Course Notes For PASCAL

Advanced Level

Table of Contents

Chapter 1.

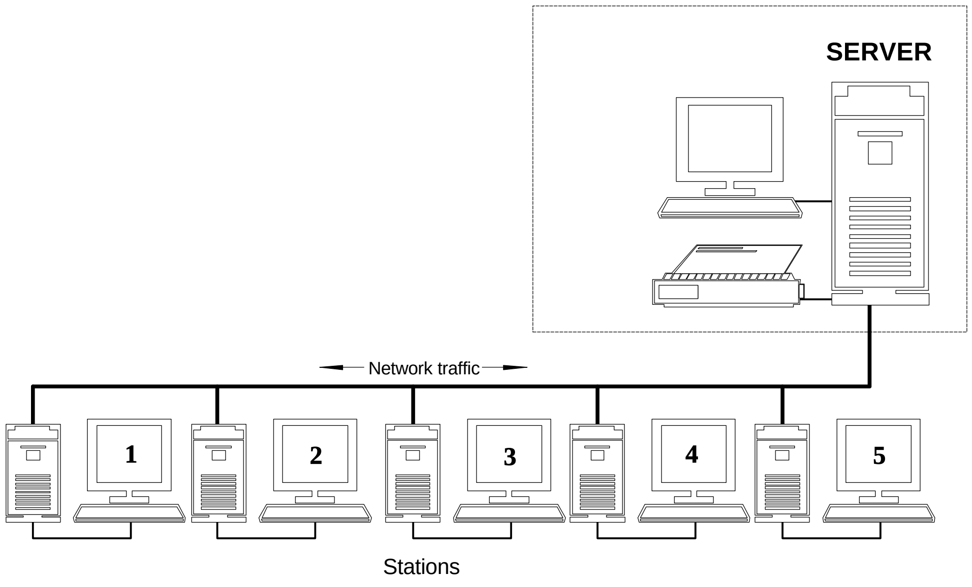
1. Using the Network

OBJECTIVES:

- In this short chapter you will learn how to work on the network - specifically how to boot, login and out, and assign yourself a private password.

1.1. The Network

Throughout this course, you'll be working on a 5-station network.



The network server is equipped with a large harddisk providing centralized shared storage for the stations. Local stations have only a single 3½ inch disk drive, used for the permanent storage of your own work. Each authorized user (and that includes you) has a private home directory on the network disk. The user has complete rights in his/her home directory, and read-only or execute-only rights in other directories containing application software or shared data.

1.2. Getting Started

To use the network, follow these instructions:

1. **If the station is switched off**, switch it on. The stations boot from an internal boot rom, but if something goes wrong you may have to use a network bootdisk obtainable from your teacher or from the administration personnel in the office, otherwise

2. Type **LOGIN** at the keyboard.
3. You will be asked for your user's name. This is a public name assigned by the network administrator to each authorized user, and usually consists of an 8-character code as follows:

   AC95<first three letters of your surname><first letter of your name>

   Thus for example, if your name is Mark Borg, your user name would be AC95BORM. The AC95 prefix indicates that you are a member of the Advanced-level Computing 1995 group.

At this point you should receive a login message. Check the system prompt to make sure that you are in your home directory (which has the same name as your user name). You should now assign yourself a private password. This password is known only to yourself, and ensures that nobody else has access to your private directory. Do this by using the network **SETPASS** command. Since the password you type in is not echoed to the screen, you will be asked to enter it twice for validation. **Make sure you choose a password which is easy to remember**. The next time you log in, you will be asked BOTH your user name AND your password **-** if you forget either of them you cannot use the network (and probably you'll be the laughing stock of the class!).

When you have finished using the network, **do not switch off -** instead use the **LOGOUT** command to sign off.

1.3. Your home directory

You can think of your home directory as an area of the server's disk which has been set aside exclusively for your own use. This is the directory where you start when you log in, and this is where you save all your files. Nobody else can use this directory, and therefore your files are safe from other users. However,

☞ **YOU ARE EXPECTED TO COPY ALL YOUR FILES ONTO YOUR OWN FLOPPY DISKETTE AT THE END OF EACH LESSON.**

You must do this not only as a safety precaution (in case of a hardware fault on the network drive), but also because you are expected to continue working at home. Note that each user's home directory has a maximum capacity of 2M.

1.4. Running programs

All the programs you will need for this course are already installed on the network. You are expected to use these installed programs, not your own. We must emphasize that

☞ **NO ONE IS ALLOWED TO RUN ANY SOFTWARE FROM A FLOPPY DISKETTE, OR TO INSTALL ANY SOFTWARE ON THE NETWORK.**

CHAPTER 2.

2. MSDOS

An Operating System provides the interface between the user and the raw hardware. Most IBM-compatible PCs run under the **Microsoft Disk Operating System - MSDOS**. This is a very simple single-user, single-programming operating system. These notes give a very brief introduction to MSDOS 5.0. Most of what we say here applies equally well to other versions of MSDOS.

In what follows, computer output is shown in **`bold`**, while user input is shown in `normal text`. MSDOS is **case insensitive**, which means that user input may be in either UPPERCASE or lowercase **-** no distinction is made. **REMEMBER to press the ENTER KEY to terminate your input**.

Your computer will come equipped with one or more disk drives. Each disk drive has a single letter name. The first drive is called A:, the second B:, and so on. Drives C: onwards usually represent hard-disk drives. On networked systems, drives F: onwards usually represent network drives (non-local drives) **-** i.e. drives which are shared among users.

The letter shown in the MSDOS prompt is the letter of the **CURRENTLY LOGGED DRIVE -** A>, B>, etc. To log onto a different drive (assuming your computer has more than one drive), type the drive name:

```
A> u:
U>
```

2.1. Listing the contents of a disk

The list of contents of a disk is called a **DIRECTORY LISTING**. You can get a directory listing by issuing a **DIR** command. For example, you would use

```
U> dir A:
```

to list the contents of the disk in drive A:. If you do not specify a drive letter, DIR will list the contents of the currently logged drive.

2.2. Directory structure

Files on a disk are stored in **DIRECTORIES.** You may think of directories as FOLDERS. Directories may be nested, forming a **DIRECTORY HIERARCHY -** i.e. directories may not only contain files, but also **SUB-DIRECTORIES**.

Every disk has at least one directory, called the **ROOT**, which is the parent of all files and directories on that disk. This directory is automatically created for you when you **FORMAT** the disk. The root directory is denoted by the backslash character \. Thus **A:\** denotes the root directory of the disk in drive A:

```
U> dir F:\
```

lists the contents of the root directory of drive F. The hierarchical structure of directories, starting with the root directory and going down into directories, subdirectories etc., is sometimes called the **DIRECTORY TREE**.

2.3. Directory and filenames

Names of directories and files (which we will collectively call **NAMES**) in MSDOS may be from 1 to 8 characters long, and may NOT contain SPACES or most special characters (such as punctuation). To be on the safe side, use only letters and digits for names.

In addition, a name may have an **EXTENSION** of from 1 to 3 characters, which must be introduced by a **FULLSTOP** (directory names can, but are usually NOT, given an extension). The EXTENSION is meant to **GIVE SOME INDICATION OF THE FILE TYPE**.

The following are all legal NAMES:

```
DIR1
LETTER.1
PROG.PAS
```

2.4. Reserved and other extensions

Some extensions are **RESERVED** by MSDOS to indicate special files. These are:

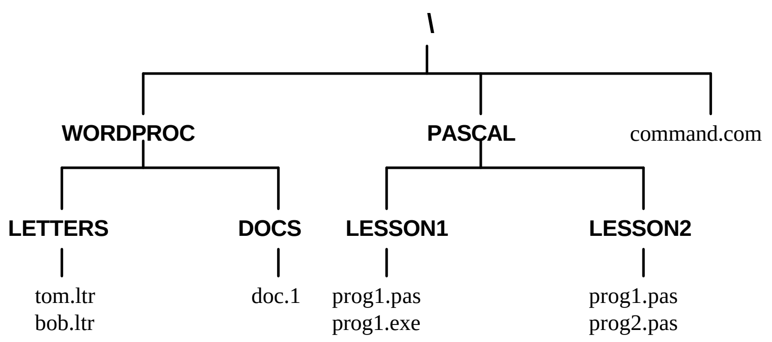
.SYS	*an operating system file*
.EXE	*an executable file (can be run by typing its name at the MS DOS prompt)*
.COM	*also an executable file*
.BAT	*a BATCH file. Batch files contain sequences of MSDOS commands which can be executed by merely invoking the batch file.*

Other extensions, while not reserved, are conventionally associated with certain types of files, for example:

.BAK	*a backup file*
.BAS	*a BASIC source file*
.PAS	*a PASCAL source file*
.ASM	*an ASSEMBLER source file*
.DOC	*a word-processor document file*
.TXT	*plain ASCII text file*
.ZIP	*a set of files compressed using the PKZIP utility*
.ARC	*a set of files compressed using the PKARC utility*
.TPU	*Turbo Pascal Unit file*
.TMP	*a temporary file*

2.5. The directory tree

Consider the following directory tree:



A directory tree

Trees in Computing differ from those in nature in that they have their roots sticking up into the air!

The **ROOT DIRECTORY** \ contains 3 items: two directories (WORDPROC and PASCAL), and a file (COMMAND.COM). The WORDPROC subdirectory itself contains two subdirectories **-** LETTERS and DOCS, but no files. And so on.

When you insert a disk into a drive, the **ROOT** directory of that disk becomes the **DEFAULT DIRECTORY OF THE DRIVE**. If you **DIR** the disk, you will only see the items in the root directory. You can change the **DEFAULT DIRECTORY** (or **CURRENT DIRECTORY**) by using the MSDOS command **CD,** for example

```
A> cd PASCAL
```

This changes the **DEFAULT DIRECTORY** to the PASCAL subdirectory. If you do a DIR now, you will only see the 2 items LESSON1 and LESSON2. You may, if you wish, again CD to one of these directories.

2.6. Pathnames

Consider file PROG2.PAS. From the directory tree you can see that PROG2.PAS is in directory LESSON2, which is itself in directory PASCAL, which is itself in the root directory \, which is on the disk in drive A: (for example). We say that the **FULL PATHNAME** of the file PROG2.PAS is

```
A:\PASCAL\LESSON2\PROG2.PAS
```

Note how the \ character NOT ONLY denotes the ROOT directory, but is also used as a **NAME SEPARATOR** in writing out the path of an object on the disk.

2.7. Relative pathnames

The full pathname is an **ABSOLUTE** pathname **-** it completely and unambiguously specifies where a file is located, whatever your default directory happens to be.

Pathnames may also be specified **RELATIVE TO YOUR DEFAULT DIRECTORY**. For example, assume your default directory is currently A:\, then the pathname of the file PROG2.PAS is

```
PASCAL\LESSON2\PROG2.PAS
```

Assume that your default directory is **A:\PASCAL** (i.e. you did a CD PASCAL), then the location of PROG2.PAS relative to the current default directory is

```
LESSON2\PROG2.PAS
```

2.8. Moving about the directory tree

Suppose your default directory is A:\, and you want to change to directory LESSON1. You can do this as follows:

```
        A> cd PASCAL
        A> cd LESSON1
```

The same effect may be achieved using a SINGLE CD command:

```
        A> cd PASCAL\LESSON1
```

The root directory is the **PARENT** directory of both WORDPROC and PASCAL. PASCAL is the **PARENT** directory of both LESSON1 and LESSON2. The special directory name **..** represents the **PARENT** of a subdirectory (note that root has no parent directory). Thus, doing a

```
        A> cd ..
```

will change the default directory to the PARENT of the current default directory. For example, if your default directory was LESSON1, the CD .. command will change to directory PASCAL, since this is the parent directory of LESSON1. Thus the effect of **CD ..** is to **MOVE UP ONE LEVEL** in the directory tree.

✎ Draw a simplified directory tree of the network drive U:. This is the drive containing all users' home directories. Note how the tree is structured, and learn to find your way around. What happens if you try to list the contents of somebody else's home directory? Why is this?

2.9. Creating and deleting directories

To create a new directory, use the **MD** (Make Directory) command. For example:

```
        MD A:\PASCAL\MYPROGS
```

will create a directory MYPROGS in directory PASCAL on the disk in drive A:. The directory will initially be empty, of course. You may use relative pathnames. For example, MD MYPROGS will create MYPROGS as a subdirectory of your current default directory.

You may delete a directory by using the **RD** (Remove Directory) command, for example:

```
        RD A:\PASCAL\MYPROGS
```

The directory to be deleted **MUST BE EMPTY**.

✎ Starting with chapter 3, you will be writing Pascal programs which you will need to save to disk. During the lesson you will save these programs in your home directory, and at the end of the lesson you should copy them onto your floppy diskette. Rather than saving all files in one directory, it is best to organize things by creating a subdirectory for each chapter. Create subdirectories in your home directory called CHAP3, CHAP4 and CHAP5. Later, you can create subdirectories for the other chapters as well.

2.10. Wildcards

A **WILDCARD** is a character which may represent any other character (like a Joker in a card game). Some MSDOS commands will let you use wildcards in filenames to specify GROUPS of files, rather than single files. There are 2 permissible wildcard characters:

*	*This represents a SEQUENCE OF 0 OR MORE CHARACTERS.*
?	*This represents any SINGLE character.*

EXAMPLES:

.	*Represents all files with ANY name and ANY extension*
*.pas	*Represents all files with ANY name and an extension of .PAS*
*.?	*Represents all files with ANY name and ANY SINGLE-CHARACTER EXTENSION*

We can use wildcards to good effect when requesting a directory listing, for example:

```
DIR pascal\lesson1\*.bak
DIR prog1.*
```

2.11. Command-line switches

Many MSDOS commands can take one or more command-line switches which change the way they work. A command-line switch is usually a single letter preceeded with a / character. Most commands in MSDOS 5.0 accept a help switch, /?, which lists help about that particular command. For example DIR /? will explain what DIR does, and list all the command-line switches it accepts.

Among the switches accepted by DIR are /p, which causes the directory listing to pause after every screenful of data, and /w, which lists the directory in a wide format. These two switches are useful when the directory being listed contains many files, since otherwise the listing will scroll off the screen before you get a chance to view it. Thus we can issue a dir command as follows:

```
DIR A:\LESSONS\*.PAS /P
```

MSDOS also has a HELP command. Typing HELP by itself produces a list of all MSDOS commands. Typing HELP followed by a command name, for example

```
HELP DIR
```

has the same effect as using the /? switch with the command.

NOTE the difference between \ (backslash) used as a directory separator, and / (slash) used to specify command-line switches.

2.12. Output redirection

Most DOS commands send their output (for example the directory listing produced by DIR) to the STANDARD OUTPUT DEVICE. The standard output device is by default the screen, but you can REDIRECT this output to any other valid output device. Valid devices include the disk (ie a file) and the printer. When you do this, the output produced by the command goes into the output device you specify instead of going to the screen.

To redirect output, you use the > character, followed by the name of the device where the output should be sent. The device name can be either a filename or PRN (the printer). Thus we can do the following:

```
DIR A:\*.* >PRN
```

which sends the directory listing to the printer - assuming a printer is connected and online.

```
DIR >LIST.TXT
```

which sends a listing of the current directory to the file LIST.TXT. Careful though, if a file called LIST.TXT already exists, it will be overwritten. Similarly, the following sends the help information about the FORMAT command to a file HELP.TXT.

```
FORMAT /? >HELP.TXT
```

This could be a very useful way for compiling a help file with information about many MSDOS commands - just call all the commands you want help about with the /? switch, and collect the information in a single file, which you could later print in the form of a manual, or load into a wordprocessor for formatting, etc. The problem is, of course, that every command's help output will overwrite all previous output. To overcome this problem, you can use the >> output redirector. This works like >, but instead of overwriting the file you specify, it appends the new information to it. Thus, doing:

```
DIR /? >>HELP.TXT
FORMAT /? >>HELP.TXT
```

will produce a file HELP.TXT containing information about BOTH DIR and FORMAT.

NOTE that >> will append output to the specified file if the file already exists. If the specified file does not exist, it will simply create it (in this case it behaves just like >).

2.13. Copying files

To copy files, you use the COPY command. This command usually takes two arguments - the name of the file you want to copy (the SOURCE), and where to place the copy (the DESTINATION). Thus

```
        COPY PROG1.PAS PROG1.BAK
```

will make a copy of PROG1.PAS and call it PROG1.BAK. Keep in mind that this command makes a COPY of a file - the original (in this case PROG1.PAS) is not changed in any way. You can specify a full pathname for the destination file if you want the copy to be placed in a different directory or on another disk.

The SOURCE argument can specify multiple files by using wildcards. Thus

```
        COPY U:*.PAS A:\PROGS
```

will copy all files in the default directory of drive U: with extension .PAS, and place them in subdirectory PROGS of drive A:

✎ Your home directory is a subdirectory of ALEV95. This directory contains all the home directories of students in the 1995 A-Level group. This directory also contains a subdirectory called COMMON. **Common** contains files which can be read by all users in group ALEV95. Amongst other things, it contains a subdirectory called CHAP3, which holds some files you will be using in the next chapter. Make sure you are in your home directory, and copy all the contents of COMMON\CHAP3 into your own CHAP3 directory.

Another way of copying files is by using the **XCOPY** command. Besides being faster than COPY for copying multiple files, XCOPY can also copy subdirectories if you specify the /S switch. Suppose you wanted to copy all the files in your home directory, together with all its subdirectories and THEIR files and subdirectories (sometimes called a **BRANCH** of the directory tree), onto the disk in drive A:. If you are in your home directory, you could do this with XCOPY as follows:

```
        XCOPY *.* A:\   /S
```

2.14. Other common MSDOS commands

The appendix gives an overview of the most common MSDOS commands. You should, however, aim to become fluent in using MSDOS - use the /? switch to learn more about commands.

✎ Using the /? switch or the HELP command, and redirecting output with >> , produce a file MANUAL.TXT with information about the following commands:

CLS	Clears the screen.
COPY	Copies one or more files to another location.
DEL	Deletes one or more files.
DIR	Displays a list of files and subdirectories in a directory.
DISKCOPY	Copies the contents of one floppy disk to another.
FORMAT	Formats a disk for use with MS-DOS.
MD	Creates a directory.

CHAPTER 3.

3. Using the IDE

In this chapter you are introduced to the Trubo Pascal Integrated Development Environment - the set of software tools you will be using to create Turbo Pascal programs. You will need the file TABLES.PAS which you copied into your CHAP3 directory in the previous lesson (this file will aslo be required for the next chapter).

3.1. Running Turbo Pascal

To run Turbo Pascal 5.5 (TP5) simply type in **PAS5**. This invokes a **batch file** which runs Turbo Pascal for you. TP5 is a complete **integrated development environment (IDE)** - which means that it integrates the following modules in a single package:

1. Text editor for writing program source.
2. Online, context-sensitive help system containing help about every Pascal construct and command, all standard libraries, the editor etc., as well as example programs which help to illustrate the use of all Pascal constructs and library procedures.
3. File librarian for loading and saving source files and navigating directories.
4. Compiler for compiling the source program to an object program.
5. Linker for linking object modules into a single executable program.
6. Loader for running executable code.
7. Source-level debugger for correcting runtime errors in your programs.

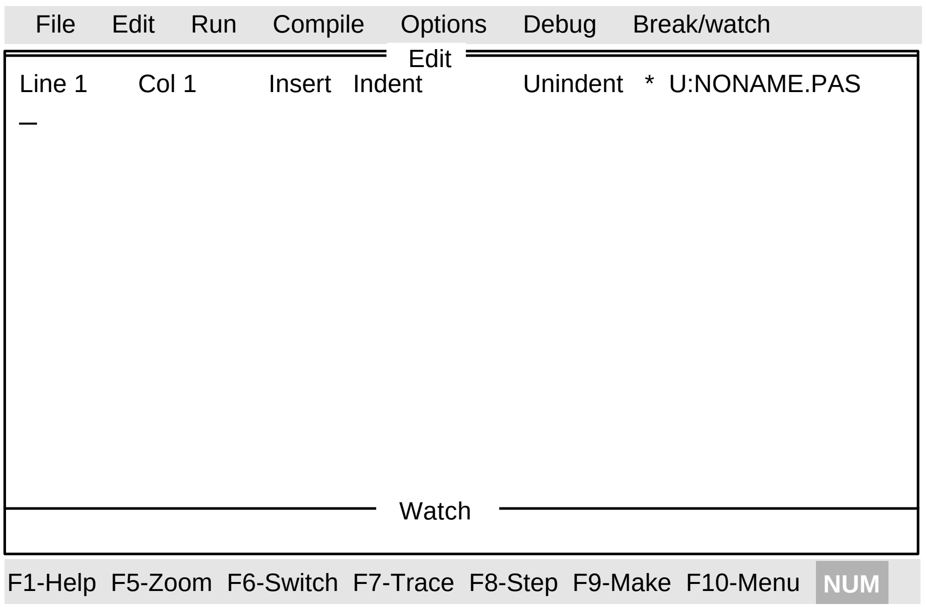
The whole environment is menu-driven - you issue commands by selecting items from the menu bar at the top of the screen.

3.2. The Turbo Pascal Environment

The main TP5 screen consists of 4 basic areas, as follows:

1. **The Menu Bar**. This gives access to 7 main menus items, most of which can be pulled down to reveal further items.
2. **The Edit Window**. This is where all file editing takes place. The first line of the edit window is a status line providing (among other things) the following information:
   - Cursor position (line and column)
   - Insert/overwrite indicator, indicating the current edit mode. The edit mode can be toggled using the Insert key.
   - The file-modified indicator (an asterisk). This indicates that the file in the edit window has been modified, and is a kind of warning to remind you that you should save the file.

- The filename. This is the name of the file being edited. If the file being edited is a new file, and therefore does not have a name yet, then the default NONAME.PAS is displayed.


File Edit Run Compile Options Debug Break/watch
Edit
Line 1 Col 1 Insert Indent Unindent * U:NONAME.PAS
Watch
F1-Help F5-Zoom F6-Switch F7-Trace F8-Step F9-Make F10-Menu
NUM

3. **The Watch Window**. This window is only used when debugging a program, and is not required otherwise. You may prefer to hide this window in order to make the edit window larger. You can do this by pressing function key F5 (zoom), which hides and shows the watch window.

4. **The Information Line**. This lists the most common hotkeys you can use. A **hotkey** is a function key or a combination of 2 keys which you can press to perform an action. Thus, to zoom the edit window and hide the watch window, you can press F5. There are many more hotkeys than are listed in the information line. If you press and hold down the CTRL key or the ALT key, a different list of hotkeys is displayed. The right side of the information line shows (in inverse video) the state of the keyboard's toggle keys - the CAPS LOCK, the SCROLL LOCK and the NUM LOCK. In the above diagram only the NUM LOCK is on, the other two toggle keys being off.

   NOTE hotkeys are sometimes also called **shortcut keys** or **accelerator keys**.

3.3. Using the Menus

To make a selection from the menu bar, press F10, use the left and right arrow keys to move to the desired menu, then press ENTER to pull down the menu and make your selection.

A faster method is to use ALT-<Letter>, where <Letter> is the first letter of the menu you want to access (usually shown in red). Thus ALT-F takes you to the file menu, etc. Then use the up and down cursor keys to move to the desired item, and press ENTER. To exit from the menu and return to the edit window, use ESC.

Many menu items have a hot key associated with them - you can use this instead of selecting the item from the menu.

3.4. Using the Online Help

TP5 has an extensive online, context-sensitive hypertext help system. You access help pages by using the F1 key, as follows:

1. **F1** gives **context help** - i.e. help relevant to what you are currently doing. Thus, pressing F1 when in the editor gives help on editor commands, and pressing F1 when in a menu gives help about the current menu item.
2. Pressing **F1 twice** takes you to a **contents help page** (help index), from where you can select a topic about which you require information.
3. Pressing **ALT-F1** displays the last help page.
4. From the editor, pressing **Ctrl-F1** brings up help about the item over which the cursor is positioned. For example, if the cursor is over the word WRITELN (which is a standard Pascal procedure), pressing Ctrl-F1 gives help about the syntax of this procedure.
5. Some help pages contain a link to an **example page**. Example pages contain small example programs demonstrating the use of a Pascal command or construct. You can run these example programs as follows
   - Press **C** (for Cut) - the cursor will appear in the help page.
   - Press **B** (for Begin), and move the cursor with the cursor (arrow) keys to drag a rectangle around the example.
   - Press **ENTER** - the marked section of text from the example page is copied into the editor at the position of the cursor. You can then compile and run the example program.

3.5. Loading and Saving Files

In the previous chapter, you created a directory called **chap3**, which contains a Pascal source file called **tables.pas**. To load this file into the editor, proceed as follows:

1. Go to the file menu (ALT-F).
2. Select **LOAD**. You will be asked for the extension of the file you wish to load - press ENTER to accept the default extension (PAS). You could also use the hotkey F3 to perform the same operation. Note that if you are already editing a file and that file has been modified, TP5 will ask you whether you want tosave it before loading a different file.
3. From the file requester, select the directory CHAP3, and then move the cursor to the file TABLES.PAS and press ENTER to load it.

Tables.pas prints the 3-times table from any number to any number. It is not important to understand how the program works at the moment - it will be explained in the next chapter.

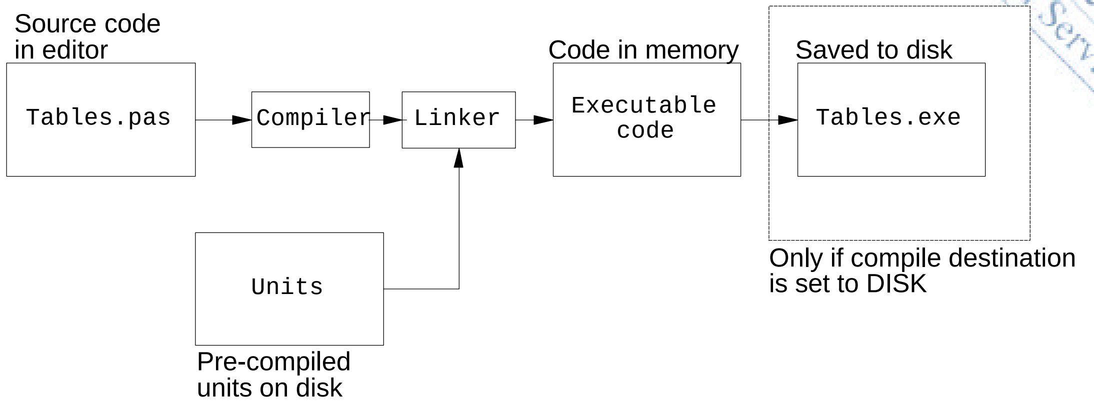
To save a file, choose save from the file menu, or use the accelerator key F2. To save the file under a different name, choose **WRITE TO** from the file menu.

To start a new file, choose **NEW** from the file menu.

3.6. Compiling a Program

It is very important to understand not just HOW to compile a program, but also WHAT HAPPENS when you compile a program. The following diagram shows an overall view of the process:

Compiling a Program using the TP5 IDE



Strictly speaking, converting a source program (e.g. tables.pas) to an executable program (e.g. tables.exe) is a 2-stage process:

1. A **COMPILER** checks the correctness of the source program and, if no errors are found, converts the source code to OBJECT CODE. This object code by itself is incomplete because a program typically makes **use** of LIBRARY ROUTINES which are in separate files. In TP5 these files of library routines are called UNITS.
2. A **LINKER** takes the object code generated by the compiler and appends to it (LINKS) all the library routines required to make it a complete program. This complete program is the EXECUTABLE FILE, which can be executed by the CPU. The linker also performs other functions which do not directly concern us here.

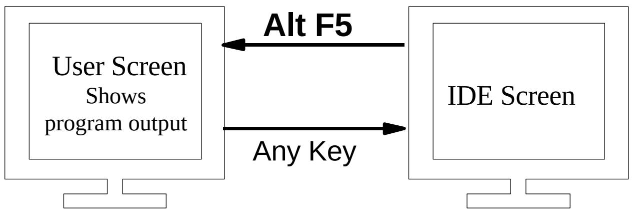
Because TP5 provides an integrated environment, this 2-stage process happens automatically when you issue a compile command. To do so, choose **compile** from the **Compile** menu (ALT-C). If the compiler encounters a **syntax error**, you will get an error message explaining what was wrong, and the editing cursor will be positioned close to the source of the error. Pressing F1 will give further help about what went wrong. Once a program has compiled successfully, it is good practice to save it.

TP5 also offers you the option to either leave the executable code in memory, or to actually save it to disk as an .EXE file. You choose which from the **destination** item of the **Compile** menu. If you save the executable file to disk, you will be able to run your program from MSDOS without having to compile it every time.

3.7. Running Programs

To run the compiled program, choose **run** from the **Run** menu. The screen will switch to the **user screen**, and the program will be run. When the program terminates (whether normally or because of a **runtime error**), you are returned to the IDE.

It is important to understand this 2-screen system. Although the computer has only one **physical screen**, TP5 maintains two **virtual screens -** the IDE screen which shows the TP5 environment, and the user screen which shows your program's output. Only one of these virtual screens can be visible at any one time.



To switch to the user screen from the IDE to see the output of your program use **ALT F5**. To switch back to the IDE press any key.

3.8. Editing Source Code

Editor commands (such as moving to the top or bottom of a file, marking a block, etc) are not available in a menu and can only be accessed from the keyboard. The appendix lists the most commonly used editor commands. The online help gives a complete listing of these commands.

> Locate the TABLEOF constant definition in the program tables.pas and change it to the value 8. Recompile and run the program to generate the 8-times table.

3.9. Tracing a Program

Tracing refers to the process of running a program line by line. It is usually used when debugging a program which has runtime errors and you want to see exactly what the problem is. However, tracing can also give you useful insights into how Pascal works, and will teach you many things about the language.

TP5 uses key F7 for tracing. Pressing F7 starts the tracing process. The first line of the program is highlighted showing that it is about to be executed. Pressing F7 again executes the highlighted line and moves the highlight to the next executable line, waiting for you to press F7 again to execute it. While tracing, you can do any of the following:

1. View the user screen (ALT F5).
2. Inspect the contents of any variables or expression. To do this, move the cursor to the name of the variable you wish to inspect, and press CTRL F4. A dialog box pops up with the name of the variable. You can either confirm that this is the variable you want to inspect by pressing ENTER, or type in the name of another variable. The contents are displayed in the middle box of the dialogue. Press escape to put the dialog away.
3. Watch one or more variables. To do this make sure the watch window is visible (use F5), and select add watch from the Break/watch menu (hotkey CTRL-F7). In the requester which pops up, enter the name of the variable you want to watch - the variable then appears in the watch window. You can watch more than one variable - the watch window will expand to accommodate all of them.

You can also set breakpoints on program lines instead of tracing a whole program line by line. The program can then be run normally, but when it reaches a line which has a breakpoint it will stop and return you to the IDE. You can then inspect variables, continue tracing forward from the breakpoint, etc. To set a breakpoint, move the cursor to the required line (which MUST be an executable line) and select **toggle breakpoint** from the **Break/watch** menu (hotkey CTRL-F8).

CHAPTER 4

4. The Structure of a PASCAL Program

4.1. Example of a Simple PASCAL Program

Consider the following PASCAL program, a simplified version of the tables program from the previous chapter. Although you are not expected to understand much of it at first, it should give you an idea of the STRUCTURE of a program in PASCAL. If you already know another programming language (such as BASIC), it would be helpful to note differences between these two languages.

```
PROGRAM tables;
USES Crt;

CONST
   TableOf = 3;

VAR
   num1, num2, i : INTEGER;

  { This function multiplies two integer numbers and returns their
    product.}

    FUNCTION ProductOf (num1,num2:INTEGER) : INTEGER;
    BEGIN
       ProductOf := num1 * num2;
    END; {Function ProductOf}


{ --- Main program starts here ------------------------------------------- }

BEGIN
   ClrScr;                               {Clear the screen}
   WRITELN ('Table of ',TableOf);
   WRITE   ('Enter start:');             {Prompt & input start}
   READLN  (num1);
   WRITE   ('Enter end:');               {Prompt & input end}
   READLN  (num2);
   FOR i := num1 TO num2 DO              {Use a loop to print out the table}
   BEGIN                                 {within the given range}
      WRITELN (i,' x ',TableOf,' = ',ProductOf(i,TableOf));
   END;
END. {Program tables}
```

NOTES :

1. The program starts with a header which consists of the reserved word **PROGRAM** followed by the name of the program.

2. Next comes a **USES** clause, listing all the library units the program uses. In this example, the program is only using one unit **-** the CRT unit which contains functions and procedures for screen display (e.g. ClrScr). TP5 has 7 standard library units, but the CRT unit is the one which every program will need to use most frequently. Later on, you will also be able to write your own units for use in your programs.

3. The next section of the program is the declaration of **CONST**ants. Constant values to be used by the program (e.g. Maximum mark for an exam) should be declared here and be given a meaningful name (eg MaxMark). Thereafter, the program should refer to the constant by its name and not by its value. This has two distinct advantages:

   (i) *it makes the program more readable,*

   (ii) *if the value of the constant is to be changed, one has only to change its declaration rather than change every occurence of the constant within the program.*

4. The declaration of **VAR**iables to be used by the program now follows. Unlike **BASIC**, all variables have to be declared before they can be used. A variable is declared by means of an **IDENTIFIER** (its name) and a **TYPE-SPECIFIER** (to indicate what type of data it will store). In the **VAR** declaration, the line

   number_of_students, mark, average_mark **: INTEGER**;

   would serve to declare three variables called **number_of_students**, **mark**, and **average_mark** each being of type **INTEGER**. For the time being, we shall introduce the following 4 types:

INTEGER	Can store an integer number (+ve or -ve) within a limited range.
REAL	Can store a real number (may include fractional part) within a limited but much wider range than above. Stored in memory as a floating point number.
CHAR	Can be used to store a single character (occupies one byte).
STRING	Can be used to store a string of zero or more characters. The maximum length of the string must be declared and must be an integer constant (it may be useful to declare the string's maximum length in the **CONST** declaration).

Example:

```
CONST
   max_name_length = 30;
VAR
   name : STRING [max_name_length];
```

5. Next, **subprograms** are declared. In the example, only one subprogram, a **FUNCTION** called **ProductOf**, is declared. This function takes two numbers and returns their product as its result. The program tables.pas from the previous chapter also contains an example of a **PROCEDURE** declaration **-** a procedure is another type of subprogram.

6. The actual program instructions start after all the declarations. Note that these instructions are enclosed within **BEGIN** and **END** statements. Furthermore, instructions may be enclosed within nested **BEGIN ... END** statements as can be seen above (we will see the reasons for this later). Consequently, it is good practice to indent the program statements according to their level of nesting. This is allowed because Pascal is a free format language. Note that both upper and lower case characters may be used. No distinction is made between the two. Note also that each statement is terminated by a semi-colon '**;**'. This also holds for **END** in some cases but the **END** statement at the very end of the program *must* be followed by a fullstop.

7. Comments can be placed anywhere in the source program and must be enclosed within curly brackets **{** ... **}** or **(*** ... ***)**.

4.2. Structure of a PASCAL Program

To summarize then, a Turbo Pascal 5 program has the following structure, made up of 4 main sections:

PROGRAM HEADING	Giving the name of the program.
UNITS TO BE USED	A list of all the units required by the program.
DECLARATIONS	Declarations of all objects to be used by the program. These include: **Constants** **Types** **Variables** **Subprograms** (functions and procedures) All these declarations are called GLOBAL declarations.
MAIN PROGRAM	This is where execution starts. It has the following syntax: **Begin** *statements* **End.**

4.3. Data Output: WRITE and WRITELN

SYNTAX **:** **WRITE**(<expression 1> , <expression 2> , ...);

This writes the specified expression(s) to the standard output device (usually the screen). The instruction **WRITELN** has the same function but will cause the cursor to skip to the next line *after* outputting the expression list. An expression may involve combinations of variables (of most types) and constants. String constants should be enclosed in **single** quotes ' '.

Examples :

```
WRITE (1+6);
WRITELN ('1 + 6 = 7');
WRITELN (1,' / ',6,' = ',1/6);
WRITE (sum);
WRITELN ('The sum of ',num1,' and ',num2,' is ',num1+num2);
```

4.4. Formatted Output

Formatting information may be specified alongside an item in a WRITE statement by using a colon. Formatting is particularly useful for printing out neatly aligned tables of data.

integers	**WRITE(i:7)**	i will be RIGHT-JUSTIFIED in a field of width 7 characters.
strings	**WRITE(s:20)**	s will be RIGHT-JUSTIFIED in a field of width 20 characters.
reals	**WRITE(r:10:4)**	r will be RIGHT-JUSTIFIED in a field of width 10 characters, with 4 digits following the decimal point.

NOTES

1. Formatting integers and strings requires only a single value after the colon - the field width. If the value to be printed is shorter than the field width, extra spaces will be inserted BEFORE it (this is what RIGHT-JUSTIFIED means).

2. Fomatting is very important when printing out real numbers, because otherwise real numbers will be diasplayed using scientific notation (which is not particularly easy to understand). When formatting reals, two items of information must be specified - the field width AND the number of digits required after the decimal point.

Examples:

```
WRITELN ('Tom':12);
WRITELN ('Jonathan':12);
WRITELN ('1 / 2 = ', 1/2);
WRITELN ('1 / 2 = ', 1/2:6:4);
```

4.5. Data Input: READ and READLN

SYNTAX : **READ** (<var 1> , <var 2> , ...);

This reads values from the standard input file (the keyboard) and places the values in the specified variables. The instruction **READLN** has the same function but will cause the cursor to skip to the next line *after* inputting. When a running program encounters a **READ** statement, it will stop running and wait for the user to enter some data. Care should be taken that the data corresponds to the type of variable it is destined for. For example, you cannot input a string into a variable of **REAL** type !

Examples:

```
READ   (a,b,c);
READ   (name,address);
WRITELN ('Enter two numbers:');
READLN  (num1,num2);
WRITELN ('The sum of ',num1,' and ',num2,' is ',num1+num2);
```

4.6. The Assignment Statement

SYNTAX : < variable > **:=** < expression > ;

Examples:

```
sum   := num1 + num2;
name  := 'George';
count := count + 1;          { Increments value of count }
```

ý **NOTES:**

1. This statement will evaluate the expression on the right-hand side and place the result in the variable specified on the left-hand side.

2. This is equivalent to the **BASIC** command **LET**. However the symbol **:=** is used in place of = to distinguish between **assigning a variable** and **testing for equality**.

4.7. Arithmetic Operators and Functions

1. Arithmetic can be performed using operators and functions. An operator works on one or two expressions e.g. x + y, -num. A function works on one or more arguments to return a result e.g. y := **SIN** (x).

2. The arithmetic operators associated with integers are **+**, **-**, *****, **DIV** and **MOD**. The effect of the **DIV** and **MOD** operators is as follows:

x **DIV** y	will divide x by y and return the integer part of the result (ignoring the remainder). eg 10 **DIV** 6 returns 1.
x **MOD** y	will divide x by y and return the remainder (integer). e.g. 10 **MOD** 6 returns 4.

3. The reserved word **MAXINT** is a constant equivalent to the maximum possible integer that can be stored in a variable of type **INTEGER**. How can you discover the value of MAXINT?

4. The arithmetic operators associated with reals are +, -, *, /. The division operator / will give a result which may involve a fraction. Integers may be involved in a real expression. They will automatically be converted to real.

Example:

```
VAR
   capacity, velocity,c ross_section : INTEGER;
   time, flowrate    : REAL;

flowrate := velocity * cross_section;
time     := capacity / flowrate;
```

5. The value of an integer expression may be passed to a *real* variable as well as an integer variable (as in the above example) . However, the converse is not directly possible. The functions **ROUND()** and **TRUNC()** can be used to convert a real to an integer.

Examples:

```
ROUND (3.232)      {returns the integer 3}
ROUND (3.8991)     {returns the integer 4}
TRUNC (3.8991)     {returns the integer 3}
```

6. Pascal also has an **INT()** function. This takes a **REAL** number and returns the INTEGRAL PART of the number as another **REAL NUMBER**.

   Example

```
INT(5.34)        {returns 5.0 (NOT 5)}
```

7. Pascal also provides other mathematical functions. A function takes one or more arguments (within brackets) and returns a value. Some functions are restricted to a particular type.

   Functions returning integers: (i represents an integer, r a real)

SUCC(i)	Returns i+1
PRED(i)	Returns i-1
ABS(i)	Returns absolute of i (i.e. removes sign)
SQR(i)	Returns the square of i
TRUNC(r)	Returns an integer representing the truncated value r
ROUND(r)	Returns an integer representing the rounded value r

   Functions returning reals: (n represents an integer or a real)

SIN(n)	Returns sine of n
COS(n)	Returns cosine of n
ARCTAN(n)	Returns arctangent of n
LN(n)	Returns natural logarithm of n (to base **e**)
EXP(n)	Returns **e** raised to the power of n (e^n)
SQRT(n)	Returns the square root of n
ABS(n)	Returns a real if n is a real
SQR(n)	Returns a real if n is a real
INT(r)	Returns the integral part of r as a REAL NUMBER

Note that in the functions **SIN** and **COS,** the argument is measured in **RADIANS** rather than degrees. Similarly, the angle returned by **ARCTAN** is also in radians.

4.8. Exercises - 1

1. Write a program which converts angular measurements from degrees to radians. Display the **SINE** and **COSINE** of the angle input at run time. NOTE that

$$\text{radians} = \frac{\text{degrees} \times 2 \times p}{360}$$

Pascal has a predefined constant **PI** to represent π.

2. Write a program which inputs a real value and displays its **ARCTANGENT** in degrees. Use the formula:

$$\text{degrees} = \frac{\text{radians} \times 360}{2 \times p}$$

4.9. Exercises - 2

In each of the exercises below, be careful to include prompts for any input required and to present output neatly with headings where possible. Use comments, where necessary, to explain what the program is doing. Use **CONST** declarations where you think it would be appropriate.

a. Write a program which takes three numbers as input, calculates their total and average and prints them on the screen. Use suitable prompts and headings.

b. Write a program which takes a person's age in years as input and outputs the person's approximate age in months and in days (ignore leap years).

c. Write a program which takes as input the name of a customer, the quantity and name of an item ordered by the customer, and the price per item, and prints out a letter acknowledging receipt of the order as follows:

Dear <name> **,**

Thank you for your order of <quantity> **of** <item name> **.**

Your order will be dispatched on receipt of Lm <total cost> **.**

The information within <> should automatically be filled in by the program from data input.

d. For a right-angled triangle having sides **a**, **b**, **c**, where **c** is the hypoteneuse, Pythagoras discovered that $\mathbf{c}^2 = \mathbf{a}^2 + \mathbf{b}^2$. Write a program which reads values for **a** and **b** and calculates the length of the hypoteneuse.

e. Write a program which calculates and outputs the diameter, circumference and area of a circle whose radius is input at run-time.

f. Write a **Farenheit** to **Centigrade** conversion program which takes a Farenheit temperature as input and outputs the corresponding temperature in Centigrade (or Celsius). Use the formula

$$\mathrm{C} = (\mathrm{F} - 32) \times \frac{5}{9}$$

where **C** is the temperature in Centigrade and **F** is the temperature in Farenheit.

g. Pascal has no exponentiation operator (like BASIC's ^ or **). How may this be rectified using **LN** and **EXP**?

CHAPTER 5

5. Conditional and Compound Statements

5.1. Boolean Values and Relational Operators

A **boolean value** is either the value **TRUE** or the value **FALSE**. It is important to understand that TRUE and FALSE are NOT STRINGS **-** they are values of type **BOOLEAN**, just as 4 is a value of type INTEGER.

Boolean values are generated by **boolean expressions**, just as integer values are generated by integer expressions. The simplest boolean expression involves the comparison of two objects using a **relational operator**. Pascal supports the following relational operators:

RELATIONAL OPERATORS	
Operator	**Meaning**
=	Equal
<>	Not equal
>	Greater than
<	Smaller than
<=	Smaller than or equal
>=	Greater than or equal

The objects to be compared using these relational operators must be COMPARABLE. For example two strings can be compared, and two numeric values can also be compared (even if one is a real and the other an integer), but a string and a number cannot be compared.

Boolean values can be printed out using WRITELN. Thus we can have:

```
WRITELN(a>b);
WRITELN(x<>0);
```

5.2. Logic Operators

Boolean expressions can be combined using the logic operators **NOT**, **AND** and **OR**, sometimes also called **boolean operators**. You are undoubtedly familiar with these from your O-level, although you probably used 1 and 0 instead of TRUE and FALSE.

LOGIC OPERATORS	
Logic Operator	**Result**
NOT(X)	The inverse of X
X **AND** Y	TRUE if and only if both X and Y are TRUE
X **OR** Y	TRUE if at least one of X and Y is TRUE

NOTE: Boolean expressions combined using these three operators should be bracketed to avoid ambiguity. Thus, **A>B OR X=0** is not acceptable. Instead it should be written as **(A>B) OR (X=0).**

5.3. Boolean Variables

Just as numeric values can be stored in numeric variables (of type real or integer), similarly boolean values can be stored in **boolean variables**. Boolean variables are only allowed to store one of the two values TRUE or FALSE. Boolean variables are sometimes called **flags**.

Example declaration of boolean variables:

```
VAR
  ready, full, more : BOOLEAN;
```

Having declared these boolean variables, we can now proceed to use them in the program:

Examples:

```
ready := full AND NOT more;      {These two}
ready := (full AND (NOT more));  {are equivalent.}
ready := (count1 > 10) AND (count2 >20);
```

In the last example, note how each individual comparison had to be enclosed in brackets.

There is a standard function **ODD(x)**, where x is an integer, which returns a BOOLEAN value. The value is **TRUE** if x is odd and **FALSE** otherwise. Thus, if NumIsOdd is a boolean variable, we can write:

```
NumIsOdd := ODD(Num);
```

EXERCISE

Consider the following section of program:

```
CONST
   number=30;
  VAR
   num1,  num2 : INTEGER;
   equal, big1, small1, big2, small2 : BOOLEAN;
BEGIN
  WRITELN ('Enter a number');
  READLN  (num1);
  WRITELN ('Enter another number');
  READLN  (num2);
  equal  := num1 = num2;
  big1   := num1 > number;
  small1 := num1 < number;
  big2   := (num2>number) OR ((num2>num1) AND big1);
  small2 := (num2<number) OR ((num2<num1) AND small1);
END.
```

At the end of the program, what will the values of all five boolean variables be if the inputs to **num1** and **num2** are:

(i) 20 and 10	(ii) 20 and 20	(iii) 50 and 10
(iv) 30 and 20	(v) 60 and 30	(vi) 40 and 50

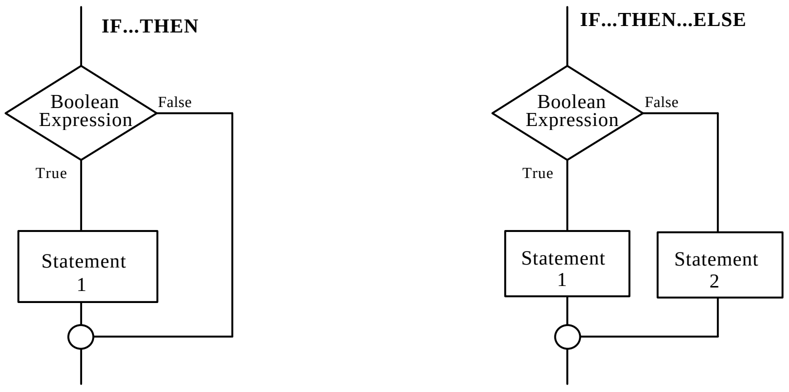
The two longest lines of the above program have some unnecessary code. Rewrite the lines in a shorter form.

5.4. IF...THEN...ELSE

Syntax 1: **IF** <boolean expr> **THEN** <statement 1> ;

Syntax 2: **IF** <boolean expr> **THEN** <statement 1> **ELSE** <statement 2> ;

The following flowcharts demonstrate the difference in logic between the two forms:



Examples:

```
IF size > maxsize THEN WRITELN ('Full up');
IF waiting THEN count := SUCC(count);
IF (x>y) OR (a<10) THEN z:=10 ELSE z:=20;
```

NOTES:

1. Using syntax 1, statement 1 will be executed if the value of the boolean expression is **TRUE**. Afterwards, execution continues at the following statement.
2. Using syntax 2, if the value of the boolean variable is **TRUE**, then statement 1 is executed. However, if it is **FALSE** then statement 2 is executed.
3. In syntax 2 there is **NO SEMICOLON** between statement 1 and the ELSE.

5.5. Compound Statements

A compound statement has the form:

```
BEGIN
  statement 1;
  statement 2;
  ...
  statement n;
END
```

NOTES:

1. A compound statement can be treated as a normal simple statement. It serves to **GROUP** many statements into a **SINGLE UNIT**.
2. A compound statement can contain more compound statements. It is useful to indent the program text so that one can easily make out the different levels of compound statements.
3. Compound statements make the **IF...THEN...ELSE** a more powerful construct. We can now do more than one thing for a single condition.

Example

```
IF count > maxrange THEN
BEGIN
      WRITELN ('Count exceeded range');
      ready := TRUE;
      count := 0;
END
ELSE
BEGIN
      WRITELN ('Count has not exceeded range');
      count := SUCC(count);
END;
```

NOTE that there is NO SEMICOLON between the **END** of the compound statement and the **ELSE**.

5.6. Exercises - If..Then..Else

a. In a guessing game, player A enters an integer between 1 and 10 while player B is not looking. If player A cheats by giving a number outside this range, the word **CHEAT** is displayed and the game stops. Otherwise the screen is cleared and player B is given a chance to guess the number. If he guesses it then he wins. If his guess is within three numbers away from Player A's number then the game is a draw. Otherwise player A wins. Write a program to implement this game.

b. In an election, people under 18 years old are not eligible to vote. Input a person's age and output a message stating whether he can vote or not.

c. Write a program which accepts a number as input and outputs a message stating whether the number lies between 1 and 100 or not. Do you think that a **CONST** declaration would be appropriate in this program ? Why ?

d. Write a program which asks the user to input 3 numbers, and outputs the largest.

e. Input a number representing a year and output a message stating whether it is a leap year or not. Note that a leap year is

```
EITHER divisible by 400
OR ((divisible by 4) BUT NOT (divisible by 100)).
```

f. Write a program which takes as input from the keyboard the boolean values **a**, **b**, **c**, **d**, and displays the result of the logic function **(a AND b) OR (c AND NOT d)**. NOTE that since PASCAL does NOT allow you to READ boolean variables, you will have to devise a different method for reading the values into variables a,b,c and d.

g. A company has three vans which are used to transport its products

VAN	**MAXIMUM CAPACITY**
A	90
B	50
C	30

Write a program which asks for the number of items to transport, validates this number (must be between 1 and 170) and suggests which vans should be used for the job. A van should not be used where a smaller one would be sufficient.

h. Quadratic equations of the form $\mathbf{ax^2 + bx + c = 0}$ have roots x_1 and x_2 which can be calculated using the formulae:

$$x_1 = \frac{-b+\sqrt{b^2-4ac}}{2a} \quad and \; x_2 = \frac{-b-\sqrt{b^2-4ac}}{2a}$$

The roots may fall into one of the following three categories:

i. They are real and equal if $b^2 - 4ac = 0$.

ii. They are real and distinct if $b^2 - 4ac > 0$.

iii. They are complex and distinct if $b^2 - 4ac < 0$.

Write a program which first accepts the values for **a**, **b**, **c**, where **a** should be checked to ensure that it is *NOT* zero. Next the program should test the value of ($b^2 - 4ac$) to determine which category the roots fall into. Finally the roots should be calculated and output (if in category i or ii) with a message indicating the category under which they fall.

i. Given an integer N, calculate the number of characters required to print N. Note that MAXINT is 32767, so a positive integer requires a maximum of 5 characters. However, the smallest integer, -32768, requires 6 because of the minus sign.

CHAPTER 6

6. The CASE Statement

6.1. Syntax of the CASE Statement

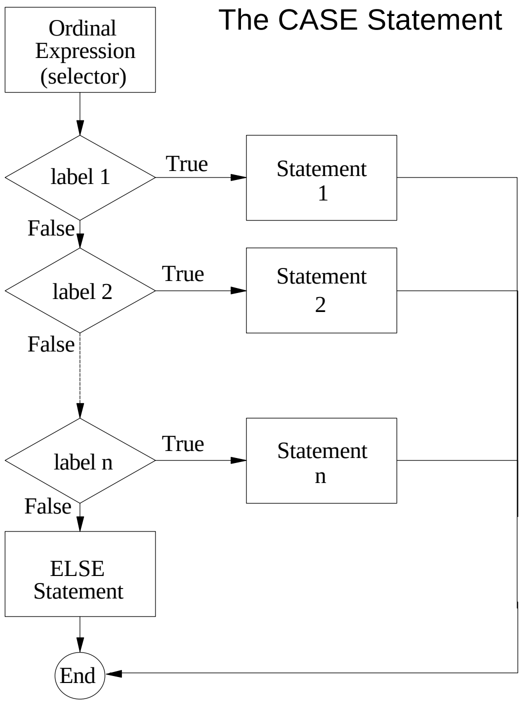
Syntax : **CASE** < expression > **OF**
const1: < statement 1 >;
const2: < statement 2 >;
...
const*n*: < statement *n* >;
ELSE < default statement >;
END { of case }

Example :

VAR mark : **INTEGER** ;
comment : **STRING** [maxlength] ;

```
CASE mark OF
10 : comment := 'Full Marks';
 9 : comment := 'Excellent';
 8 : comment := 'Very Good';
 7 : comment := 'Fairly Good';
 6 : comment := 'Not Bad';
 5 : comment := 'Can do better';
 4 : comment := 'Poor';
 3 : comment := 'Very Poor';
 2 : comment := 'Terrible';
 1 : comment := 'Rubbish';
 0 : comment := 'Total loss!'
ELSE  comment := 'Invalid Mark';
END;
```

The CASE Statement



NOTES :

1. The **IF** statement allows a process to select one of two possible choices of action. The **CASE** statement is a generalization of this, which enables the process to execute one of several actions according to the value of an expression.

2. The expression being tested must be one returning a **ORDINAL TYPE** (ie integer or character, sometimes also called **SCALAR** types). **Strings may NOT be tested.**

3. The **ELSE** as used in the syntax for **CASE** as shown above is not included in standard PASCAL, but is an enhancement of **TURBO PASCAL**. If the value of the expression does not match any of the constants, then the default statement (ie the one following the **ELSE**) is executed.

6.2. Using Many Labels In A Single Case

More than one constant may be associated with a single action. The action would then be executed if the expression being tested matches one of these constants.

Example

```
VAR   reply     : CHAR ;
      continue  : BOOLEAN;

BEGIN
   WRITE ('Do you want to continue? <Y/N> ');
   READLN (reply);
   CASE reply OF
      'Y','y' : continue := TRUE;
      'N','n' : continue := FALSE;
      ELSE WRITELN ('You should press Y, y, N or n');
   END;
END.
```

You can also use a range of constants, for example:

```
VAR c : CHAR;

BEGIN
   WRITE  ('Enter a character: ');
   READLN (c);
   WRITE ('You have entered a');
   CASE c OF
      'a'..'z'  :  WRITELN (' lower case character.');
      'A'..'Z'  :  WRITELN ('n upper case character.');
      '0'..'9'  :  WRITELN (' digit.');
      ELSE WRITELN (' punctuation character.');
   END;
END.
```

6.3. Simple Menus

A SIMPLE MENU is a list of options offered to the user, together with a KEY (usually a single character) for each option. The user selects the required option by pressing the associated key. The program performs the following steps:

1. *Displays the menu*
2. *Prompts user to select option*
3. *Waits for, and reads, a single character from the user (the option key).*
4. *Uses a CASE statement to determine what processing is required depending on the key input by the user.*

Reading the single-character option key in step 3 is best accomplished using Turbo Pascal's **READKEY** function. Readkey will wait for the user to press any key, and will return the corresponding character as its result. The advantage of using READKEY for this job is that the user is not required to press ENTER after making the selection (as would be the case if READLN had been used). Also, READKEY does not echo the character it reads.

6.4. Exercises

1. Write a simple calculator program which first takes two INTEGERS **int1** and **int2** as input and then presents a menu of five arithmetic operations: addition, subtraction, multiplication, division, quotient and modulo:

```
               MENU

      +     num1 + num2
      -     num1 - num2
      *     num1 * num2
      /     num1 / num2
      Q     num1 div num2
      R     num1 mod num2
      X     EXIT

           Enter choice ...
```

   Your program should use READKEY to read the menu option. The option should be handled using a **CASE** statement (which should treat upper and lower case characters as equivalent), and the corresponding result displayed. Your program should ignore invalid choices.

2. i. Input the number of a month (1..12) and use a **CASE** statement to assign the number of days in the given month to the variable **days**. Output this variable with a suitable message at the end of the program (ignore leap years).

   ii. Extend the above program to read in the year number and test for leap years. Use the algorithm you learned in a previous lesson to set the boolean variable **leap** to true if the year is a leap year. Use this variable in the case that the month is 2.

   iii. Modify the CASE statement in the above so that it also assigns the month name to the (string) variable **monthname**.

3. Write a program which reads in a number from 1 to 31, and sets a string variable SUFFIX to ST, ND, RD or TH depending on the number read. The program should then print out the number followed by its suffix (for example, if the user inputs 21, the program should print 21st).

4. Write a program which reads in three integers representing an abbreviated date and output the date in full eg. **2 4 95** → **2nd April 1995**. Modify and use the program from 2.ii to check that the date is valid before printing it. An invalid date such as **50 13 77** should result in the message **ERROR** being displayed.

CHAPTER 7

7. Arrays and Loops

7.1. Single Dimensional Arrays

A single dimensional array is a SEQUENTIAL COLLECTION OF STORAGE LOCATIONS, ALL OF THE SAME TYPE. Each LOCATION (or **CELL**) is given an ADDRESS (or **INDEX**) to identify it from the other cells in the same array. In this respect, a single dimensional array may be compared to HOUSES (cells) in a STREET (array).

A single dimensional array must be declared in the VAR section of a PASCAL program using the syntax:

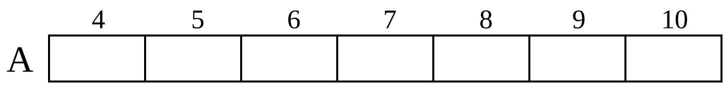
VAR <array name> : **ARRAY** [<lower bound>..<upper bound>] **OF** <type>;

The notation **X..Y** is called a **SUBRANGE**. Pascal allows two types of subranges:

integer	:	-3..2 which is the range -3,-2,-1,0,1,2
character	:	'a'..'z' which is all letters between and including 'a' and 'z'.

The subrange used to dimension the array declares the RANGE OF ADDRESSES (house numbers) to be associated with each cell in the array. Thus, the declaration:

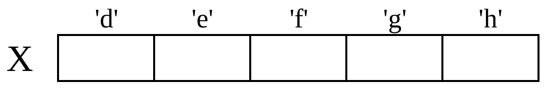
```
VAR a : ARRAY [4..10] OF INTEGER;
```
creates the following structure called A:



NOTE that the numbers 4..7 are the **ADDRESSES** or **INDICES** of the array A, **NOT** its contents. Thus, the first cell of A is A[4], and the last is A[10].

Pascal allows arrays to be dimensioned using a character range. In this case the array must be indexed using a character in the range used to dimension the array. For example, the declaration:

```
VAR x : ARRAY ['d'..'h'] OF REAL;
```
creates the following structure called X:



In this case, the first cell of X is X['d'] and the last X['h'].

NOTES:

1. In PASCAL, only a constant may be used to dimension an array.
2. A real may NOT be used to index an array.

3. An uninitialized array contains only garbage.
4. one array may be assigned to another array **IF AND ONLY IF** (both arrays are of the same type) **AND** (both arrays are the same size).

7.2. Two Dimensional Arrays

In a 2-dimensional (or RECTANGULAR) array, cells are arranged in ROWS and COLUMNS. Hence, each cell is identified by TWO ADDRESSES - a ROW index and a COLUMN index. The following rectangular array R has columns numbered 2..6 and rows numbered 6..8.

R	**2**	**3**	**4**	**5**	**6**
6					
7		H			
8					

The cell marked with a **H** is in row 7 column 3. Hence, its address is R[7,3]. To declare this array in PASCAL, use:

VAR R : **ARRAY** [6..8 , 2..6] **OF** <type> ;

Where <TYPE> is whatever type of value is to be stored in each cell. As with single-dimensional arrays, characters may also be used to index either the columns, or the rows, or both.

Examples

```
VAR a   : ARRAY [1..6] OF INTEGER;
    b   : ARRAY [0..2 , 4..7] OF REAL;
    c   : ARRAY ['a'..'z'] OF INTEGER;

    a[5] := 12;
    j := b[1,4];
    x := 6 ; a[x] := -8;
    c['f'] := 70;
```

7.3. Higher Dimensional Arrays

Arrays with more than 2 dimensions may also be declared. A 3-dimensional array is like a cuboid - with an X,Y and Z index. For example, the following declaration defines a 3-dimensional array T:

```
VAR T : ARRAY [1..4, 1..3, 1..8] OF INTEGER;
```

Higher dimensional arrays are more difficult to visualise than the 1- and 2-dimensional arrays we will mostly be concerned with.

7.4. Exercises - Arrays

1. Assume the following declarations:

```
VAR array1    : ARRAY  [-2..2]  OF INTEGER ;
    array2    : ARRAY  [1..5]   OF INTEGER ;
    array3    : ARRAY  [1..10]  OF INTEGER ;
    array4    : ARRAY  ['b'..'g'] OF REAL;
    int1      : INTEGER ;
    real1     : REAL ;
```

   a. How many elements are there in each array ?

b. which of these statements is illegal? Why?
1. array1 := array2 ;
2. array3 := array2 ;
3. array1 [4-3] := 55 ;
4. int1 := 3 ;
   array3 [int1 * 4] := 0 ;
5. array1 := array2 * 3 ;
6. array2 [3] := real1 ;
7. real1 := 2 ;
   array3 [real1] := 200 ;
8. array4['a'] := real1;

2. Draw a representation of the rectangular array

```
        array5 : ARRAY [1..4 , 2..4] OF INTEGER ;
```

then fill in the array by following these statements :

array5 [2,3] := 4 ;
array5 [1,4] := array5 [2,3] * 2 ;
array5 [4,2] := array5 [1,4] + array5 [2,3] ;
array5 [2,2] := array5 [4,2] DIV array5 [2,3] ;
array5 [1,3] := array5 [1,4] - (array5 [2,2] * 2) ;
array5 [array5 [2,2] , array5 [1,3] * 2] := 1 ;

7.5. Loops

A **LOOP** is a **CONTROL STRUCTURE** which **REPEATS** one or more statements.

Loops are what mostly make the **GOTO** statement redundant in Pascal. Pascal supports three kinds of looping mechanisms: **FOR, REPEAT** and **WHILE**.

- The **FOR** loop has two variations **-** increasing counter and decreasing counter.
- The **REPEAT** loop differs from the **WHILE** loop in that the **REPEAT** loop tests its termination condition at the end of the loop, while the **WHILE** loop tests its termination condition at the beginning of the loop.

It is very important to determine the kind of loop which best serves a particular situation.

7.6. The FOR..DO loop

SYNTAX 1	:	**FOR** <counter> := <start value> **TO** <end value> **DO** <statement> ;
SYNTAX 2	:	**FOR** <counter> := <start value> **DOWNTO** <end value> **DO** <statement> ;

Examples :

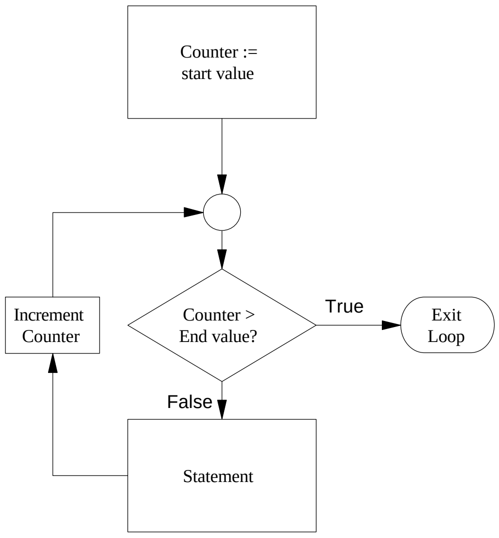
```
1.  FOR i := 1 TO 5 DO WRITELN (i) ;

2.  WRITE  ('Enter an integer: ') ;
    READLN (int) ;
    FOR j := ABS(int)  DOWNTO 2 DO
    BEGIN
       WRITE (j) ;
       IF (int MOD j = 0) THEN       { if INT is divisible by J }
          WRITE(' -- is a factor') ;{ then it is a factor of J }
       WRITELN ;
    END ;
```

NOTES :

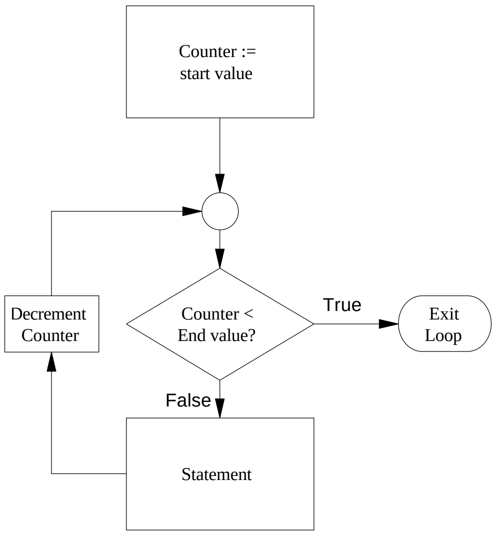
1. the <counter> must be a variable of ordinal type (i.e. NOT of type REAL or STRING).

2. <start value> and <end value> may be constants or expressions, but must be of the same type as <counter>.
3. <statement> may be a simple statement (including another FOR..DO statement), or a compound statement.
4. in syntax1, if start value > end value, then the loop will NOT be executed.
5. in syntax2, if start value < end value, then the loop will NOT be executed.
6. no program statement within the FOR..DO loop may change the value of the <counter> variable.
7. after the loop has been executed, the value of the <counter> variable is undefined. This variable should not be used again until it has been assigned a new value.
8. the <counter> variable can be of type character, in which case <start value> and <end value> must also be character, and the <counter> variable takes on the values of characters in the range <start value>..<end value>.

Upcounting FOR loop



Downcounting FOR loop



Examples :

```
1. VAR c : CHAR;
   FOR c := 'a' TO 'z' DO WRITE(c);

2. { This prints out all 3-character sequences generated
   by characters 'a'..'d', from 'aaa' to 'ddd' }

      FOR c1 := 'a' TO 'd' DO
         FOR c2 := 'a' TO 'd' DO
            FOR c3 := 'a' TO 'd' DO
                  WRITELN(c1,c2,c3);

3. { This prints out all possible 5-number combinations in
   the SUPER FIVE lottery.  However, since there are
   376992 such combinations, it might make quite a while
   for this program to execute! }

   FOR n1 := 1 TO 36 DO
      FOR n2 := n1+1 TO 36 DO
         FOR n3 := n2+1 TO 36 DO
            FOR n4 := n3+1 TO 36 DO
               FOR n5 := n4+1 TO 36 DO
                     WRITELN(n1:3,n2:3,n3:3,n4:3,n5:3);
```

7.7. The REPEAT..UNTIL loop

SYNTAX : **REPEAT** <statements> **UNTIL** <condition> ;

Example :

```
VAR j : INTEGER ;
   REPEAT
      READLN  (j) ;
      WRITELN (j) ;
   UNTIL ODD(j) ;
```

NOTES :

1. the statements in the REPEAT..UNTIL are executed repeatedly until <condition> becomes TRUE.
2. since the condition is tested at the end, the REPEAT..UNTIL loop will be executed at least once.
3. any number of statements may be inserted between the REPEAT and UNTIL keywords. No BEGIN..END pair is required.
4. the body of a REPEAT..UNTIL loop may be a null statement, eg.

```
REPEAT UNTIL KeyPressed;  {wait until any key is hit}
```

7.8. The WHILE..DO loop

SYNTAX : **WHILE** <condition> **DO** <statement> ;

Examples:

```
1.  n := 0;
    WHILE n < MAXINT DO
       inc(n) ;
```

```
2. {  Starting with any positive integer, generate a sequence such
   that the successor of a number N in the sequence is N*3+1 if N is
   odd, and N/2 if N is even. It appears that from whichever INTEGER
   you start, you will eventually generate a 1. This, however, has
   not been proved }

      VAR j : INTEGER;

      WRITE  ('Enter a positive non-zero integer: ');
      READLN (j);
      WHILE j <> 1 DO
      BEGIN
         IF ODD(j) THEN j := j * 3 + 1
         ELSE j := j DIV 2;
         WRITE (j:8);
      END;
```

NOTES :

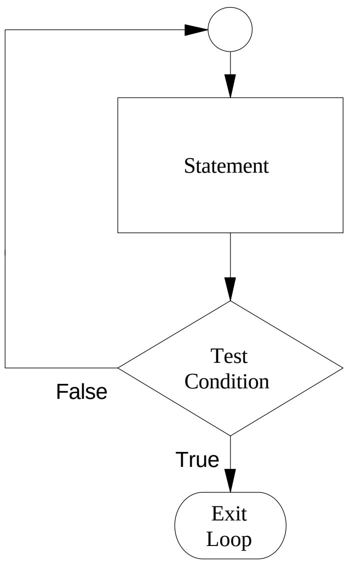
1. the statements in the WHILE..DO loop are executed repeatedly until <condition> becomes FALSE.
2. since the condition is tested at the beginning of the loop, the statements in the WHILE..DO loop may never get executed.
3. <statement> may be any statement, simple or compound (bracketed with a BEGIN..END pair).
4. the body of a WHILE..DO loop may be a null statement, eg.

```
WHILE NOT KeyPressed DO;  {wait until any key is hit}
```

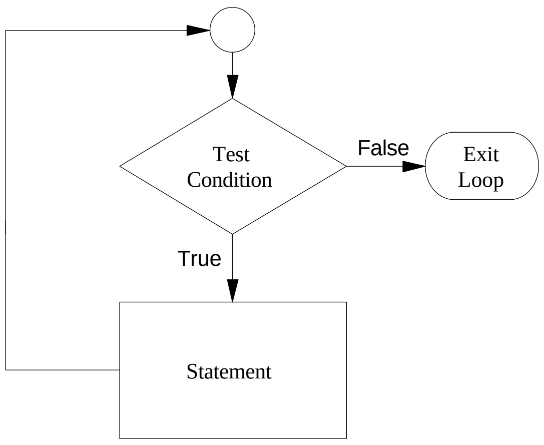
7.9. Comparison of REPEAT..UNTIL and WHILE..DO loops

The following pair of flowcharts demonstrate the different control logic of the REPEAT..UNTIL and the WHILE..DO loops.

REPEAT ***statement*** **UNTIL** ***condition***



WHILE ***condition*** **DO** ***statement***



7.10. Exercises - FOR, REPEAT and WHILE loops

1. How many times will the following loops be executed?

   a. **FOR** i := 12 **TO** 12 **DO WRITE**(i);
   b. **FOR** i := 12 **TO** 10 **DO WRITE**(i);
   c. **FOR** i := 12 **DOWNTO** 10 **DO WRITE**(i);
   d. **FOR** i := 10 **DOWNTO** 12 **DO WRITE**(i);
   e. **FOR** c := 'd' **TO** 'h' **DO WRITE**(c);

2. Write a short program to execute this FOR..DO loop:

   FOR j := -4 **TO** 20 **DO WRITELN**(j) ;

   i. rewrite your program using a REPEAT..UNTIL loop, and check it by comparing the output of the two programs.

   ii. do the same using a WHILE..DO loop.

3. The **FOR..DO** statement in Pascal lacks a **STEP** clause. Also, the counter variable may not be of type **REAL**. Show how these constraints may be worked around by using

   i. a **REPEAT..UNTIL** loop

   ii. a **WHILE..DO** loop

   to implement the BASIC program:

```
10 FOR i = 3.5 TO 10 STEP 0.5
20 PRINT i
30 NEXT i
```

4. Turbo Pascal has an inbuilt random-number generator called by a statement of the form **num := RANDOM (x);** where **num** is an integer variable, and **x** is an integer-value. RANDOM returns a random integer between 0 and x-1.

Sorting is a very important activity in any type of programming application. There are dozens of different sorting algorithms, varying in speed, complexity and memory requirements. Some well-known sorts are the Quicksort, Heapsort, Insertion sort, Batcher sort and Bubble sort. The bubble sort is the slowest and least complicated.

Fill an array of 100 integers with some random numbers (using RANDOM(1000)-500), sort it using a bubble sort, and output the result. The bubble sort algorithm in pseudo code is:

```
ALGORITHM BUBBLE SORT [ sort an array 1..MaxIndx ]
1.  [ SET UPPER SORT INDEX ]
    UpperBound ← MaxIndx - 1.
2.  [ FINISHED? ]
    if UpperBound = 0 then array is sorted. End.
3.  [ PERFORM ONE PASS ]
    LastSwap ← 0.
    For i ← 1..UpperBound, if array[i] > array[i+1] then swap
    and set LastSwap ← i
4.  [ SET UPPERBOUND TO LAST SWAP AND PERFORM
      ANOTHER PASS ]
    UpperBound ← LastSwap , goto step 2.
```

5. Searching is just as important as sorting **-** and done more often. Suppose we have a table (array) of values (for example, integers), and we would like to know if a particular value is in the table. If the table is not sorted, then the table must be searched sequentially, cell by cell, to determine whether or not the value is in the table. This may not matter much if the table is small, but with large tables the search slows down too much.

However, if the table is sorted, there is a better searching method, called the BINARY SEARCH, as described in the following pseudo-code:

```
ALGORIHTM BINARY SEARCH

[search a sorted array 1..MaxIndx for value K. Returns FOUNDAT=0 if not found,
else FOUNDAT=position of K in the array.]

1.  [ INITIALISE ]
    left ← 0
    right ← MaxIndx + 1
2.  [ COMPUTE MIDPOINT ]
    mid  ←  (left + right) DIV 2
3.  [ UNSUCCESSFUL? ]
    if (mid = left) then the value K is NOT in the table, so set FOUNDAT ← 0
    and terminate.
4.  [ COMPARE ]
    At this point we compare K with the value at position array[mid], and take
    different action depending on the value of the comparison:
```

IF	THEN
K < array[mid]	right ← mid, goto step 2
K = array[mid]	foundat ← mid, terminate
K > array[mid]	left ← mid, goto step 2

Fill an array of 1000 integers with random integers using (RANDOM(4000)-2000), sort it using the BUBBLE SORT, then repeatedly ask the user to input values to be searched for in the array using the BINARY SEARCH. For each value, the program should print

NOT IN TABLE if not found, or FOUND AT POSITION **n** (where **n** is the position at which the value was found) if found.

6. Write a program which generates the sequence described in example 2 of the section on the WHILE..DO loop, for all integers in the range 1..400 (do not display the sequence). The program should report which integer took longest to degenerate to 1, and the number of steps it required, for example ***327 took longest, requiring 143 steps***.

CHAPTER 8

8. Strings and Text Files

8.1. String Variables

Turbo Pascal supports a STRING type. A string variable is declared by

```
VAR s1 : STRING[50] ;
```

Variable s1 may then be assigned any string up to 50 characters. It is important to remember that the size specified when a string is declared dictates the MAXIMUM length to which the string may grow. The actual size of the string at any one time will vary depending on its current contents. If a string variable is assigned a string longer than its maximum length, the string will be truncated.

The maximum length of a Turbo Pascal string is 255 characters. If the length of a string is not specified in its declaration, it is assumed to have the maximum allowed length (255 characters). Thus, the following two declarations are equivalent:

```
VAR s : STRING[255] ;
VAR s : STRING ;
```

8.2. Operations On Strings

Turbo Pascal supports a number of inbuilt string-functions:

Finding the actual length of a string (LENGTH)

SYNTAX	**Length** (string)
RETURNS	An integer - the length of the string
EXAMPLE	i := Length('abcdefg'); {Sets i to 7}

Concatenating a two or more strings (+)

SYNTAX	string + string + ... + string
RETURNS	A string - the concatenation of all strings.
EXAMPLE	s := 'ab' + 'bc'; {Sets s to 'abbc'}

Extracting a substring from a string (COPY)

SYNTAX	**Copy** (string, start position, number of chars)
RETURNS	A string - that part of STRING starting at the specified position and of the specified length.
EXAMPLE	s := Copy('abcdefg',2,3); {Sets s to 'bcd'}

Searching for a substring in a string (POS)

SYNTAX	**Pos** (Substring, String);
RETURNS	An integer - the starting position at which SUBSTRING occurs in STRING. If the substring is not found, returns 0.

EXAMPLE	i := Pos ('cd','abcdefg'); {Sets i to 3}

The following are the standard string procedures

Removing a portion of a string (DELETE)

SYNTAX	**Delete** (string, start position, number of chars);
NOTES	Removes that part of a string of the specified length starting at the specified position. The string must be a variable.

EXAMPLE	s := 'abcdefg';
	Delete (s,4,2); {s becomes 'abcfg'}

Inserting a substring into a string (INSERT)

SYNTAX	**Insert** (substring, target string, position);
NOTES	Inserts the string SUBSTRING into the string TARGET STRING starting at the specified position. The target string must be a variable.

EXAMPLE	s := 'abcdefg';
	Insert ('XYZ',s,3); {s becomes 'abXYZcdefg'}

8.3. Reading and Writing Characters in a String

Characters in a string can be accessed by subscript **1..LENGTH(string)**. Thus `WRITE(s[4]);` prints the FOURTH character of string S. Single characters may be stored in a string position, eg `s[3] := 'x';` changes the THIRD character of string S to 'x'.

NOTES

1. The subscript (position) must be in the range 1 to LENGTH(string).
2. Inside a string, a quote character is written as 2 quotes.

   Example:

   ```
   s1 := 'Here''s how.';  {sets s1 to Here's how.}
   ```

8.4. Comparing Strings

Strings and characters may be compared using the normal relational operators < , > , <= , >= , = , <> .

NOTES

For two strings S1 and S2:

1. S1=S2 only if both are of the same length and both match character for character. Thus `'tom '` and `'tom'` are not equal (the first has trailing spaces), and neither are `'tom'` and `'Tom'`.
2. S1 < S2 if S1 comes before S2 in dictionary order. This dictionary order is defined by the collating sequence of the ASCII character set - ie. the order in which characters appear in the ACSII set. Thus letters of the alphabet follow each other in the normal alphabetical order, digits and punctuation symbols preceed uppercase characters, and uppercase characters preceed lowercase characters.
3. S1 > S2 if S1 comes after S2 in dictionary order as explained above.

8.5. Text Files

Text files are files containing printable characters. The characters are structured into lines of possibly varying length, and the file can be displayed on screen or loaded into a text editor. Your PASCAL source files are text files, as are all files prepared using a text editor (files prepared using a wordprocessor are usually not text files, because they contain formatting codes in addition to normal text).

Pascal uses **SEQUENTIAL ACCESS** with text files. This is because a text file may contain lines of varying length, and therefore the position of a given line cannot be calculated. Furthermore, input and output cannot be performed simultaneously to a text file - a text file can be opened EITHER for reading, OR for writing, but NOT FOR BOTH.

8.6. File Variables

Pascal accesses disk files by means of a **FILE VARIABLE**. A FILE VARIABLE is like an **INTERNAL** name PASCAL uses to refer to a file. A file variable to read or write a text file must be declared with type **TEXT**:

```
VAR f : TEXT;
```

To associate a file variable with a particular disk file use ASSIGN. The format is:

```
ASSIGN ( <file variable> , <filename> );
```

where <file variable> is a variable of type TEXT

and <filename> is the name of the file as it appears when you do a DIR from MSDOS. The filename may including a path if necessary, and an extension if any, and can be either a string constant or a string variable.

For example:

```
ASSIGN (f,'myfile.txt');
```

ALL REFERENCES TO THE FILE WILL NOW BE MADE THROUGH THE FILE VARIABLE.

READ, **READLN**, **WRITE** and **WRITELN** can be used for reading and writing from and to files. In this case, THE FIRST ARGUMENT must be the file variable, for example:

```
WRITELN(f,'Hello Universe');
```

will write the string 'Hello Universe' to the file associated with the file variable F.

8.7. Reading From a Text File

To READ data from a text file, the following steps are necessary:

READING FROM A TEXT FILE

1	*Declare a file variable of type TEXT*	VAR f : TEXT;
2	*Assign it to an existing text file*	ASSIGN (f,'prog1.pas');
3	*Use RESET to move the FILE POINTER to the beginning of the file. NOTE that the file MUST ALREADY EXIST.*	RESET (f);

4	*Use EOF to determine when the END-OF-FILE has been reached.*	WHILE NOT EOF(f) DO
5	*Use READLN to read in strings from the file.*	READLN (f,s);
6	*Close the file when finished.*	CLOSE (f);

Besides strings, you can also read INTEGERS and REALS from a text file if the file contains strings representing numbers of the correct type. When reading from a text file, it is best to use READLN rather than READ.

EXERCISE:

Write a program to read in, and display, a Pascal source file.

8.8. Creating a New Text File

CREATING A TEXT FILE

1	*Declare a file variable of type TEXT*	VAR f : TEXT;
2	*Assign it to an existing text file*	ASSIGN (f,'prog1.pas');
3	*Use REWRITE to create the file. NOTE that if the file already exists, it will be overwritten.*	REWRITE (f);
4	*Use WRITELN to write strings to the file.*	WRITELN (f,s);
5	*Close the file when finished.*	CLOSE (f);

Besides strings, you can also write variables of type INTEGER and REAL to a text file, but they will be stored as strings. You can use formatting information when writing to a file, and can write more than one variable at a time. In other words, writing to a text file is identical to writing to the screen - except that you must specify the file variable of the file to write the data to.

EXERCISE:

1. Write a program to create a text file of integers 1..40
2. Write a program to read in and total a text file of numbers.
3. Write a program to join two text files, saving the result as a third.

8.9. Appending Data to a Text File

To append data to an existing text file, use the same procedure as for writing, but **instead of rewrite(f) use append(f)**. The file must already exist.

8.10. The Printer Text File

Turbo Pascal treats the printer device as if it were a text file. To send data to the printer, your program should use the standard library unit called PRINTER. Thus the uses statement of your program should look something like this:

```
        uses crt, printer;
```

The printer unit exports a single text variable called LST, which is assigned to the printer device. Writing data to the LST file results in the data being sent to the printer (assuming a printer is connected and online).

Example:

The following simple program prints a textfile called prog1.pas:

```
program PrintFile;
uses crt,printer;

var NextLine : String;
       InputFile : Text;

begin
   assign(InputFile,'Prog1.pas');
   reset(InputFile);
   while not eof(InputFile) do
   begin
      readln(InputFile,NextLine);
      writeln(Lst,NextLine);
   end;
   close(InputFile);
end.
```

NOTE that the **LST** file is automatically opened by the printer unit when your program is run, and automatically closed when it terminates. DO NOT use ASSIGN, RESET, REWRITE, APPEND and CLOSE on this file.

8.11. Exercises - Text Files

1. Write a program which takes a Pascal source file (eg. prog1.pas) and writes it to a new file (prog1.txt), preceeding each line with a line number and changing all characters to upper case (using the Turbo Pascal function UPCASE).
2. i. Write a program which reads a SINGLE string from the keyboard and writes it out again, replacing every sequence of consecutive spaces by a single space.

   ii. Write a program which does the same thing for every line read from a text file.
3. Write a program which reads details of transactions on a bank account from a text file, and outputs a bank statement summarizing the transactions. The input transaction file has the following format:

LINE	**CONTENTS**
1	account number
2	initial balance (in Lm)
3..eof	each line contains a number representing a transaction (in Lm) - positive for credits, negative for debits.

Your program should produce a bank statement with credit transactions in the first column and debits in the second, followed by the initial and final balances. If either balance is negative, the word **OVERDRAWN** should be printed next to it.

Use the following transaction file for testing your program:

```
102-AAD-119876725
1000
200
```

```
-500
100
-1000
-100
```

which should result in a bank statement like this:

```
Account number:     102-AAD-119876725

CREDITS (Lm)        DEBITS (Lm)
      200
                           500
      100
                          1000
                           100

Initial balance :   Lm    1000
Final balance   :   Lm     300   OVERDRAWN
```

4. Write a program which reads arithemetic expressions from a file and checks that the parenthesis balance. The expressions should be written to screen followed by an 'OK' or 'NOT OK' as appropriate.

   HINT use an integer counter which is incremented whenever a left-parenthesis is found, and is decremented whenever a right-parenthesis is found.

5. Explain what this program does. In particular, explain the function of all variables and replace them with meaningful variable names. Insert helpful WRITE statements to prompt the user. An important operation has been ommitted from the program. What is it?

```
PROGRAM p;
VAR s1,s2     : STRING;
    f         : TEXT;
    i1,i2,j   : INTEGER;
BEGIN
   i1 := 0;
   i2 := 0;
   READLN(s1);
   ASSIGN(f,s1);
   RESET(f);
   WHILE NOT EOF(f) DO
   BEGIN
      READLN(f,s2);
      i1 := i1 + LENGTH(s2);
      FOR j := 1 TO LENGTH(s2) DO
         IF s2[j] = '.' THEN i2 := i2 + 1;
   END; {while not eof}
   WRITELN(i1,i2);
END. {Program p}
```

CHAPTER 9

9. Procedures - 1

9.1. Top-Down Design

One approach to solving a complex problem is to divide it into subproblems. These subproblems can in turn be divided repeatedly into smaller problems, until the individual problems are so simple that they can easily be solved. This approach of designing a solution for a main problem by obtaining solutions for each of its subproblems is called **TOP-DOWN DESIGN**.

In order for this approach to be effective, it is desirable that the ***SUBPROBLEMS BE INDEPENDENT OF EACH OTHER***. Then each subproblem can be solved and tested by itself without fear of interfering with the solution of other subproblems.

Top-down design solutions of complex problems can be easily implemented for computer solution by using **BLOCK-STRUCTURED** high-level languages such as PASCAL. The main problem is solved by the **MAIN PROGRAM** (or DRIVER), and the subproblems by **SUBPROGRAMS**, known as ***PROCEDURES*** and ***FUNCTIONS*** in PASCAL. In this scheme:

1. each **SUBPROGRAM** executes the instructions necessary to solve a particular **SUBPROBLEM**. Like a worker in a factory, a subprogram is responsible for carrying out a specific task.
2. the **MAIN PROGRAM** co-ordinates the execution of the subprograms in order to affect a complete solution of the main problem. Like a manager in a factory, the main program is responsible for delegating work to the subprograms.

Since each subprogram solves one subproblem, and subproblems should be independent of each other, subprograms should also be, as far as possible,, independent of each other in the sense that one subprogram should not be written in such a way as to interfere with the working of the other (for example by overwriting another's variables). For this reason, each subprogram should use it's OWN PRIVATE WORKSPACE to save temporary results, etc. It is very much like workers in a factory or an office having their own workdesk so as not to get in each other's way.

9.2. Simple Procedures

Consider the following simple PASCAL program which uses a single procedure called DRAW_STARS:

```
SCOPE            PROGRAM prog1;                                    1
  1              CONST maxstars   = 20;                            2
                 VAR   i,numstars : INTEGER;                       3

       SCOPE       PROCEDURE draw_stars;                           4
         2         VAR i : INTEGER;                                5
                   BEGIN                                           6
                      FOR i := 1 TO numstars DO WRITE('*');        7
                    END {procedure draw_stars};                    8

                 BEGIN                                             9
                    FOR i := 1 TO maxstars DO                     10
                    BEGIN                                         11
                       numstars := i;                             12
                       draw_stars;                                13
                    END {for};                                    14
                 END {program}.                                   15
```

NOTES:

1. A procedure is like a small program - it has a heading, declarations and a body. Procedures and functions are collectively known as **SUBPROGRAMS** - they handle the subtasks required by the main task. Subprograms enable the user to extend the language by defining new commands.
2. Procedure declarations must come between the variable declaration section and the program **BEGIN** statement.
3. Program execution starts from the program **BEGIN** statement (line 9 in the example above). Procedures are not executed unless explicitly called (invoked) by the program (line 13).
4. Variables declared within a procedure (line 5) are called **LOCAL VARIABLES.** Local variables hold data which is private to the procedure - the main program has no access to variables local to a procedure. More importantly, the procedure may store data in local variables without the danger of corrupting any program data.
5. Local variables are **CREATED ANEW** everytime the procedure is called and **CEASE TO EXIST** when the procedure terminates - which also means that a procedure cannot place a value in a local variable and expect to find it there the next time it is called.
6. Procedures have access to all **GLOBAL VARIABLES** - ie those declared in the main program - except when a variable with the same name as a program variable is declared within the procedure (in which case the program variable is said to be **MASKED OUT** or **OCCLUDED** for the duration of the procedure call).
7. That part of a program in which a variable is 'visible' is called the **SCOPE** of the variable. The example program above has 2 scopes - scope 1 encompasses the whole program and is called the **global scope**. Scope 2 encompasses just the procedure draw_stars, and is called a **local scope**.

9.3. Procedures and the Stack

All high-level programming languages use a stack for the subprogram-calling mechanism. When a subprogram is invoked, the **RETURN ADDRESS** of the calling program is pushed on the stack. When the subprogram terminates, the return address is popped and becomes the address of the instruction to be executed next.

The stack is also used to store local variables in a subprogram. When the subprogram is invoked, sufficient space to hold all its local variables is reserved on the stack. The area allocated on the stack to hold the return address and the local variables is called a **STACK**

FRAME. When the subprogram terminates, its STACK FRAME is completely popped, thus explaining why the values of local variables are only defined when a subprogram is actually being executed.

9.4. Value Parameters - Parameters for Input

A procedure can take **PARAMETERS**. Parameters pass values to the procedure on which the procedure is to act. An example of this is **WRITELN(s1) -** s1 is a parameter which holds data on which the procedure **WRITELN** is to work. This method of passing data to a subprogram is to be preferred to passing data through global variables for the following reasons:

1. if data is passed to a procedure in a global variable, the procedure may unintentionally change the value in the global variable, possibly corrupting some data which is important for the program to function correctly.
2. if data is to be passed to a procedure in a global variable, the procedure MUST KNOW the name of the variable in which it is to receive data. This goes against our design objective of writing procedures which work irrespective of the program they are used in, since such procedures would only work in a program which has declared a global variable with the expected name. This also goes against another important design principle - reusability, the facility of writing a procedure once but using it in many different programs. We'll talk more about this later on.

Thus, the procedure draw_stars in our example program is not designed properly because it gets the data it needs (the number of stars to draw) from the global variable *numstars*. A better way of writing the procedure would be:

```
PROCEDURE draw_stars(numstars:INTEGER);
VAR i : INTEGER;
BEGIN
   FOR i := 1 TO numstars DO WRITE('*');
END {procedure draw_stars};
```

The main program then does not need to declare a variable *numstars*, and can invoke the procedure from the for loop by using **draw_stars(i) -** i being the number of stars the procedure is to draw.

Here's another example of a procedure declaration with parameters:

```
PROCEDURE print_sequence  (  first : INTEGER ;
                             last  : INTEGER );
VAR i : INTEGER;
BEGIN
   FOR i := first TO last DO
       WRITE(i:4);
END; {Procedure print_sequence}
```

FIRST and *LAST* are called **FORMAL PARAMETERS**. We can call the procedure *PRINT_SEQUENCE* by

```
a := 4;
b := 10;
print_sequence(a,b);   {equivalently, print_sequence(4,10)}
```

A and *B* are called **ACTUAL PARAMETERS**. *A* is associated with the formal parameter *FIRST*, and *B* is associated with *LAST*. When a procedure is called, the **FORMAL PARAMETERS** receive the **VALUE** of their corresponding **ACTUAL PARAMETERS.** Thus, in the example above, WITHIN the procedure PRINT_SEQUENCE, *FIRST* assumes the value of *A*, and *LAST* assumes the value of *B*.

This method of feeding data into a procedure is called **CALL BY VALUE**, because the **FORMAL PARAMETERS** of a procedure are **ASSIGNED VALUES** by the caller.

NOTES:

1. Formal parameters are **LOCAL** to the procedure. The main program has no access to variables *FIRST* and *LAST*. Procedure *print_sequence* may change the values of *FIRST* and *LAST*, but **THIS DOES NOT AFFECT THE VALUES OF VARIABLES *A* AND *B*.**
2. Because the calling program passes values to the procedure by a process of assignment, the **ACTUAL** and **FORMAL** parameters **MUST BE ASSIGNMENT COMPATIBLE** (eg integers may be assigned to reals, but NOT vice versa).
3. There is a **ONE TO ONE** positional correspondence between the **ACTUAL** and the **FORMAL** parameters: for EACH formal parameter there must be an actual parameter of the correct type in the SAME position.
4. As with variable declarations, formal parameters of the same type may be grouped together in a list. Thus the example above could have been written as:

```
PROCEDURE print_sequence (first, last : INTEGER);
```

5. The following are all legal calls to *PRINT_SEQUENCE*

```
print_sequence (12, 20);
print_sequence (A*2, B*(A+12));
```

9.5. String Parameters - Using Type

Turbo Pascal will not accept a procedure declaration such as

```
PROCEDURE dosomething ( s : STRING[80] ); {illegal}
```

To do this, one must first declare a **STRING TYPE** using the **TYPE** statement before the **VAR** declarations at the top of the program:

```
PROGRAM prog1;
TYPE  astring     = STRING[60];
VAR   name        : astring;

   PROCEDURE CentreString ( s : astring );
   CONST screenwidth = 80;
   BEGIN
      WRITELN(s:(screenwidth+LENGTH(s)) DIV 2);
   END{procedure CentreString};

BEGIN
REPEAT
   READLN(name);
   CentreString(name);
UNTIL name = '';
END{program}.
```

NOTE that by convention the order of declaration is CONST, TYPE, VAR and PROCEDURE. What is the logic behind this?

It is important to note that the way we defined CentreString above allows us to centre strings of only up to 60 characters. This is because CentreString takes a parameter of type astring, which was defined as being a type of string with a maximum length of 60. This leads to two problems:

1. Procedure CentreString is not general enough, because it will not accept just any string as its parameter.

2. Moreover, it can only be used in a program which has defines a type called astring, thus limiting its reusability.

TP5 offers a solution to both problems by allowing us to declare string parameters to be of the generic type string (ie without a length). Parameters of type string will accept strings of any length (up to 255 characters). Also, because string is a standard type, the main program is not required to define it in a type declaration. Thus we can declare CentreString as follows:

```
PROCEDURE CentreString ( s : string );
CONST screenwidth = 80;
BEGIN
   WRITELN(s:(screenwidth+LENGTH(s)) DIV 2);
END{procedure CentreString};
```

This is a very convenient way of declaring string parameters (although it is slightly wasteful of memory), but in the case of CentreString it creates an additional problem. What is it, and how can it be solved?

9.6. Some More Standard Turbo Pascal Procedures

We have already encountered many standard (i.e. predefined) Pascal procedures, such as WRITE, READ, DELETE and INSERT. Here are two more:

GotoXY (Column,Row)

parameters:	Column is an integer (the column number - 1..80)
	Row is an integer (the row number - 1..25)
purpose:	Moves the cursor to position X,Y on the screen. The screen is 80 characters wide and 25 lines high, with position 1,1 being the top left corner. This procedure is in the CRT unit.
example:	GotoXY(6,12) moves the cursor

Val (String,Num,Error)

parameters:	STRING is a string to be converted to a number.
	NUM is an integer or real variable
	ERROR is an integer variable
purpose:	Converts the STRING into a number, which is returned in NUM. STRING must represent a numeric value - no leading or trailing spaces are allowed. ERROR is set to 0 if the conversion succeeds. Else it is set to a non-0 value.

EXAMPLES:

STRING (input)	NUM (output)	ERROR (output)
'12'	12	0
'ABC'		not 0
'12A'		not 0
' 12'		not 0
'12 '		not 0
'-12'	-12	0
'12.345'	12.345	0

9.7. Exercises - Procedures 1

1. Use the **CentreString** procedure in a program which reads in a text file and lists it centred on the screen.

2. Write a procedure which prints a string on the screen at a position (X,Y) and underlines it using '-'. The procedure should be declared as:

   ```
           PROCEDURE Uline (s:string; X,Y:INTEGER);
   ```

   Modify the procedure so that it underlines the string using any character passed to it as a parameter.

3. Write a procedure which prints a title string nicely centred on the top line of the screen. The procedure should be declared as:

   ```
           PROCEDURE DrawTitle (Title:string);
   ```

   The title should be printed in some attractive colours using the procedures **TextColor**, **TextBackground** and **ClrEol** from the CRT unit (see online help for how to use these procedures).

CHAPTER **10**

10. Procedures - 2

10.1. Variable Parameters - Parameters for Input and Output

In the previous chapter we covered **VALUE PARAMETERS**. These parameters can only be used to pass data **FROM THE CALLER TO THE SUBPROGRAM**, and not vice versa, since whatever changes the subprogram makes to the parameters are purely local and cannot affect the caller.

However, a subprogram may sometimes want to RETURN DATA back to the caller. This can be done if **VAR PARAMETERS** (sometimes called **REFERENCE PARAMETERS**) are used. An example of this is the standard procedure READLN. For example, in

```
    READLN(x);
```

the parameter x receives the data read in by the READLN procedure.

10.2. Declaring Variable Parameters

A reference parameter is declared by prefixing it with the keyword **VAR** in the procedure declaration:

```
        PROCEDURE p1( VAR a : INTEGER );
```

If we now call this procedure with

```
        p1(x);
```

then any changes to parameter A within procedure p1 will also affect the value of the corresponding argument X in the calling program. Rather than A getting the value of X, **A BECOMES X**. So VAR parameters may be used for **OUTPUT** from the procedure as well as for INPUT to it **-** they serve as a two-way channel of data between a procedure and its caller. This is called **CALL BY REFERENCE**.

DO NOT CONFUSE VAR AS USED HERE WITH VAR AS USED IN VARIABLE

DECLARATIONS.

Example :

This program calculates the permutations of **n** distinct objects taken in groups of **r** distinct objects, where the order of the objects in a group is important (eg 4-digit positive numbers that can be composed using the digits 123456 **-** n=6, r=4). The number of permutations of r out of n objects is given by

$$P(n,r) = \frac{n!}{(n-r)!}$$

```
    PROGRAM permutations;
    VAR n,r    : INTEGER;
        f1,f2  : REAL;
        perm   : REAL;

        PROCEDURE getfactorial(num : INTEGER ; VAR factorial :
  REAL);
        VAR i : INTEGER;
        BEGIN
            factorial := 1;
            FOR i := 1 TO num DO
               factorial := factorial * i;
        END; {Procedure getfactorial}

    BEGIN
       REPEAT
         WRITE('Number of different objects (n) : ');
         READLN(n);
         WRITE('Number of objects in a group (r) : ');
         READLN(r);
         getfactorial(n,f1);
         getfactorial(n-r,f2);
         perm := f1/f2;
         WRITELN (perm:1:0,' permutations.');
       UNTIL FALSE;
    END.
```

NOTES:

1. the procedure uses *NUM* for input, *FACTORIAL* for output.
2. the factorial of a number is an integer, but *FACTORIAL* is declared **REAL** because a Pascal **INTEGER** is too small to accommodate anything bigger than 7! (see if you can find the greatest factorial REAL can handle).
3. because *FACTORIAL* is a **VAR** parameter, whenever the procedure changes the value of *FACTORIAL*, the value of *F1* (in the first call) and *F2* (in the second call) also change.

Certain constraints apply when using VAR parameters:

1. A constant (value or expression) may not be passed to a procedure which expects a **VAR** parameter. eg

   ```
   getfactorial(n,6);
   getfactorial(n,f1+7);
   ```

   are **NOT legal.** This is because the procedure must be able to assign values to the VAR parameter, which it cannot do if the parameter is a constant.
2. The types of the actual parameter and a **VAR** formal parameter **MUST MATCH EXACTLY**. If a **VAR** parameter is declared **REAL**, then only **REAL** variables may be passed to it. This is because the **VAR** formal parameter is merely a different name for the actual parameter.

10.3. Exercises - Procedures with VAR Parameters

1. Modify the program PERMUTATIONS so that it finds the COMBINATIONS of selecting **r** objects from **n** (eg the number of ways 4 people can be chosen from 6 people). In combinations, the order of selection is not important. The combinations of **r** out of **n** objects is found by

$$C(n,r) = \frac{n!}{r!(n-r)!}$$

2. The uppercase characters in the ASCII set have codes from 65 to 90. The corresponding lowercase characters have codes from 97 to 122. Thus, if C is an upper case character, its corresponding lowercase character can be obtained by

   ```
   Chr(Ord(C)+32).
   ```

   Write a procedure

   ```
   PROCEDURE LowerCaseChar(var C:Char);
   ```

   which converts its parameter C to lowercase, but only if it is an uppercase character (otherwise it leaves it unchanged). Note how the var parameter C here serves to carry data both into and out of the procedure.

10.4. Using String Types in Var Parameters - Relaxing Type Checking

Because of the constraint that the types of the actual parameters and **VAR** formal parameters must match exactly, procedures employing **VAR** parameters of type **STRING** will only accept strings of the **EXACT length declared in the procedure**. Normally TURBO PASCAL will check for this during compilation, and will give an error if you attempt to do something like this:

```
PROGRAM x;
VAR s1 : STRING[40];

   PROCEDURE process_string ( VAR s : string);
   BEGIN
      { procedure body }
   END;

BEGIN
   READLN (s1);
   process_string(s1);  { error here, because
                          process_string only accepts
                          parameters which are strings of 255
                          characters. }
END.
```

This is a nuissance because it means that you cannot write a procedure to process ANY string. You can get around this problem by using the **COMPILER DIRECTIVE**

{$V-}

somewhere at the top of your program. A **COMPILER DIRECTIVE** controls the way the compiler behaves. In this case {$V-} tells the compiler **not** to check that **VAR** string parameters match exactly (however, it will still complain if you try to pass, say, an **INTEGER** to a **REAL VAR** parameter). We say that this directive **RELAXES TYPE CHECKING**.

Another method you can use to relax var-string type checking is to set the **var-string checking** option from the **Options|Compiler** menu to **Relaxed**. If you do this, all programs will be compiled with relaxed string type checking, whether they have the {$V-} directive or not. Having set this option from the menu, you should also save the options using **Options|Save options** so that everytime TP5 loads it will retrieve the option settings. For this to work, you should save the options to a file called TURBO.TP in your home directory - TP5 will only automatically load the saved options if it finds them in the directory you are in when you run it.

10.5. Procedures Which Call Other Procedures

It frequently happens that a procedure needs to call another procedure. Consider a procedure called LowerCaseString, which converts a whole string to lowercase. This procedure will need to call the LowerCaseChar procedure you wrote earlier for each character in the string, as follows:

```
PROCEDURE LowerCaseString(VAR s : STRING);
VAR i : INTEGER;
BEGIN
   FOR i := 1 TO LENGTH(s) DO
      LowerCaseChar(s[i]);
END;
```

A procedure may call other procedures subject to the condition that **THE PROCEDURE BEING CALLED MUST APPEAR BEFORE THE CALLER IN THE SOURCE CODE**. Since the main program always appears last in the source code, it may call any procedure.

10.6. Creating a Library Unit

By now you should be familiar with Turbo Pascal's library units such as CRT and PRINTER, containing precompiled functions and procedures which can be called by any program simply by including the unit in the program's **uses** statement.

You can write your own units in which to store subprograms of a general nature for use in later programs. This saves work and is an important programming principle called **CODE REUSE**.

Subprograms intended for a library unit should adhere to the following design principles:

- they should be of general use.
- they should be thoroughly debugged.
- any communication with the calling program should be through parameters.

From now on you should be on the look out for useful subprograms to add to your library unit or units. Any units you write now will prove of great help when you start on your project.

10.7. The Structure of a Turbo Pascal Unit (TPU)

A TPU **exports** objects to other programs or units which use it. These objects may include constants, types, variables and subprograms - any program using the unit will have access to these exported objects as if they were defined in the program itself.

The structure of a TPU is as follows:

Interface part - public	`UNIT <unit name>;`
	`INTERFACE`
	`USES <list of units>`
	`<public declarations>`

Implementation part - private	`IMPLEMENTATION`
	`USES <list of units>`
	`<private declarations>`
	
	`BEGIN`
	`   <unit initialization>`
	`END.`

10.7.1. Interface part

The interface part of a unit contains the declaration of all exported objects (called **public declarations**). These are the objects accessible to other programs using this unit.

Unit Name	This must be the same as the filename under which the unit source code will be saved, except that it must not contain an extension. Thus, if we were to save the unit as MYUNIT.PAS, the unit name should be MYUNIT. When TP5 compiles the unit, it will generate a file called MYUNIT.TPU instead of an executable program. Note that yu cannot run a unit **-** you can only use it.
Uses statement	A list of all other units required for the public declarations in the interface part. If no other units are needed, this statement should be omitted.
Public declarations	Constants, types and variables to be exported should be declared here as they would be declared in a normal program. In the case of subprograms, however, only the heading should be listed **-** the actual subprogram will be written in the implementation part.

10.7.2. Implementation part

The implementation part of a unit contains the actual subprogram code corresponding to the subprograms declared in the interface part. It may also contain other declarations not declared in the interface **-** such declarations are private to the unit and invisible to any program using the unit.

The implementation part may optionally contain unit initialization code, which is written between the final BEGIN and END. (and thus corresponds to where the main program would be in a normal program). When a program uses a unit, the unit's initialization code (if any) is executed before the main program begins. The initialization code may be used to perform any startup processing **-** for example to initialize any variables required by the unit. If no initialization code is required, the BEGIN may be omitted, but the END. statement must still be written.

10.8. Example of a TPU

Here is a simple TPU called SCREEN which provides support for programs which need to display things in attractive ways. As it stands, the TPU only exports one constant, two variables and a single procedure which displays a colourful title centred on the top line of the screen, but the unit can be easily extended.

```
UNIT Screen;
INTERFACE

   CONST ScreenWidth = 80;
   VAR   TitleBackground,
         TitleColour : INTEGER;

   PROCEDURE DrawTitle(Title : STRING);

IMPLEMENTATION
   USES Crt;

   PROCEDURE DrawTitle(Title : STRING);
   BEGIN
      TextBackground(TitleBackground);
      TextColor(TitleColour);
      GOTOXY(1,1);
      ClrEol;
      WRITE(Title : (LENGTH(Title) + ScreenWidth) DIV 2);
   END;
```

```
{ Unit initialization }
BEGIN
   TitleBackground := Green;
   TitleColour     := Red;
END.
```

NOTES:

1. The DrawTitle procedure draws the title using the TITLEBACKGROUND and TITLECOLOUR variables. These two variables are initialized in the unit's initialization code to GREEN and RED respectively, but since these variables are declared in the interface part any program using the unit can alter them to any colours it chooses.
2. The implementation unit uses the standard CRT because it needs the two procedures GOTOXY and CLREOL, as well as the colour definitions GREEN and RED, which are all exported by CRT. The implementation part, on the other hand, does not need to use any units.

10.9. Using a TPU

Having compiled the SCREEN unit (generating the library unit SCREEN.TPU), we can use it in a program just like Turbo Pascal's standard units. For example:

```
PROGRAM TestScreen;
USES Screen, CRT;

BEGIN
   TitleColour := Blue;
   DrawTitle('Testing the SCREEN unit.');
END.
```

Note how this test program can access the variable TITLECOLOUR, which is declared in the SCREEN unit. It also needs the CRT unit in order to have access to the constant BLUE.

When you use a unit in a program, TP5 needs to be able to locate the TPU file when it is compiling the program. If the TPU file is not in the current directory, you must tell TP5 where to find it. You do this by using the **unit directories** option in the **Options|Directories** menu. This option contains a list of directories, separated by **semicolons**. When a unit used by a program is not in the current directory, TP5 will search for it in all these directories in order.

You should add the name of the directories containing your own units to the front of the **unit directories** list. Be careful not to delete any of the directories already listed, since these are required by TP5 to locate the standard units such as CRT. The best thing to do is to place all your units in a single directory and add this directory to the list. Remember to save the modified options to TURBO.TP so that they will be automatically reloaded everytime you run TP5.

10.10. Exercises - Writing a TPU

Create a unit called EXTRAS containing the following procedures:

1. **SWAP_INT (var num1,num2 : integer)**

   swaps two integer variables, NUM1 and NUM2, setting NUM1 to the value of NUM2 and NUM2 to the value of NUM1. Useful for sorting.

2. **SWAP_STR (var str1,str2 : maxstring)**

   swaps two string variables.

3. **CENTRE_STRING (str : string; width : integer)**

   prints the string STR centred in a field WIDTH characters wide. Useful for screen formatting.

4. **UP_CASE (var str : string);**

   converts STR to upper case.

5. **PRINT_STRING_AT (str: string; posX,posY: integer; UpCase: boolean)**

   prints the string STR starting at screen positions POSX and POSY. If the parameter UPCASE is TRUE, then the string is converted to uppercase before printing, otherwise it is printed as is. Use the procedure GotoXY to print at a location.

Write a program to test unit EXTRAS.

CHAPTER 11

11. Functions

11.1. What are Functions?

In Computer Science, a function is a **SUBPROGRAM WHICH YIELDS A (SINGLE) VALUE OF A PARTICULAR TYPE**. Examples of functions are:

sqr(i)	which yields the result i^2
sin(x)	which yields as its result the sine of x radians
upcase(c)	which yields as its result the uppercase version of the character c (if c is lowercase).

It is important to note that, while procedures are like **COMMANDS** which perform some action, functions are like **VALUES**. Thus, **SQR(2)** is **THE SAME** as **4 -** we can use the function SQR(2) wherever we can use 4.

Note also that the concept of functions in computer science is similar, but NOT identical, to the concept of functions in mathematics. Thus SQRT (the square root routine) is a function in computing, but is NOT a function in the mathematical sense (bacause there are 2 square roots for any positive number, while a function can only return ONE value).

Exercise

Which of the following statements are legal, and which are not?

1 write(UPCASE(c));

2 sin(3.2);

3 r := sqrt(x);

4 IF SQR(i) < 100 THEN WRITE('ok');

5 IF READLN(a) = 4 THEN WRITE('ok');

11.2. Declaring Simple Functions

Example of a function declaration:

```
FUNCTION factorial (num : INTEGER) : REAL;
VAR i     : INTEGER;
     fact  : REAL;
BEGIN
   fact  := 1;
   FOR i := 1 TO num DO
      fact := fact * i;
   factorial := fact;
END; {Function  factorial}
```

NOTES:

1. A function declaration is similar to a procedure declaration except that:

   a. *the reserved word **FUNCTION** is used*

   b. *the function itself must be given a type (here REAL).*

2. Somewhere in the body of the function, the function name must be assigned a value **AS IF IT WERE A VARIABLE NAME**. The calling program receives this value as a result of calling the function.

3. Although functions always return a value to the caller by means of the function name, they may also return values by using VAR parameters like procedures. This, however, should be avoided.

The function **FACTORIAL** is better suited for calculating factorials than the procedure we saw in an earlier chapter. To calculate the permutation *P(n,r)* we use:

```
perm := factorial(n) / factorial(n-r);
```

Thus, function calls (unlike calls to procedures) can be used in expressions as if they were variables. This is similar to the use of inbuilt PASCAL functions such as **SIN**, **POS**, **LENGTH**, **RANDOM** etc.

11.3. Exercise - Writing Functions

Add these 3 functions to your EXTRAS unit, and test them by using the unit in a testbed program.

1. **FUNCTION** MAX_OF (a , b : **INTEGER**) : **INTEGER**;
   Returns the bigger of A and B.

2. **FUNCTION** FACTOR_OF (a , b : **INTEGER**) : **BOOLEAN**;
   Returns TRUE if A is a factor of B. Else returns FALSE.

3. **FUNCTION** ASK_YN (prompt : **STRING**) : **BOOLEAN**;
   Writes PROMPT on the bottom line of the screen using TITLECOLOUR and TITLEBACKGROUND (see the DRAWTITLE procedure in the previous lesson), and waits for the user to press one of the characters 'Y', 'y', 'N' or 'n' (use READKEY). Returns TRUE if user presses 'Y'or 'y', FALSE if user presses 'N' or 'n'.

11.4. Passing Arrays as Parameters to Subprograms

Arrays can be passed to subprograms (functions and procedures) just like any other type. However, as with strings (which are, in fact, ARRAY OF CHAR), Pascal will not allow us to write, for example:

```
FUNCTION sum_array (a : ARRAY[1..100] OF INTEGER) : INTEGER;
```

Instead, we must define an array type, using for example:

```
TYPE Array100 = ARRAY[1..100] OF INTEGER;
```

and then use the type *ARRAY100* in the parameter declarations:

```
FUNCTION sum_array (a : Array100) INTEGER;
VAR sum,i : INTEGER;
BEGIN
  sum := 0;
  FOR i := 1 TO 100 DO
    sum := sum + a[i];
  sum_array := sum;
END;
```

11.5. Small Project - Implementing a Stack

In this section we will develope a small project which demonstrates the principles of top-down design, and how a program may be build up from many simple subprograms. The program implements an integer **STACK** - operations on the stack are performed by subprograms. The subprograms share some common data structures, thus forming a **MODULE**.

11.6. Nature of problem to be Solved

A stack is a LIFO structure. Associated with a stack are:

THE STACK POINTER	which keeps track of the items on the stack
THE MAXIMUM STACK SIZE	which places a limit on the number of items which the stack can hold.

The operations on which can be performed on a stack are:

CLEAR_STACK	empties the stack of any items
IS_FULL	check whether the stack is full
IS_EMPTY	check whether the stack is empty
POP_ITEM	get the last item we saved on the stack
PUSH_ITEM	save a new item on the stack
ITEMS_ON_STACK	returns the number of items on the stack

11.7. Designing the Interface

We will collect all the functionality associated with the stack and its handling into a single unit called INTSTACK - since this is going to be a stack of integers. Any program needing to use an integer stack can then simply use the INTSTACK unit.

It is important at this stage to clearly separate those objects which the unit needs to export from those objects which can remain hidden within the implementation part of the unit. Programs using INTSTACK will need to perform the seven operations listed above, and therefore the unit must export subprograms which implement these operations. In fact, these operations are ALL that programs using the stack unit will require - everything else can be hidden in the implementation part.

```
UNIT IntStack;
INTERFACE

 PROCEDURE Clear_Stack;
 FUNCTION  Is_Full : BOOLEAN;           {TRUE if stack is full}
 FUNCTION  Is_Empty : BOOLEAN;          {TRUE if stack is empty}
 FUNCTION  Pop_Item : INTEGER;          {Pops item from non-empty stack}
 PROCEDURE Push_Item (Item : INTEGER);  {Pushes item onto non-full stack}
 FUNCTION  Items_On_Stack : INTEGER;    {How many items on stack}
```

If the stack is empty, Pop_Item is undefined - i.e. it will not return a valid value. Programs using the unit are thus expected to first check that the stack is not empty (by calling Is_Empty) before calling Pop_Item. Similarly, Push_Item is undefined if the stack is full.

11.8. Designing the Implementation

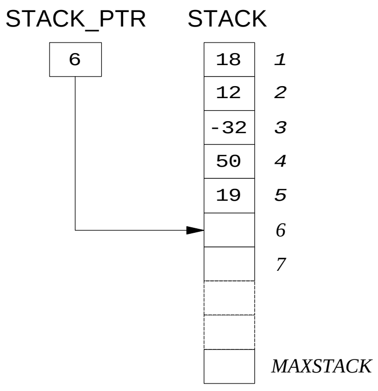
The implementation details - how the stack is actually implemented and how the procedures and subprograms work - are hidden from any program using the unit by placing them in the implementation part. This way, we can change the way the stack is implemented without affecting programs using the unit. This is sometimes called **data abstraction**.

DATA REPRESENTATION

1. The stack will be represented as an integer array [1..MAX_STACK] called STACK.
2. MAX_STACK is a constant.
3. Items are stored starting at location 1 (this is called an ASCENDING STACK). The last item in a FULL stack occupies location MAX_STACK.
4. A variable, STACK_PTR, points to THE NEXT FREE LOCATION. The stack pointer takes on values 1 (when the stack is empty) to MAX_STACK+1 (when it is full).

STACK, STACK_PTR and MAX_STACK are declared within the IMPLEMENTATION part of INTSTACK so that they are accessible to all the subprograms within the unit, but are inaccessible to programs using it.

The diagram shows how the stack would look after 5 items have been pushed.



IMPLEMENTATION OF THE STACK OPERATORS

The implementation of the functions and procedures listed in the interface part is quite straightforward.

UNIT INITIALIZATION

We will use the unit initialization code to initialize the stack by calling Clear_Stack.

11.9. Exercises - Stack Unit

1. Implement the INTSTACK unit described above.
2. Test the unit by writing a program which uses INTSTACK to fill the stack with the numbers 1,2,etc using PUSH_ITEM (you should NOT use the value of the constant MAXSTACK in your program, since this is hidden within the implementation part of the unit). When the stack is full (i.e. Is_Full becomes TRUE), the program should print out the contents of the stack as returned by POP_ITEM.
3. Add an exported function Peek_Item to the INSTACK unit which returns the item at the top of stack without actually popping it. Like Pop_Item, this function is only defined if the stack is not empty.
4. Add an internal procedure STACKERR to INTSTACK which prints the error message *STACK ERROR* on the bottom line of the screen and halts the program (use the standard procedure HALT). Modify Pop_Item and Peek_Item so that if they are called when the stack is empty they will call STACKERR, causing the program to abort. Similarly modify Push_Item to call STACKERR if it is called when the stack is full. Note that it is pointless to export the procedure STACKERR.

CHAPTER 12

12. Enumerated Types, Ranges and Sets

12.1. Enumerated Types

There are many programming situations in which we know that a variable can assume only a small number of distinct values, such as, for example the seven days of the week. There are various things we could do:

Method 1

We could use integers to represent the seven days of the week, e.g. Sunday = 0, Monday = 1 etc, and in our program write something such as:

```
    IF Today = 5 THEN IssuePayslips;
```

to mean that if the integer variable Today indicates a Friday (day 5) then the procedure IssuePayslips should be called. The problem with this method is that it is not very readable, with the result that it is very easy to make a mistake when writing the program.

Method 2

A better method would be to define 7 integer constants at the beginning of the program, or in a unit, thus:

```
    CONST Sunday = 0; Monday = 1; Tuesday = 2; {...etc}
```

so that we could then write:

```
    IF Today = Friday THEN IssuePayslips;
```

This is very much more readable, but we might have to define quite a large number of constants. Moreover, the variable TODAY would probably be declared as an integer, and therefore Pascal would not complain if we were to write something such as:

```
    Today := 100;
```

which is clearly a mistake. We would like to tell the compiler that TODAY should only be allowed to assume one of the values from 0 to 6.

Method 3

The prefered method is to us an **enumerated type**, which is essentially a new type created by the programmer and defined within a **type** declaration as shown by the following example:

```
        TYPE
            day =
      (sunday,monday,tuesday,wednesday,thursday,friday,saturday);
```

We can now declare variables of type **day** as follows:

```
        VAR    holiday,workday : day;
```

NOTES:

1. Any enumerated type such as the one illustrated above can be treated just like any other type with a few exceptions. Thus the following statements are all valid:

```
VAR
   daily_pay : ARRAY[monday..friday] OF REAL;
   today,pay_day,work_day : day;
   total_pay : REAL;

pay_day := friday;

total_pay:=0;
FOR work_day := monday TO friday DO
   total_pay := total_pay + daily_pay[work_day];

CASE today OF
   sunday    :  WRITELN('Sleep late.');
   saturday  :  WRITELN('Wash car.');
   ELSE WRITELN('Go to work.');
END;

IF today = pay_day THEN   WRITELN('Today is pay day.')
ELSE                      WRITELN('Not pay day yet.');
```

2. Variables of an enumerated type **cannot** be used with input/output commands such as **READLN** and **WRITELN**. Thus statements such as

```
   READLN(today);
   WRITELN(pay_day);
```

   are **not** accepted.

3. The only operators that may be used with enumerated types are the relational operators =, <, >, <> which all return **boolean** values.

   Example:

```
   monday < tuesday        {returns true}
   tuesday = wednesday     {returns false}
   thursday > friday       {returns false}
```

   The ordering of these values is determined by the order in which they appear in the type declaration. The function **ORD()**, which was previously used to find the ASCII code of a character, can also be used to find the **ordinal number** of the identifier in the list that defines the enumerated type. The 1st value in the list has ordinal number 0, the 2nd value has ordinal number 1 etc.

   Example:

```
   ORD(monday)  {returns 1}
```

4. A value may not belong to more than one type. Thus the following type declaration is not valid since the value **tomato** cannot belong to both **fruit** and **vegetable** types.

```
TYPE
      fruit = ( apple, orange, lemon, banana, tomato );
      vegetable = ( onion, potato, tomato, pea );
```

12.2. Subrange Types

A subrange type is defined by two constants, both of the same type, within a type declaration.

Example:

```
TYPE
  digit   = '0'..'9';
  letter  = 'a'..'z';
  weekday = monday..friday;
  age     = 0..100;
```

NOTES:

1. The constants used to define the subrange are called the **lowerbound** and the **upperbound**. The upperbound must be greater than or equal to the lowerbound. Only ordinal types (integer, char, boolean, and enumerated types) may be used to define the bounds (thus, **real** numbers and **strings** are not allowed).

2. A variable of subrange type is only allowed to take values within the specified bounds. A runtime error occurs if the variable exceeds its range, thus making it easier for the programmer to detect the presence of some related logical error. Turbo Pascal only performs runtime range-checking if you set the **range checking** item in the **Options|Compiler** menu to **ON**.

3. The two declarations necessary for a subrange variable can be combined into one:

```
VAR
    work_day,pay_day   : monday..friday;
        retirement_age,school_leaving_age : 0..100;
```

4. A subrange cannot have gaps. Thus:

```
consonants = 'b'..'d','f'..'h','j'..'n','p'..'t','v'..'z'
```

   is **not** acceptable.

5. Subranges should be used as much as possible since they offer the following advantages:

   i. Program readability is improved, because the allowed range of possible values of a variable is clearly stated.

   ii. Logical mistakes can be easily detected if range-checking in Turbo Pascal is enabled.

12.3. Set Types and Sets

A set is a **collection** of objects of the same type. We can define a set based on any **ordinal** type (i.e. char, boolean, integer subranges, and user-defined enumerated types).

To declare a set type, we use:

```
TYPE identifier = SET OF base type;
```

The objects which can be contained in a set include all those values belonging to the base type.

For example, suppose we need to keep a record of which subjects students learn. We can first define an enumerated type of all allowed subject options, and a set type which can contain any number of allowed subjects:

```
    TYPE   Subjects =
  (English,Maths,Computing,Accounts,SOK,Economics);
        SubjectSet = SET OF Subjects;
```

Next, we can define a SubjectSet for each of the students:

```
VAR
Mark,Joan,Petra : SubjectSet;
```

and initialize them to the appropriate subjects in the program:

```
Mark   := [English,Computing,SOK,Economics];
Joan   := [Accounts..Economics];
Petra  := [English..Computing,SOK];
```

NOTES:

1. The contents of a variable of type **SubjectSet** is a **set** (collection) of values belonging to the type **Subjects**. A set is represented by a list of its members enclosed within square brackets. The members should generally be seperated by commas. However, if some members are consecutive values of the base type, a subrange notation may be used.

2. A set may have no members. Such a set is called the **empty set** and is written as [ ].

3. The relational operators may be used to compare sets.

=	denotes set equality
<>	denotes set inequality
<=	denotes "is contained in"
>=	denotes "contains"

   Note that a set A is said to contain a set B if every member of B is also a member of A.

4. The **union** (+) of two sets is a set containing the members of both sets. The **intersection** (*****) of two sets is a set containing only those objects which are members of both sets. The **difference** (-) of two sets is a set conatining all the members of the first set that are not members of the second.

   Example:

```
IF (Mark * Joan) = [] THEN
   WRITELN('Mark and Joan take no subjects in common.');
Mark := Mark + [Maths]; { Now Mark takes Maths too.}
Joan := Joan - [Economics]; { Joan has dropped Economics.}
```

5. The reserved word **IN** is used to test whether a particular element is in a set, and returns a boolean value accordingly.

   Example:

```
   IF Economics IN Petra THEN WRITELN('Petra takes
Economics.');
```

6. **READ** and **WRITE** cannot be used with sets. If we want to list all the elements of a set, we can use a loop to test each possible element, and print out those which are in the set:

```
WRITELN('Mark takes: ');
FOR Subject := English TO Economics DO
IF Subject IN Mark THEN
CASE Subject OF
   English   : WRITELN('English');
   Maths     : WRITELN('Mathematics');
   Computing : WRITELN('Computer Studies');
   Accounts  : WRITELN('Accounts');
   SOK       : WRITELN('Systems of Knowledge');
   Economics : WRITELN('Economics');
END;
```

7. Sets can be very useful in validation.

   - Range checks such as ('0' <= digit) **and** (digit <= '9')
     can be shortened to digit **IN** ['0'..'9']

   - An expression such as (x=2) **OR** (x=8) **OR** (x=9)
     can be shortened to x **IN** [2,8,9]

12.4. Exercises - Enumerated types, Ranges and Sets

1. Show how

   i. a **case** statement, and
   ii. an array

   may be used to print values of an enumerated type.

   Since enumerated types may neither be read nor written, I/O of enumerated types requires conversion functions to translate between the enumerated type and a string representation. Given the enumerated type:

   ```
   TYPE suite = (clubs, diamonds, spades, hearts);
   ```

   and a variable CARD of type SUITE, write two functions:

   ```
   FUNCTION StringToSuite (s:str10) : suite;
   FUNCTION SuiteToString (s:suite) : str10;
   ```

   to convert between type string and type suite. Hence write a program which reads in a string from the user, converts it to suite for assignment to variable CARD, and then converts CARD back to a string for display.

2. Given

   ```
   TYPE  numberset = SET OF min..max;
   ```

   where **min** and **max** are integer constants and **min** < **max**, write a procedure **printset** which prints the value of a variable of type **numberset**. e.g. the set whose members are 3, 7, 11, and 9 should be output as *[3,7,9,11]*.

3. Given the type **`CHSET = SET OF CHAR`** write a function

   ```
   FUNCTION ReadChar(allowed:CHSET):CHAR;
   ```

   which waits for the user to enter one of the allowed characters in the set, and returns that character. The function should not accept any characters except those allowed (use READKEY). For example, the following can be used to get the user to enter a hexadecimal digit into the character variable C:

   ```
   C := ReadChar(['0'..'9','A'..'F','a'..'f']);
   WRITELN('You entered ',C);
   ```

4. Write a program which generates 20 **different** random numbers between 0 and 100. (**Hint**: Use a set to 'remember' which values have already been generated).

CHAPTER 13

13. Static Records and Random Files

13.1. The RECORD Type

A **record** is a structure consisting of a fixed number of components called **fields**. The fields may be of different types and each field is given a name (called the **field identifier**) which is used to select it. The following example defines a record type **car** which contains the fields **licence_no**, **make**, **model**, **colour**, **miles_to_gallon, cylinders**.

```
TYPE
  colourtype =  ( red   , green , blue , white , yellow ,
                  black , silver );
  CarRec =
   RECORD
       licence_no      :  STRING[7];
       make,model      :  STRING[10];
       colour          :  colourtype;
       miles_to_gallon :  REAL;
       cylinders       :  INTEGER;
   END;
```

Record variables of type CarRec can now be declared as follows:

```
VAR
  mycar,myoldcar,sportscar:CarRec;
```

Each record variable of type **CarRec** can be thought of as a variable which consists of **six** component variables of various types. To access a particular component, one should specify the name of the record variable followed by a period '**.**' and the field identifier of the required component. These constitute a **record selector** which may be used just like any other variable of the same type.

Example:

```
sportscar.licence_no      := 'X-1234';
sportscar.make            := 'Ferrari';
sportscar.model           := 'TestaRossa';
sportscar.colour          := red;
sportscar.miles_to_gallon := 18.6;
sportscar.cylinders       := 20;

WRITELN('My car is a ',mycar.make,' ',mycar.model);

gallons_needed := distance / mycar.miles_to_gallon ;
```

NOTES:

1. A record variable can be *copied* into another by means of an assignment statement such as

```
mycar := sportscar;
```

This statement is equivalent to six assignment statements which *copy* each component one at a time

```
mycar.licence_no    := sportscar.licence_no;
mycar.make          := sportscar.make;
mycar.model         := sportscar.model;
etc.
```

2. The names of the fields of a particular record type must be unique. However, different record types may make use of the same field identifier (with different types if necessary). The following example type can therefore also be declared along with **CarRec** :

```
TYPE
  BikeRec =
    RECORD
      make,model :  STRING[8];
      colour     :  colourtype;
      gears      :  INTEGER;
    END;
```

3. There are no operators that can be used with records. Also, there is no ordering whatsoever associated with records (i.e. one cannot say that a record variable is greater than another), and therefore records cannot be compared.
4. Records may be passed to procedures and functions as parameters, but a function cannot return a record as its result. However, records can be passed as **VAR** parameters so that subprograms can change data in a record.
5. Pascal is very flexible as far as type declarations are concerned. We may have **arrays of records**, **records of arrays**, and even **records of records** as shown below:

```
TYPE
   monthtype = (    Jan , Feb , Mar , Apr , May , Jun ,
                Jul, Aug , Sep , Oct , Nov , Dec );
   DateRec =
      RECORD
        day      : 0..31;
        month    : monthtype;
        year     : 1900..2099;
      END;

   PurchaseRec =
      RECORD
        date        :  DateRec;
        sold_car    :  CarRec;
        buyer       :  RECORD
                        name    : STRING[20];
                        address : STRING[50];
                        tel_no  : ARRAY[1..3] OF STRING[6];
                       END;
        amount     : REAL;
      END;


VAR
   cars_for_sale   : ARRAY [1..200] OF CarRec;
   purchases       : ARRAY [1..200] OF PurchaseRec;
```

In this example we would use the record selector **purchases[5].buyer.tel_no[2]** in order to refer to the 2nd telephone number of the buyer involved in the 5th purchase.

Exercise:

Write down the record selectors required to access each of the following items of data:

1 Month in which 4th purchase was made.

2 Colour of 2nd car in table of cars for sale.

3 No. of cylinders of car sold in 9th purchase.

4 Name of buyer involved in **i**th purchase.

5 First character of name of buyer involved in **i**th purchase.

6 jth digit of the 3rd telephone number of the buyer involved in the **i**th purchase.

6. It is often necessary to access different components of the same record several times in a small section of the program. The **WITH** statement can be used to avoid repeating the record identifier each time.

Syntax: **WITH <record identifier R> DO <statement S>**

Within the statement **S**, components of the record **R** may be referred to by field name alone. The following code displays some information about the first 5 cars available for sale:

```
FOR i := 1 TO 5 DO
WITH cars_for_sale[i] DO
BEGIN
   WRITELN('Make  :',make);
   WRITELN('Model :',model);
   WRITELN('No. of cylinders :',cylinders);
END;
```

Exercise:

Using the with statement, write program sections to display the following:

i) The make, model, and cylinders fields of the cars sold in the first 10 purchases.

ii) The dates of the first 5 purchases in the form dd mm yyyy (remember that the month field is based on an enumerated type - use ORD as appropriate).

iii) The make, model of the 3rd car sold together with the amount it was bought for and the name of its buyer.

13.2. Files of Records

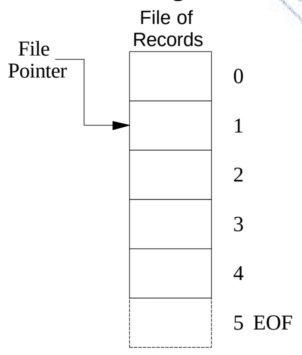
The previous section outlined the organization of data for a typical data processing application whereby records of available cars for sale and purchases made are kept. These two tables of information were stored in arrays, thus giving a useless system since the data would be lost on switching off. Moreover, arrays are of a fixed size. These records should instead be stored on disk as files. We can redefine the variables **cars_for_sale** and **purchases** as follows:

```
TYPE
  cars_for_sale  :  FILE OF CarRec;
  purchases      :  FILE OF PurchaseRec;
```

Thus **cars_for_sale** and **purchases** are now file variables rather than arrays. As in the case of **text** files, the file variable is needed in order to identify the file on which an input/output (I/O) operation is to be performed. Unlike **text** files however, all records are now of equal length. This allows **random** (or **direct**) **access** to records from the file as we shall see later.

Records in a TP5 file are indexed by their **position**. Record positions start at 0. Pascal maintains a **file pointer** which indicates the position of the **current record** in the file. The current file record can be read into a variable of the same record type using **READ**, and a record can be written to the current file position using **WRITE**.

The diagram shows a file of 5 records **-** numbered 0 to 4 **-** with record 1 being the current record. The file pointer can be moved using **SEEK** to any position between 0 and 5, position 5 being the end of file (EOF) position. Attempting to move the file pointer to a position greater than 5 will cause a runtime error.



e can read a record from any position between 0 and 4, but attempting to read a record from position 5 (EOF) will result in a runtime error, because there is no record at this position. We can write a record to any position in the file. Writing a record to any of positions 0 to 4 will simply cause the record which is already there to be overwritten. Writing a record to the EOF position will cause the file to grow by 1 record **-** i.e. the record will be **appended** to the file. After every READ or WRITE operation, the file pointer is automatically advanved to the next record position.

13.3. Opening a Record File

The procedure for opening a record file is identical to that for opening a text file.

ASSIGN(<file variable>,<filename string>)

As for text files, this associates a DOS filename with a file variable.

REWRITE(<file variable>)

As for text files, creates and opens a new file with the name previously assigned to the file variable and positions the file pointer at the start of the file.

RESET(<file variable>)

As for text files, this opens an EXISTING file for processing. The file pointer is positioned at the beginning of the file (record 0). If the file does not exist then an I/O error is returned.

The file should be closed to ensure that all records written to the file are actually saved to disk. A typical dataprocessing application using record files would normally open all the files it requires at the beginning of the program and close all its files before exiting. Such an application would also need to check whether the data files already exist before opening them, creating new empty data files if they do not yet exist. Here's an example of how this can be done:

```
VAR cars_for_sale : FILE OF CarRec;

PROCEDURE OpenFiles;
CONST CarsFileName = 'CARS.DAT';
BEGIN
   ASSIGN(cars_for_sale,CarsFileName)
   IF FileExists(CarsFileName) THEN RESET(cars_for_sale)
   ELSE REWRITE(cars_for_sale);
END;
```

Note that the cars_for_sale file variable is global to the whole program so that all procedures needing to process records in the file have access to it. At the beginning of the main program, procedure OpenFiles is called to open all necessary data files. OpenFiles uses RESET or REWRITE depending on whether the data file already exists or not. Function FileExists is not a Pascal function **-** we'll see how to write such a function later on.

13.4. Reading and Writing Records

The standard I/O procedures READ and WRITE are also used for transferring records between memory and files (READLN and WRITELN cannot be used in this case).

READ(<file variable>,<var1>,<var2>,...,<var*n*>)

Reads the next ***n*** records from the file and places them in the record variables **var1,...,var*n***. Note that these variables must be of the same record type as the file component type. After each read, the file pointer is advanced to the next record so that at the end, the file pointer points to the start of the record following that read into **var*n***.

WRITE(<file variable>,<var1>,<var2>,...,<varn>)

Writes the record variables **var1** up to **var*n*** (having the same type as the component type of the file) to the file starting from the current record position. After each write operation, the file pointer is advanced to the start of the next record.

SEEK(<file variable>,<position>)

Positions the file pointer to the record indicated by **position**. The first record has **position 0**, the second has **position 1** etc. It is this statement which allows us to perform random access.

It is important to remember that records in a file cannot be manipulated unless they are first read into a memory variable. Since usually records are only processed one at a time, a single memory record variable (called a **buffer**) will suffice no matter how large the file is.

13.5. Other File Operations

CLOSE(<file variable>)

Closes the file associated with the file variable making the latter available for use with a different file.

ERASE(<file variable>)

Erases the corresponding file from disk. Should not be used on an open file.

RENAME(<file variable>,<new filename string>)

The disk file associated with the file variable is renamed to the given name. This should never be used on an open file.

13.6. File Functions

EOF(<file variable>)

Returns **true** if the file pointer is positioned at the end of the file (i.e. **after** the last record), otherwise returns **false**.

FILEPOS(<file variable>)

Returns an integer which is the current position of the file pointer within the file. Positioning starts from 0 as for **SEEK**.

FILESIZE(<file variable>)

Returns the total number of **records** in the file. If this returns 0 then the file is empty. In order to position the file pointer to the end of the file **f**, one should use

```
SEEK(f,FILESIZE(f));
```

Subsequent writes will cause the file to grow.

13.7. Handling File I/O Errors

If an **I/O error** occurs when using any of the above procedures and functions, execution of the program is terminated with a run-time error unless the compiler directive **{$I-}** is used. This directive will not stop execution whenever an I/O error occurs. However, after each I/O operation, all I/O is suspended until the function **IOResult** is called. This returns an integer which is either zero (if the operation was successful) or a positive number which indicates the type of error encountered. It is then left up to the programmer to handle errors. Directive **{$I+}** cancels the **{$I-}**.

One situation where this directive is indispensible is when checking whether a file already exists or not. This could be implemented as follows:

```
FUNCTION File_Exists(filename:STRING):BOOLEAN;
VAR
  f:TEXT;
BEGIN
  ASSIGN(f,filename);
  {$I-}
  RESET(f);
  CLOSE(f);
  {$I+}
  File_Exists:=(IOResult=0);
END;
```

Without the {$I-} directive, the RESET procedure would halt the program with a runtime error if the file did not exist. The final line of the function returns TRUE or FALSE depending on whether IOResult is 0 (i.e. RESET did not cause an error), or non-0 (i.e. RESET could not open the file - obviously because the file does not exist).

13.8. Using Random-Access Files - Some Notes

1. Since records in a file are most easily accessed by their position within the file (ie the record number), it makes sense to use the record number as the key. Of course, it is pointless to store the record number as part of the record itself.

2. In many applications, record 0 is not used, records being numbered from 1 upwards. In this case, a **DUMMY** record number 0 should be stored when the file is created. This ensures that when the real records are stored, they will occupy positions starting at 1.

3. Since records in a file cannot be modified directly, a record variable of the same type as the records in the file is used as a **DATA BUFFER** in the program. The program then proceeds as follows:

```
SEEK   (f, recordnumber);
READ   (f, bufferrecord);
    modify bufferrecord;

SEEK   (f, recordnumber);
WRITE  (f, bufferrecord);
```

13.9. Exercises - Random Access Files

Copy the subdirectory COMMON\RECORDS into a subdirectory of your own home directory. There are 3 files in this directory:

STUDDEF.PAS is a unit containing the definition of a record type called STUDENTREC. This is a very simple type of record containing some student particulars. The file STUDDEF.TPU is the compiled version of this unit. The file STUDENTS.DAT is a file of STUDENTREC containing 101 records, the first of which is just a dummy record containing only garbage data.

Here is a listing of STUDDEF.PAS for your reference:

```
UNIT StudDef;
INTERFACE

CONST MaxName  = 40;
      MaxAddr  = 60;

TYPE StudentRec = RECORD
                    Left  : BOOLEAN;          {Has left school}
                    Name  : STRING[MaxName];
                    Addr  : STRING[MaxAddr];
                    Form  : INTEGER;
                  END;


IMPLEMENTATION
END.
```

Using the unit STUDDEF, write a program called STUDENTS containing the following procedures, and test the procedures as you write them:

Procedure OpenFiles;

Which opens, or creates, the file STUDENTS.DAT. The main program should call this procedure first.

Procedure DisplayRecord(S:StudentRec);

Which clears the screen and displays the contents of the student record S, with field names and proper formatting. Write another procedure DisplayAllRecords which displays the whole file, waiting for the user to press a key after each record.

Procedure StudentsInForm(Form : Integer);

This procedure takes a form and displays all records of students in that form (except those who have left). At the end it should display a count of the number of students in the form.

Procedure EndOfYear;

Which promotes all students who have not left to the next form (i.e. increments the FORM field). Students who are in form 5 should not be promoted, but should be marked as LEFT.

Procedure NewBatch;

Which creates a new file of student records called NEWSTUD.DAT into which it copies all records of students who have not left school. Remember that record 0 of the new file should not be used for storing a valid record.

Procedure Statistics;

Which counts the number of students in each form and the number of students who left, and presents the data neatly formatted.

Procedure AddRecord;

Which prompts the user to input details for each field of the record (except LEFT, which is automatically set to FALSE), and appends the record to the students' file.

CHAPTER **14**

14. Recursion

14.1. A Simple Example of Recursion

A **RECURSIVE** procedure or function is one which contains a call to itself. The following function calculates the factorial of a positive number recursively:

```
FUNCTION factorial ( number : INTEGER ) : REAL;
BEGIN
   IF    number = 0 THEN factorial := 1
   ELSE  factorial := number * factorial(number - 1);
END;
```

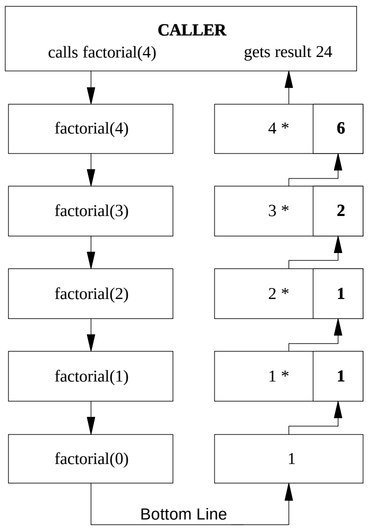
It will be helpful to return to this example after the following discussion.

14.2. Recursive vs. Iterative Strategies

Like many other problems, that of finding the factorial of a number N can be tackled either **ITERATIVELY** or **RECURSIVELY**.

ITERATIVELY, N! is calculated as N * N-1 * ... * 2 * 1.

Thus 4! is 4 * 3 * 2 * 1



RECURSIVELY, N! is calculated as N * (N-1)!.

Thus 4! is

4 * 3!

In solving 4! we must thus first solve 3!, which is

3 * 2!

But now we must solve 2!, which is

2 * 1!

and 1!, which is

1 * 0!

and 0! - but 0! is known to be 1 (by definition), so there is no need to go further. Once we know that 0! = 1, we can solve 1!. Once we have solved 1! we can solve 2!. Once we have solved 2! we can solve 3!, whereupon we can solve 4! - the original problem.

14.3. Characteristics of Recursive Algorithms

The factorial example in the previous section demonstrates the general characteristics of all recursive procedures:

1. The problem is broken down into a SIMPLER FORM OF THE SAME PROBLEM eg 4! → 4 * 3!. The harder problem (4!) is SUSPENDED until the simpler one (3!) can be solved.

2. THERE EXISTS A VERY SIMPLE PROBLEM OF THE SAME TYPE (0!) FOR WHICH THE SOLUTION IS KNOWN (1). This is called the **BOTTOM LINE** of the recursion, and halts infinite recursion (the recursion **BOTTOMS OUT**). The term *A SIMPLER FORM OF THE SAME PROBLEM* used above can thus be defined as a problem which is in some sense CLOSER TO THE BOTTOM LINE. Thus 3! is simpler than 4! because it is closer to 0! (the bottom line).

3. When the bottom line is hit, the procedure goes back up and solves all the 'harder' problems it has suspended on its way down (1!,2!,3! and finally 4!). This is called **UNWINDING THE RECURSION**.

14.4. The Uses of Recursion

The recursive factorial function provides a good demonstration, but is otherwise a useless exercise because the iterative function is faster and simpler. Typical uses of recursion are

- in tree traversal
- in parsing
- in some sorting algorithms
- in graphics (e.g. shape filling)

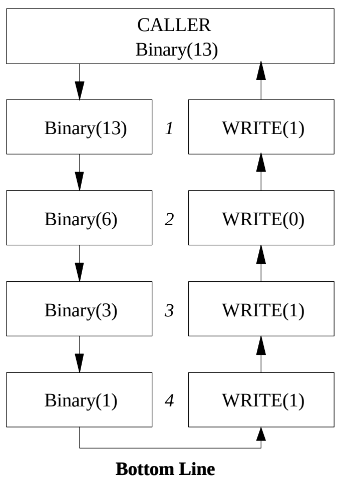
14.5. Examples of Recursion

Study the following recursive routines. For each identify the **BOTTOM LINE**. DRY RUN each routine for two simple cases.

1. This procedure takes a number and prints it out in BINARY (base 2)

```
PROCEDURE Binary (n : INTEGER);
BEGIN
   IF n < 2 THEN WRITE (n)
   ELSE
   BEGIN
      Binary (n DIV 2);
      WRITE  (n MOD 2);
   END;
END;
```

The diagram shows a call to the BINARY procedure with a parameter of 13 (=1101_2).



2. This procedure accepts strings from the console. When the string 'END' is input it prints out all the strings in reverse order. Note that the procedure does not need an array to 'remember' the strings which have been input. Why?

```
PROCEDURE getstring;
VAR s : STRING [20];
BEGIN
    READLN (s);
    IF s <> 'END' THEN getstring;
    WRITELN (s);
END;
```

   Modify the procedure to read in a (short) text file and write it to screen in reverse order WITHOUT USING AN ARRAY. What would be the BOTTOM LINE of this procedure?

3. This procedure lists a text file to screen. When it finds a line starting with a '#' it takes the rest of the line to be the name of another text file. So it stops listing the first file, lists the second, and then resumes listing the first. Of course, the second file may include references to other files within it:

```
PROCEDURE listfile (filename : STRING);
VAR line : STRING;
    flvr : TEXT;
BEGIN
   ASSIGN (flvr,filename);
   RESET (flvr);
   WHILE NOT EOF (flvr) DO
   BEGIN
      READLN (flvr,line);
      IF line[1] = '#' THEN listfile (COPY(line,2,255))
      ELSE WRITELN(line);
   END;
   CLOSE (flvr);
END;
```

14.6. Exercises - Recursion

To better understand how recursive algorithms work, you should single step through the following procedure. Use the call stack display (Debug menu) - hotkey Ctrl-F3 - to inspect the pending calls on the stack.

1. Write a procedure similar to BINARY which converts an integer to OCTAL (base 8).
2. Write a procedure CONVERT(number, base : INTEGER); which converts NUMBER to base BASE (between 2 and 10).
3. Extend the procedure CONVERT to cater for bases up to 16.
4. Ackerman's function A(m,n) is defined as

```
A(m,n)  ::=   if m=0 then          result is n+1
              else if n=0 then     result is A(m-1,1)
              else                 result is A(m-1, A(m,n-1))
```

   Identify the BOTTOM LINE in this function and DRY RUN it for A(1,2) - the result should be 4. Implement it in PASCAL and check it using the following TEST CASES:

A(3,2) = 29.	A(3,4) = 125.	A(3,6) = 509.
A(2,21) = 45.	A(0,0) = 1.	A(4,0) = 13.

NOTE the recursive solution of Ackerman's function for m > 4 takes a long time and is likely to run out of memory long before a solution is found. These are common problems of recursive algorithms.

5. Euclid's algorithm for finding the highest common factor of two integers is defined as

```
HCF(m,n) ::=    if n > m  then          result is HCF(n,m)
                else if n = 0  then      result is m
                else                    result is HCF(n,m MOD n)
```

Identify the BOTTOM LINE of this algorithm. Implement it in Pascal and test it using the following test cases:

```
HCF(128,240) = 16.          HCF(217,93)   = 31.
HCF(559,344) = 43.          HCF(1695,904) = 113.
```