples is a structured record type called customer. The type CustomerPointer is a pointer to this type and the variable NewCustomer can be used to point to a variable of type customer. (think about it.....)

The piece of program:

```
New (NewCustomer) ;
```

- sets the pointer NewCustomer to point to a newly created customer record.

Remember that although the word new is used this does not mean that the record will be empty. The new procedure simply tells PASCAL to set aside memory space to hold the information, this memory may have held some other information previously so you must first set information into the new item.

You can access fields in the record in exactly the same way as normal, although you must remember to refer to the contents of the variable pointed at, rather than the pointer itself:

```
Writeln ( `The name is : ', NewCustomer^.name ) ;
```

When you have finished with an area of memory you can use the procedure dispose to make it available again.

Memory Allocation

PASCAL gets the space for new variables from an area of memory called the heap . The heap is a large area of memory space, usually the space left when the program and its data areas have been set up. It is called a heap because data is piled into it in an untidy order, for example:

```
New (Customer1) ;
New (Customer2) ;
New (Customer3) ;
New (Customer4) ;
 Dispose (Customer1) ;
New (Customer5) ;
```

As each of the first four customers is set up space is taken from the heap and allocated for them. However when the fifth customer is added the entry is actually placed where the first customer used to be.

If we had wanted to allocate space for something larger than a customer entry after disposing of customer1 memory would have been taken from the top of the heap and we would have a hole of free space right at the bottom of the heap. This would be used if we need space for something which is smaller than the hole, PASCAL will not try and split a record over more than one hole in the heap.

As the program runs, newing and disposing data, the heap becomes a very untidy structure. You can reach a stage where although there is enough free space to hold a particular item it cannot be allocated as it is not in one block. At this point some systems call in the garbage collector, which is a special routine which compacts the heap, removing all the holes and putting all the free space at one end. To do this it also has to change the contents of all the pointer variables, because it will be moving data around. Garbage collection is a very slow process and some implementations of PASCAL, Turbo PASCAL included, do not provide a garbage collector. In Turbo PASCAL if you try to allocate space larger than the biggest free block your program will fail.

Note that you can prevent the build up of garbage by considering how your program will allocate and return the memory it wants to use.

Lists and Pointers

You may still be wondering what use pointers, new and dispose are. They allow us to grab memory as the program runs but, as we still need a pointer to point to what we want are we not still restricted when we write our program, only now the limitation is the maximum number of pointers that we set up. However, consider the following:

```
type
    CustomerPointer = ^customer ;
    customer = record
        surname : string [20] ;
        firstname : string [10] ;
        secondname : string [10] ;
        balance : real ;
        overdraftlimit : real ;
        NextCustomer : CustomerPointer ;
    end ;
```

One of the fields in the record is a pointer to the next customer. This neat trick means that whenever we get space for another customer we also get a pointer to the next customer, and so on. The customers will be held as a linked list, with each one providing a pointer to the next and so on.

The PASCAL reserved word nil can be used to denote a null entry, and would mark the end of the list. Note that unlike an array of customers we are not able to access any particular customer directly, i.e. to

look at the fifth customer we would have to chain down via the first, second, third and fourth. An example to print out the surnames of all the bank customers would be as follows:

```
var
FirstCustomer, ThisCustomer : CustomerPointer ;
begin
ThisCustomer := FirstCustomer ;
while ThisCustomer <> nil do
begin
Writeln ( `Surname : ', ThisCustomer^.surname ) ;
ThisCustomer := ThisCustomer^.NextCustomer ;
end ;
```

Exercises

Write a program to help at the reception desk at a 50 rom hotel. The program should hold information about each room in the hotel as follows:

?? date of arrival

?? date of departure

?? name of the guest

?? billable items

Because the number of billable items cannot be predicted for each guest a single item should be held as follows:

?? Name of item

?? Cost of item

?? date and time requested

?? Is VAT required?

?? A pointer to the next item

The program should allow guests to be entered into the system, calculate their bill at the end of the stay, add billable items for a guest and list out all the occupants of the hotel.

Variant Records

Records of the same type do not necessarily have to hold exactly the same arguments. To go back to our banking example, the bank will hold records for many different types of customers, current account holders, deposit account holders, loan account holders, credit card users etc.

With the record structures we know about at the moment there are two ways that you could implement this, either you could have a different type of record for each type of account which would make it very difficult to access references to all the account holders, or you could have provision in the record for an entry for each possible account, which would waste a lot of space.

PASCAL allows you to get around this by allowing what are called variant records. These can hold different sets of fields depending on the data you wish to store in them, consider the following:

```
type
    CustomerType = ( Current, Loan, CreditCard ) ;
    Customer = record
        FirstName : string [20] ;
        Surname : string [20] ;
        Address : string [50] ;
        case AccountType : CustomerType of
            Current :
```

```
            ( Balance, OverdraftLimit : real ) ;
        Loan :
            ( LoanAmount, MonthlyRepay : real ) ;
        CreditCard :
            ( CreditLimit, Credit : real ;
            PIN : integer ) ;
        end { of case };
    end ( of record definition };
```

CustomerType is a type with three possible values, Current, Loan and CreditCard. It is used in conjunction with another form of the case construction to select a particular set of fields within the definition of the Customer type. If you wished you could add many additional values to CustomerType, and put additional clauses in the of part of the case statement to match them, so widening the range of possible accounts which can be handled.

Each record has what is called a tag field, which is used to keep track of what information the variant part of the record contains. In the case above the tag field is called AccountType and you can use this to decide how to process the record within the program:

Although the case construction used within the record definition is quite similar to the one that we are used to within our programs it is not the same. Firstly the case selector is a variable followed by a type, not a simple variable. Secondly the case construction does not have an end. This is because variant fields are always defined as the last portion of a record, with nothing following them. This means that the end marking the end of the record is also used to end the case construction used for the variant part.

All the field identifiers in the variant part must have different names, it is not possible for example to have a field called MonthlyRepayment in both the Loan and CreditCard variants.

As you might expect, PASCAL actually reserves space in the record it creates to make room for the largest of the variant sections. In the case above the record would have the size of one with the CreditCard variant, with the additional integer available in this variant not being accessable when the other variants are used.

```
var
    ThisCustomer : Customer ;
begin
    with ThisCustomer do
        begin
        if AccountType = CreditCard
        then
            Writeln ( `Credit Limit is : ', CreditLimit ) ;
```

The posh term for a record with variant types is a union type . This term is used because a record with variants is a union of two or more types. It is possible to define a record type without a key field. Records with tag fields are called discriminated unions, because something within them exists to determine what they contain. Records without tag fields are called free unions, the particular type of the record is specified when the variable is used. Consider:

```
type
    thingtype : ( int, re, bool ) ;
    thing = record
        case thingtype of
            int :
                ( intval : integer ) ;
            re :
                ( reval : real ) ;
            bool :
                ( boolval : boolean ) ;
            end { of case } ;
    end { of record definition }
```

```
var
    thingy : thing ;
```

Note that thing is a free union because only a type is specified as the case selectore, i.e. there is no named location in the record which keeps track of what it contains. The variable thingy is a free union. We can specify the fields within it in exactly the same way that we access fields in ordinary record types, and the variable will take the appropriate type, e.g.

```
thingy.re - is a real variable
thingy.int - is an integer variable
```

Once in a bright blue moon you may find a use for a free union, although the chances to use it are few and far between! They tend to be when you want to fool PASCAL into allowing you to do naughty, system type, things like access the same piece of memory in different ways.

Sets

A set is a collection of objects of a particular type. Each object in the set is said to be a member or an element of it. If S is a set of objects of type T, then any object of type T is either a member of S or not a member of S. Examples of sets are:

```
the integers 1 to 100
the numeric characters in the ASCII character set
```

Two sets are said to be equal if they contain the same elements, irrespective of their order, i.e. the sets: [1,2,3,4] and [4,3,2,1] are equal.

A set is a subset of another set if all its members are contained in the other set, e.g. [1,2] and [2,3] are subsets of the sets above.

Within PASCAL a set must be a collection of members of the same type, called the base type. The base type can be any scalar type except real. A set of items is introduced by the reserved words set of followed by the base type:

```
type
ingredients = ( apples, strawberries, bananas, nuts, icecream,
    chocalatesauce, cream, pastry, sugar, ice, fudge,
    littleumbrella ) ;
```

- ingredients is a type which can have any of the above values, and denotes a particular ingredient in an icecream sundae. This is the base type.

```
type
    sundae = set of ingredients ;
var
    knickerbokerglory, applesurprise : sundae ;
begin
    applesurprise := [ banana, nuts, icecream,
        chocolatesause, fudge ] ;
end .
```

sundae is a set type, which is the associated set type of the type ingredients. We can now declare variables of this associated set type.

These variables can hold a sub-set of the sundae set. In all there are 212 possible different sets of sundae, ranging from the empty set:

```
[ ]
```

- to the full one:

```
[ apples, straberries, bananas, nuts, icecream, chocolatesauce,
    cream, pastry, sugar, ice, fudge, littleumbrella ]
```

Within a PASCAL program the contents of a particular set are given by a list of the set contents enclosed in square brackets. Note that the surprise in the applesurprise is that there is no apple in it!

If the members of the set are consecutive values of the base type you can use double dot ".." to specify a range of items:

```
theworks := [ apples .. littleumbrella ] ;
```

Note that unlike when giving subranges, which is what you normally use .. for, this means all the items between the end points, not one item with that range of values.

Some of the standard PASCAL operators can be used with sets, although their meanings are different:

+ is used for set union, which is conventionally expressed as &. The union of two sets is a set containing the members of both sets, e.g.

```
[ apples, pastry ] + [ sugar ] =
    [ apples, pastry, sugar ]
```

* is used for set intersection, which is conventionally expressed as (. The intersection of two sets is a set which contains all members common to both sets, e.g.

```
[ bananas, icecream ] * [ apples, fudge ] =
    [ ]
```

- is used for set difference and allows you to obtain all the members of a set which are not contained in another, e.g.

```
[ bananas, fudge ] - [ fudge, apple, icecream ]
    = [ bananas ]
```

The relational operators can also be used with sets, although their meanings are also different:

= is used for set equality. When two sets are compared using this operator the resulting boolean expression is true if both sets have the same members, e.g.

```
[ bananas, fudge ] = [ fudge, bananas ]
```

- is true.

<> is used for set inequality. When two sets are compared using this operator result true is returned if both sets are different.

<= is used for is contained in. True is returned if the first set is contained by the second (i.e. the second set contains all the members of the first), e.g.

```
[ bananas, fudge ] <=
    [ fudge, apples, bananas ]
```

- is true.

>= is used for contains. True is returned if the first set contains all the members of the second, e.g.

```
[ sugar .. fudge ] >= [ apples .. fudge ]
```

- is false.

IN An additional operator is provided to allow you to test whether a particular expression of the base type is contained in a set type. This operator is IN, e.g.

```
apples IN applesuprise
```

- would, if the assignment above had been carried out, return false.

IN is the one part of sets which I make use of the most, and I use it for input verification, as in the following:

```
repeat
    Write (
        'Press command key (N, R, H) : ' ) ;
```

```
        Read ( kbd, key ) ;
until key IN [ 'N', 'R', 'H' ] ;
```

You may also find sets useful when restricting ranges of characters. They can also come into their own when you wish to keep track of a large number of possible states.

Note that parameters to procedures can be of set type so that, for example, you could write a procedure to which you pass a set of items to be looked for, or set of error types to be reported. Note also however that, as with types that we declare ourselves, the set types and their values only have meaning within the program and we have to write our own code to input and output the set values.

Within Turbo PASCAL you can have up to 255 members in a set and 255 different types of member.

Prettier Printing

Up until now when we want to print a value from a program we let PASCAL decide for us how the value is to be printed. This can lead to problems, for example if we wish to print out in columns, or if PASCAL suddenly decides to drop into exponential mode for no readily apparent reason.

PASCAL therefore allows us to specify more precisely how values are to be printed. You do this by placing formatting instructions after the name of the item to be printed, separated from the item by a colon, i.e:

```
Writeln ( fred : formatting information ) ;
```

In these descriptions of the different formats the symbols have the following meanings:

i, j, k	integer expressions
r	real expression
ch	char expression
s	string expression or array of char
b	boolean expression
^	a single space

ch:i the character ch is output, right justified in a field i characters wide, e.g.

```
Writeln ( `B':5 ) ;
```

- would produce:

```
^^^^B
```

s the string (or array of characters) s is output, e.g.

```
Writeln ( `HELLO' ) ;
```

- would produce:

```
HELLO
```

s:i the string s is output right justified in a field width of i characters, e.g.

```
Writeln ( `MUM':10 ) ;
```

- would produce:

```
^^^^^^^MUM
```

i.e. the string is preceded by i-length (s) spaces.

b if the boolean expression b is true the word TRUE is printed, otherwise the word FALSE. e.g.

```
Writeln ( (1 = 2) ) ;
```

- would produce:

FALSE

b:i the word TRUE or FALSE is output, right justified in a field i characters wide. e.g.

```
Writeln ( (1 = 1):15 ) ;
```

- would produce:

^^^^^^^^^^TRUE

i the decimal value of the integer expression i is output, in as many columns as it takes to print it. e.g.

```
I := 99 ;
Writeln ( I ) ;
```

- would produce:

99

i:j the decimal value of the integer expression i is output, right justified in a field width of j characters. e.g

```
J := 108 ;
Writeln ( J:6 ) ;
```

- would produce:

^^^108

r the value of the real expression r is output in a field 18 characters wide according to the following:

If r>0 then:

^^#.##########E*##

If r<0 then:

^-#.##########E*##

- where # means decimal digit and * means plus or minus.

for example:

```
Writeln ( 1/3 ) ;
```

- would produce

^^3.3333333333E-01

r:i the value of the real expression r is output right justified in a field i characters wide according to:

if r>0 then

spaces#.digitsE*##

if r<0 then

spaces-#.digitsE*##

- where spaces denotes a number of spaces and digits denotes a number of digits which is at least one. The number of spaces is adjusted to right justify the actual value. Note that the width must be at least 7 characters for a positive number and 8 characters for a negative one.

for example:

```
Writeln ( 1/5:10 ) ;
```

- would produce:

^^2.0E*-01

r:i:j the real expression r is output, right justified, in a field i characters wide with j characters after the decimal point. If j is given as 0 no decimal point, or characters after it, will be output.

for example:

Writeln (1/6:10:5) ;

- would produce:

^^^0.16667

Note that the last digit is rounded up automatically when the number is output.

Of Numbers and Accuracy

Computers that run PASCAL are digital devices and work in terms of finite states. This means that they are not able to hold real values to absolute accuracy (the thought of holding something like q to absolute accuracy leads to a few philosophical problems anyway!). This means that the accuracy to which computers work is limited. How much accuracy you can get varies from machine to machine, in Turbo PASCAL each real occupies 48 bits and gives an accuracy of around 11 digits.

However as we have seen above it is possible to get the printing formats to print up to 24 decimal places. This does not mean that the accuracy works to this level, rather that the printing routine will "invent" values to fill out the number. So do not think that just because you can get the computer to print values of extreme accuracy it will work to those levels. If you have cause to require high levels of accuracy you must take special steps to get them. Turbo PASCAL provides an additional compiler which stores numbers to a much higher degree of accuracy, at the expense of space and speed. Other PASCAL systems provide similar facilities.

The GOTO statement

Lovers of structured programming regard the goto statement in a similar manner to the way that sheep regard mint sauce . However, much to their annoyance, there are times when such a construction is invaluable.

Consider a situation where you are initiating a conversation with another computer, each part of the conversation is set up by a different protocol, and the conversation can fail at any time. You could write a piece of code to handle each phase and use a global variable to signal the success or failure of each phase. You would end up with something like this:

```
var
    StillTalking : boolean ;
begin
    StillTalking := True ;
    Phase1 ;
    if StillTalking
    then
        begin
        Phase2 ;
        if StillTalking
        then
            begin
            Phase3 ;
            if StillTalking
            then
                begin
                Phase4 ;
```

Our program is vanishing off the right of the paper, and when we get to the end of our conversation we have to remember to terminate all the compound statements at the right level. This construction will work, and it allows us to avoid that anathema, the goto statement. But does it really represent a good solution to the problem. Consider the alternative:

```
var
    StillTalking : boolean ;
label
        1 ;
begin
    Phase1 ;
    if not StillTalking then goto 1 ;
    Phase2 ;
    if not StillTalking then goto 1 ;
    Phase3 ;
    if not StillTalking then goto 1 ;
    1: Writeln ( `Session Failed' ) ;
end .
```

Even the most hardened anti-goto programmer would probably prefer the second version of the code!

The syntax of the goto statement is simple. A label is declared at the beginning of the program. In standard PASCAL a label can only be an integer value (this restriction seems to have been imposed in a fit of pique). In Turbo PASCAL you can use a variable name as a label as well. If you stick to using integer values however you will guarantee that the your program will work on other machines. Note that copious comments will help at this point! The keyword goto is followed by the identification of the label you wish to jump to and control is transferred to that label immediately. This allows a sudden and complete transfer of execution without having to create constructions to "drop through".

Goto Restrictions

You cannot use the goto statement to jump into the middle of a procedure or function, for obvious reasons, i.e. these must be entered via a call so that parameters and local variables can be properly set up.

You are not allowed to jump into compound statements, although you are allowed to jump out, which is what you will generally want to do.

Within Turbo PASCAL you are not allowed to jump out of procedures or functions, all jumps must take place to labels within the same program block.

Goto Philosophy

I do not use goto statements routinely, but I recognise that there are occasions when they are invaluable. If you take the approach that you will mainly use them to terminate large constructions immediately then that is probably best. You should never need to use goto constructions for the majority of your programming.

Forward Declarations

It is perfectly OK for one procedure or function to call another, however I think that you could forsee problems if two procedures wish to call each other. PASCAL does not mind this, but you will find that when you try to call the second procedure from within the first the compiler will growl at you for trying to use something before you have declared it.

This poses an interesting problem which is solved by an additional keyword, forward. This enables you to give the heading of a procedure or function (i.e. the name, the type if it is a function, how many parameters and what their types are etc.) before the actual body of the function. When the PASCAL system sees the heading it makes a note of the fact that such a procedure with the given characteristics

exists, and then allows this name to be used within code. At some later stage in the program text the actual body of the procedure will be given. You give a forward reference of this kind by replacing the body of the procedure with the keyword forward. Later on in the program you define the procedure body, e.g.

```
procedure advance ( movex, movey : real;failed : boolean ) ;
    forward ;
{ procedures and functions using advance }
procedure advance ;
    { body of advance }
```

Note that the heading information is only given once, when the forward declaration takes place. When the body is given only a minimal heading is used.

Procedures and Functions as Parameters

It is perfectly possible for one procedure or function to call another, and the above forward declaration makes this much easier. However sometimes you may not know when you write the program which function or procedure you wish to use at a particular time. If you were choosing between variables the answer would be simple, supply the variable as a parameter.

PASCAL will also allow you to pass functions or procedures as parameters, by specifying one in the parameter list:

```
function Integrate ( function f ( x : real ) : real;
                     a, b : real ) : real ;
```

Integrate is a function which returns the integral between the range a and b for the function f. Note that the parameter x to the function is a "place marker" telling PASCAL that this function has one real parameter. It is not used in the normal sense of parameters to pass information into Integrate, merely to specify how f should be called within Integrate. Once the procedure heading has been given only precisely correct invokations of the function will be correct, i.e.

```
function makereal ( i : int ) : real ;
Integrate ( makereal, 1, 2 )  ;
```

- would cause an error because the function must have a real parameter. Note that when the actual name of a function or procedure to be used in a call is given you do not give any parameters, these are supplied in calls made from within the procedure.

Graphics

Up until now we have concentrated on programs which display their output in the form of characters. It will come as no surprise to find that you can use computers to draw pictures (if it does come as a surprise the door is over there!). The standard computer graphics display can be regarded as very large array of dots, each of which can be set to a particular colour. The complete array of dots is called a raster and each individual dot is called a picture element, which is abbreviated to pell or pixel.

The number of dots which are displayed determines the resolution of the display, the more dots the greater the resolution. Note that increasing the number of dots raises the amount of memory needed to store the dots (the state of each dot in the display must be held in some way) and also gives the computer more work to do if a complex picture is to be displayed.

In order to make driving the display easier a library of routines is usually used to drive the display, allowing lines and characters to be drawn. The software may also provide windows which can be regarded as separate "mini-screens". The windowing software will allow these windows to be driven from your program irrespective of their size or position on the screen.

Graphics Standards

Whilst there are established standard programming languages which can be run on a variety of computers we are still waiting for a graphics standard to appear. This means that any code you write to drive a display will be specific to the machine and software you are using at the time, and will need to be converted for another. However, languages like PASCAL do provide a way out of this dilemma in that you can split off the part which does the drawing from the code which does the work. If you do this you can easily move code from one machine to another by just re-writing the machine specific code. It also helps if you avoid any fancy routines which are provided by a particular system which may not be available elsewhere. (I usually stick with simple "move and draw" commands.)

When you wish to move around your graphics area you do so by giving x and y values to, for example, identify the start and end of a line you wish to draw. Usually the values being plotted will be specific to your application, for example temperatures between 0 and 50 degrees. Most graphics software allows you to scale the movement within your window to reflect the application so that for our temperatures 0 could be the bottom of your graph and 50 the top. This allows you to forget about the actual size and resolution of the window you are drawing, indeed you would be able to change the size of the window at any time. If you suspect that you may have to transfer your program to a very limited graphics system you should do the scaling within your program by declaring some constants at the beginning which are then used to scale your plots.

A number of graphics standards are now emerging which incorporate windowing, on prospective one for workstations is called XWINDOWS. This also caters for the output of different processes which are send to their own windows.

Text and Graphics

If you are using an IBM computer in text mode you can only write characters of a particular size in particular positions on the screen. This is because of the way that the text display works, in effect a code is stored which tells the machine to display a particular shape at that position. If you are drawing your text on the screen with a graphics package you can position text anywhere on the screen and in any size. However the graphics system does need to know the shape of the characters so that they can be drawn. Turbo PASCAL supports two ways of drawing text on the screen, bit mapped and stroked

Bit Mapped Fonts

A bit mapped font is made up of a series of dots on a matrix. If you look closely at the display on a machine you will see that the characters are actually made up in this way. This form of font is OK for small characters but if we make the characters bigger without increasing the number of dots, (i.e. simply by making the dots bigger), the characters start to look poor.

Stroked Fonts

A stroked font adopts a different approach to character drawing. Instead of a series of dots giving the pattern of a character a stroked font is stored as a number of drawing actions which, when performed, draw the required character. In effect the character is stored as a number of co-ordinates, along with information about whether they are joined with straight or curved lines. This means that characters can be enlarged simply by scaling the drawing operation. Furthermore, larger characters do not suffer any drop in quality.

The only difficultly with stroked fonts is that they can sometimes take longer to draw, because of the calculations involved in putting them on the screen. Also, because usually only the outline of the character is stored, they are sometimes not drawn as solid. Turbo PASCAL provides a number stroked fonts. These are loaded in the same way as the BGI files (see later), they have the extension CHR.

Graphics in Turbo PASCAL

Turbo PASCAL includes a library of procedures which allow you to produce graphics. These are supplied in the GRAPH unit. They also need access to the graphics interface program which is used when your drawing program runs. There is a different interface program for each type of display system, the display on our machine is the Hercules so we need the file HERC.BGI to be present on the current disk when our graphics program starts.

The Co-ordinate System

The origin of the screen is the top left hand corner, how far you can draw in the X and Y directions depends on the graphics adapter you are using. There are routines which will tell you the maximum X and Y values for your current display so that you can work out scaling, I would advise you to use these for all your programs so that they will draw graphs of the same size on all displays.

The two routines are called GetMaxX and GetMaxY, they return values of type word. The word type is identical to an integer, except that it can only contain positive values. This allows a greater range to be encompassed.

Starting Graphics

You begin graphics by calling the InitGraph procedure. This finds out what kind of display you have, loads the appropriate display driver and then puts the screen into graphics mode. Note that for this procedure call to work you must have the appropriate .BGI file available:

```
program sillytest ;
uses
     graph ;
var
     GraphDriver : integer ;      { current graphics driver }
     GraphMode : integer ; { current graphics mode }
     ErrorCode : integer ; { error code from routines }
begin
     GraphDriver := Detect ;
     { Detect is a constant defined in the graph module. }
     { It tells InitGraph to find out for itself what }
     { driver is present. If you want to force the use }
     { of a particular display card you  can set this to }
     { a fixed value, e.g. CGA or EGA or HercMono. }
     InitGraph (GraphDriver, GraphMode, '') ;
     { The first parameter gives the display type, the }
     { second the mode and the third a path to the .BGI }
     { files. If you put nothing in the path  the system }
     { looks at the current disk. }
     ErrorCode := GraphResult ;
     { GraphResult is a procedure which will find out }
     { how the previous graphics call got on. If it did }
     { not work you can find out why from the value }
     { you get back. }
     if ErrorCode <> grOK
     then
          begin
          { grOK is a constant which is set to OK value }
          { if we get something different back we decode }
          { the error message using GraphErrorMsg: }
          writeln ( 'Graphics error : ',
                    GraphErrorMsg ( ErrorCode ) ) ;
          writeln ( 'Program aborted' ) ;
```

```
            halt (1) ;
            end ;

        { Now, with things safely set up we can draw }
        Rectange (0, 0, GetMaxX, GetMaxY) ;
        Readln ; { Wait for a keypress }
        CloseGraph ; { shut down graphics }
end.
```

Index