Learning PASCAL

By: M.F.Somers, 2000.

Getting started…

Starting the Borland Turbo Pascal 7.00 IDE…

Before you can start with the basic concepts of programming in PASCAL, you should install a PASCAL compiler. The compiler most suitable for these lessons is the Borland Turbo Pascal 7.00 compiler.

Three versions of this compiler are available. The best one to use is the interactive TPX.exe. This version has all the interactive possibilities the TURBO.exe compiler has, but the IDE (Integrated Development Environment) is slightly better. Though this is not the main reason for using this version, it certainly makes a difference.

Besides the two IDE compilers, the Borland package also contains a command line compiler. We won't use this one because the advantages of the IDE version will be enormous when trying to learn programming or PASCAL. It offers simple menus with on-line help-pages of everything.

These on-line help-pages will become very important for you throughout the complete course because lot's of references are made to them. The idea is that you'll learn to program in PASCAL through practise and examples interactively and not just by reading a book but in a guided way through these help-pages. This will probably be an easier learning curve addressing the important things first…

So, to get started, make a copy of the 'TurboP' directory to some hard-disk. As long as the copied directory is located at the root level of the disk, things have already been set up for you.

After having copied the compiler, copy the example files wherever you want on the same disk. Go to the example directory and use the 'Turbo Pascal' short-cut to fire-up the compiler and it's IDE… Now you can start with the first example…

After a while you might have a look at the extra examples Borland has to offer you. You can find them in the TurboP directory.

Good luck!

The first example…

Compiling, running, output and help…

```
{ This is going to be your first PASCAL program... So let's make things easy
  for starters. First of all you probably have figured out how to open the
  Example1.pas file... You probably did this by having a look under the 'File'
  menu and tried the 'Open' option. Good for you... That's the right way! Now
  for something more sophisticated... Have a look at the 'Compile' menu...

  The first thing you'll see is the 'Compile' option. Try this one...
  You should get a simple window telling you the file compiled okay and
  as you might thing... Yep, you just compiled your first PASCAL program...

  Of course you can have a look in the current directory and find an .EXE
  file with the name Example1 has been created. This is the executable code
  the compiler made of this simple first program. Surely, it is clear that
  the computer can only execute machine instructions and not PASCAL statements.
  This is where the compiler comes in... It translates the PASCAL program into
  instructions the CPU can understand and run your code at full CPU speed
  instead of 'interpreting' the program like BASIC does...

  Now you know what compiling is about and you are now able of compiling a
  given PASCAL program. What about running it then? Of course, try the 'Run'
  menu... Have a look at this menu... Several options are available... But for
  now, just use the 'Run' option. The other stuff will be explained later on...
  or you'll find out yourself.

  Having pressed Run you probably saw your screen flicker... What the @#$!@$
  happened ??? Well, you just ran the first example... How about that then...
  Nothing on the screen, no input asked whatsoever... H'mmm, any ideas???
  Have a look at the 'Debug' menu... Try the 'User Screen' thing... or press
  CTRL-F5.

  Aha, there's the output... Although just a simple message, you just learnt to
  open a file, compile and run it and also inspect the output!

  Having dealt with these basic tasks first, let's start with the real thing...
  PASCAL, that's what this is about... Well, have a look below... The code...
}
```

```
Program Example1;
 Begin
      Writeln( 'This is the first example...' );
 End. { Program }
```

```
{ That was the complete program. All the grey stuff between the parentheses {}
  were comments. The compiler just ignored it but hey, it explained you,
  the programmer what to do... This will become more important later on.
  So get used to putting in plenty of remark statements in your sources!

  Let's analyse the little program. First of all, the program has a name.
  Guess what the name is... Yep right, 'Example1'... Easy enough...
  But then, wow, what's that ?!?!? Well, a program has to start somewhere,
  doesn't it ? Of course the program starts with the begin statement.

  Because a program needs to end as well, the End statement is present at the
  end. Easy enough I should think.
```

```
  You might have noticed, I use indents to 'beautify' sources. This will
  become more and more important when things get bigger. So again, get used
  to doing this as well!

  You also might have noticed the extra comment at the end of the program:
  { Program }. Why is this one needed you might ask... Again, when things get
  nastier there will be quite a lot of 'End' statements in your code... These
  extra comments will help you to figure out which Begin and End belong
  together... Not only the comments but also the indenting will help with this
  in future...

  Oh, I nearly forgot, the dot at the end of the 'End.' statement is VERY
  important. You'll learn in the next few examples that statements like
  'End;' also exist... They are quite different! The one with the dot tells the
  compiler it can stop and ignore the rest of the file.

  EXERCISES:

  1) Make the program print out your name and try to find out by using the help
     what Writeln does... Do this by putting the cursor on the Writeln statement
     and pressing CTRL-F1... Also find out what the difference between Write and
     Writeln is...
  2) Browse through the 'File' and 'Window' menus of the IDE. Try to do some
     simple tasks like opening, saving and creating multiple files. Switch
     between windows and try to rearrange them on screen or resize them.

}
```

The second example…

Variables, basic arithmetic. input, If and For statements…

```
{ The second example.

  In this example you will learn how to get input, use simple variables and do
  basic arithmetic with them. Have a look at the code below first...
}
```

```
Program Example2;

Var
   TheNameEntered   : String;
   TheNumberEntered : Integer;
   ACounterToUse    : Integer;

 Begin
      Write( 'Enter your name: ' );                 { get input }
      Readln( TheNameEntered );
      Write( 'Enter a number between 1 and 20: ' );
      Readln( TheNumberEntered );
                                                    { verify the input }
      If( ( TheNumberEntered < 1 ) Or ( TheNumberEntered > 20 ) ) Then
       Begin
            Writeln( 'The number is not between 1 and 20 ! Program stopped !' );
            Exit;
       End; { If Then }
                                                    { start doing the stuff }
      For ACounterToUse := 1 To TheNumberEntered Do
       Begin
            Writeln( 'Counter: ', ACounterToUse, ' Name: ', TheNameEntered );
       End; { For Do }

      Writeln( 'Program is done...' );              { and we're done... }
 End. { Program }
```

```
{ Well, this program is surely more complicated than the previous one.
  Let's analyse it step by step:

  First the program statement again. This shouldn't be a problem.
  Then a new statement appears. The 'Var' keyword. This keyword is used
  to declare variables of different types. Just put the cursor on the Var and
  press CTRL-F1... You'll see all sorts of possible types. Try to figure out
  what each type means and what kind of values they can contain.

  Question: What are boolean, extended and word variables?

  Okay, now you know how to declare variables. Now you could start using them.
  This is done in the rest of the code. As you might have noticed I used pretty
  obvious names so the meanings are clear. I recommend you to do the same when
  programming. This will make life more easier for you and others in future.
  Some hints: Use capitals and clear names and have a look in the help what
  valid 'identifiers' (variable names) are.

  Using the variables. As you might have noticed, variables are just storage
  places for different kinds of data. These storage places, the memory
  associated with them, do not have a defined contents when a program starts.
  The program should either assign values to them before use or store the
  users input in them. This is done in this example by the Read and Readln
  statements. Try to find the help-pages of these Read(ln) statements.

  The other new things in this example is the use of the 'If' statement.
  Again, go to the statement and get the help-page of it... Also try to find
  the help about the 'relational operators', the things like <= and <>. What
```

will the results be for different types of variables being compared? What types can you compare with these operators?

EXERCISES:

1) Try to modify the 'If' statement in such a way it contains an 'Else' part that if the input has been validated it prints this on screen.

As you might have noticed, the 'If' statement and the next 'For Do' loop also use the keywords 'Begin' and 'End'. You can imagine the number of these statements will become huge if the program gets bigger and bigger. Again this emphasises the use of indents and comments for each 'Begin' and corresponding 'End'!

Next to the 'For Do' loop. At first glance it appears simple. Well, go to the 'For' statement and get the help. Okay, might have studied the examples given on the help-pages and you should now know how to let the counter count down.

2) Modify the program in such a way it will count down. If the counter value is even print an 'even' string and if it is odd an 'odd' string. You might use the 'Mod' keyword for this. This operator calculates the modulo of two things: A Mod B.

After doing all this you probably also noticed the := operator in the program. This is the assignment operator. It enables you to assign a value to a variable. It is also used in the 'For Do' loop!

A question: Is it possible to use a 'While' statement instead of the For loop construction? If so then try this. Also take a look at the 'Repeat' statement. Try to figure out the difference between the 'While' and 'Repeat' constructs. Of course look at the corresponding help-pages.

3) Modify the program in such a way that if the user just pressed enter instead of entering a name, a default value string of 'Your name here' will be used.

4) Try to write a simple program that will solve the quadratic equation: A*X^2+B*X+C=0. The program should accept the A,B and C values with the types of 'Real' and give the two X values by using the well-known ABC formula. Extend the program in such a way it will print out where the maximum or minimum value is like 'Max(X=....)=....' or 'Min(X=....)=....' with the correct values placed on the positions of the dots. You might use the 'Sqrt' function for this. Look at the help-pages to get the information of this 'Sqrt' function.

   Try to make the program 'neat' and produce clear output to the user if wrong input is given and hints on what kind of input is expected. Also make use of comments and indents throughout the complete code...

   Try the following input after your program is finished and watch carefully what your program does:

   A=1, B=2, C=3
   A=0, B=3, C=8
   A=-1, B=0, C=7

```
    If your program aborts with some 'run-time' error message, press F1 to
    figure out what crashes your program and then fix it.

}
```

The third example…

Top-Down model, procedures, functions, units, constants, arrays, global or local variables and pass-by-references…

```
{ The third example... Introducing procedures and functions...

 In this example small sub programs are introduced. These little
 programs within the complete program are called subroutines, procedures
 or functions... Also units will be introduced in this example.

 Have look below at the code which implements two different methods to
 calculate the faculty of a number...
}
```

```
Program Example3;

Uses DOS;                    { use DOS unit for timing... }

Const TestTimes = 100000;

Var LoopVar : Byte;          { global data }
    TimedRecursion : Real;
    TimedLoop      : Real;
    TemporaryTime  : Array[ 1 .. 4 ] Of Word;

{ - - - - - - - - - - - - - ALL FUNCTIONS - - - - - - - - - - - }
{ Calculates the faculty of Nr through a loop method...         }

Function FacultyLoop( Nr : Byte ) : Real;
Var LoopVar : Byte;
    Faculty : Real;
 Begin
     Faculty := 1.0;                    { calculate the faculty Nr }
     For LoopVar := 1 To Nr Do          { by using the loop }
      Faculty := Faculty * LoopVar;     { notice if Nr < 1 the loop does nothing }

     FacultyLoop := Faculty;            { return the result }
 End; { Function }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Calculates the faculty of Nr through recursion...             }

Function FacultyRecursion( Nr : Byte ) : Real;
 Begin
       If( Nr < 1 ) Then
        FacultyRecursion := 1.0          { if Nr is 1 or 0 then result is 1 }
       Else                              { else result is Nr * (Nr-1)! }
        FacultyRecursion := FacultyRecursion( Nr - 1 ) * Nr;
 End; { Function }

{ - - - - - - - - - - - - - - ALL PROCEDURES - - - - - - - - - - - }
{ Prints a result...                                              }

Procedure PrintResult( Nr : Byte; Result : Real );
 Begin
      Write( 'Faculty of ', Nr, ' is : ', Result:9:0 );
 End; { Procedure }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Prints a given timer...                                         }

Procedure PrintTimer( TheTimer : Real );
 Begin
      Writeln( ' TIME: ', TheTimer:4:5 );
 End; { Procedure }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Starts a given timer...                                         }

Procedure StartTimer( Var TheTimer : Real );
 Begin
      GetTime( TemporaryTime[ 1 ], TemporaryTime[ 2 ],   { store the current time  globaly }
               TemporaryTime[ 3 ], TemporaryTime[ 4 ] );
      TheTimer := 0.0;                                   { set specif. timer to 0 }
 End; { Procedure }
```

```
{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Stops a given timer...                                                  }

Procedure StopTimer( Var TheTimer : Real );
Var Hours,
    Minutes,
    Seconds,
    Hundr    : Word;
    TempTime : Real;
 Begin
      GetTime( Hours, Minutes, Seconds, Hundr );      { get current time }
      TempTime := ( Hours - TemporaryTime[ 1 ] );     { and calculate time dif. from global value in hundr.
seconds }
      TempTime := ( Minutes - TemporaryTime[ 2 ] ) + 60 * TempTime;
      TempTime := ( Seconds - TemporaryTime[ 3 ] ) + 60 * TempTime;
      TheTimer := TheTimer + 100 * TempTime + Hundr - TemporaryTime[ 4 ]; { return through pass-by-ref. }
 End; { Procedure }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Starts a loop test and a recusion test...                               }

Procedure StartFacultyTest( Nr: Byte );
Var NrOfTimes : LongInt;
    Result    : Real;
 Begin
      Write( 'RECURSION => ' );               { do the recursion test }
      StartTimer( TimedRecursion );
      For NrOfTimes := 1 To TestTimes Do
        Result := FacultyRecursion( Nr );
      StopTimer( TimedRecursion );
      PrintResult( Nr, Result );
      PrintTimer( TimedRecursion );

      Write( 'LOOP => ' );                     { now do the loop test }
      StartTimer( TimedLoop );
      For NrOfTimes := 1 To TestTimes Do
        Result := FacultyLoop( Nr );
      StopTimer( TimedLoop );
      PrintResult( Nr, Result );
      PrintTimer( TimedLoop );
 End; { Procedure }

{ - - - - - - - - - - - - - MAIN PROGRAM - - - - - - - - - - - - - }

 Begin
      For LoopVar := 1 To 15 Do            { test for different values }
        StartFacultyTest( LoopVar );
 End. { Program }
```

{ Well, in this example lot's of things are demonstrated at once.

First of all, two general programming techniques are shown. The Top-Down model and the Down-Top model. Let's analyse the code with the Top-Down model.

Have a look at the main program. It looks simple enough, just a loop going from 1 to 15 and for each value a test is being done. In a way this is the whole concept of the program. But of course, you should still know what kind of tests are being done... To understand this, you go down a level in the program. Look at the StartFacultyTest procedure.

In the StartFacultyTest procedure, two variables are used of different types. These variables are only present in the Procedure and not the complete program. They are called 'local'. The variables defined at the beginning of the code, TimedRecursion etc. are active throughout the complete program and can be used by any function or procedure. These are called 'global'.

Why on earth not make all variables 'global'? Well, think about the FacultyLoop function. In this function another loop variable is used. If you made all vars global, they should all have different names for the program to work. Imagine having lots of procedures with loops... You see the problem?

For every loop variable you need another name... Makes things complicated huh?

Besides the extra typing and complexity you introduce by making everything global, you cannot use the Top-Down or Down-Top models of programming anymore. You are forced to study the code completely at once and not in parts...

But because I use local variables, we can use the Top-Down model and so let's continue using it... If you take a closer look at the StartFacultyTest procedure, you'll notice the big problem is being reduced to simpler ones by going down to even lower levels... A test is actually a test for the recursion method and the loop method consecutive. Also the use of timings is done with the Top-Down model. Big and complex tasks are broken down to simpler tasks all the way down to the lowest level!

For the StartFacultyTest procedure it just needs to start, stop or print timers and results... Go another level deeper to the StartTimer procedure.

Go to the Procedure keyword of this routine and press CTRL-F1... You see, using procedures and parameters is actually very simple. It makes your code 'neat' and most of the times smaller! You can use the StartTimer procedure multiple times at different places in your program without doing all the work over again each time.

A very important thing with functions or procedures is the 'pass-by-reference' or the 'pass-by-value' models. PASCAL uses 'pass-by-value' by default.

Any parameter used in the call to a function or procedure will only pass the value of the variable to the subroutine. If the subroutine alters the contents of the parameter being passed, the original value of the variable will not change because a local copy is actually created. So 'pass-by-value' means that parameters are local variables and can be modified without any trouble...

What is 'pass-by-reference' then? Well, have a look at the StartTimer procedure. The parameter list contains an extra 'Var' statement. This keyword at that place tells PASCAL to use pass-by-reference for that parameter. This obviously means that if the subroutine alters the contents of the passed thingy, the value of the variable a level higher (at the level of the caller) will change as well! Sometimes you need to use these pass-by-references as is demonstrated in the StartTimer routine. It needs to 'reset' the timer to zero before things will work.

In the same routine you probably noticed the use of the global variable called 'TemporaryTime'. This global variable is actually an Array of reals. An array is just a list of the same type of variables you can access with an index. It is also possible to declare arrays of multiple dimensions. Try to find the help-pages of the Array keyword and study them carefully... You'll use arrays in lots of ways in the future...

EXERCISE:

1) Try to write a small program that calculates the determinant of a given three by three matrix. Try to use a two dimensional array of reals for this. Try to put the calculation of such a determinant into a function. Also Write a string on the screen telling the user a given matrix is 'singular' or not. If the determinant is zero, the matrix is singular.

The next thing you should have noticed in the StarTimer routine is the use of

the GetTime procedure. Skimming through the code you probably won't find a routine called GetTime. Where does it come from? Well, this specific routine is implemented in an 'unit'. The actual name of the unit that contains GetTime is called 'DOS'. Units are just collections of variables, constants and subroutines a programmer will use frequently in several programs. In order to let PASCAL know whether or not to 'use' a unit... Well you can guess, you must use the 'Uses' keyword. Look at the very start of the program... Go to the keyword and get the help-pages...

As you might have read from the help-pages the compiler automatically loads a few units for you. One of these is called the 'system' unit. It contains all the basic routines needed for a PASCAL program. Try to find out what routines and variables the system unit implements for you... Quite a lot huh? Imagine you doing all that work every time you start a new program?

Back a level higher, the StartFacultyTest procedure. The next thing that happens is a loop over the FacultyRecursion function. This function just calculates the faculty of the parameter given through a recursion method. But if you look carefully you'll notice something new in the for loop. Instead of just filling in the loop ranges I used a constant for the maximum value. Why did I do this? Well, imagine a program with lots of loops over the same array. Putting in the array maximum index in each loop will work but what happens when the array is made bigger or even worse smaller? You should adapt all the loops to the new ranges. Quite a lot of work then... You can avoid this by defining a constant for the ranges and use the same constant to declare the array variable itself. This way you only have to modify the constant and the whole code works with the other array size...

Well that's the theory... How does it work in practice? Have a look at the top of the code. You'll see the 'Const' keyword. Get the help-page of it and study it... Having dealt with that constant trick, the code becomes more general, better to maintain in future and last but not least even more readable...

Let's continue with the rest of the procedure... Well nothing really new there... Some extra procedures are being called to stop the timers and print out results. Have a look at each of these routines and make sure you understand them. Pay extra attention to the Write(ln) formats used in this example... Get the help-pages again and notice they speak about ‘field specifiers’...

Having dealt with most of the new PASCAL stuff with the Top-Down approach, let's pay our attention to the functions FacultyLoop and FacultyRecursion. The difference between Functions and Procedures is the way you use them. Functions return a value and can be used in expressions like:

```
If( FacultyLoop( 10 ) > 100 ) Then
 Begin
      ....
 End;
```

Procedures can't be used like this. Although they can give results back by using pass-by-references, they cannot be used like in the above expression. Remember even a function can use pass-by-reference and return more values than just the function value used in an expression. A function returns the result it was assigned with the := operator in the function body. Have a look at one of the functions and look for the :=.

```
  Okay, now what about the two algorithms. They are quite different. The
  recursive method looks really simple... The faculty of N is N times the
  faculty of N-1. That's the trick being used. The calculation of (N-1)! ? Well,
  you could use the FacultyRecursion routine again. Doing this the whole problem
  will reduce at the end to the calculation of 1! which is easy and just 1.

  Recursion is a trick often used to make algorithms simple and robust. But one
  always needs to keep in mind the recursion rule should end with the simplest
  problem being solved directly... If this is not done, the whole program will
  end up getting stuck. 'It hangs' is what's being said.

  The other trick of calculating faculties is using a loop. N! is 1*2*3*...*N so
  a loop going from 1 to N can be used to calculate the products... Easy huh?
  Though the code looks a bit more complicated then the recursion method, for
  larger N this method is actually faster as you might have noticed when running
  the program. Why is this?

  Well, in the loop method, each calculation will just do a single call to the
  FacultyLoop function and will immediately result in an answer. With the
  FacultyRecursion method, each N value being calculated will result in N calls.
  So the number of calls when calculating M factorials will become N*M instead
  of just M. This takes time and becomes noticeable with larger M and N's.

  The lesson being learnt here is that the simplest most easiest algorithm is
  not always the fastest... When programming fast code, one should always keep
  this in mind!

  EXERCISES:

  2) Try to figure out the workings of the program using the Down-Top model.
  3) Make two functions that calculate x^n+x^(n-1)+...1 for a given x and n
     using a loop and a recursion method. Of course n is an integer value.
     Test the timings of these routines. You might want to use the 'Pow'
     function present in the system unit for this. Although, try to make the
     functions NOT using this 'Pow' function because it is relatively slow and
     can be avoided by accumulative multiplication's of X with itself...
     Think things through first and then try to find recursion relations or
     loops tricks...
  4) Try to adapt the program you made for calculating the quadratic equation
     from the previous example in such a way it uses functions and procedures...
     Make sure the complexity of your program becomes less!
  5) Okay, now a simple thing, rewrite all the programs you've written so far in
     such a way that the code becomes 'clean' and readable. Put in remark lines
     per routine and use separator lines like I did. Get used to doing this!
     You'll appreciate it later on... Of course put in plenty of comments...
     That goes without saying!
  6) Study the 'Forward' keyword. Why is it used? What does this mean for the
     ordering of subroutines in your program?

}
```

The fourth example…

Types, file IO, the CRT unit and development…

```
{ The fourth example. Things will get a bit more complex...

  In this example we'll extend the use of variables, introduce records, types,
  arrays of types and of course use units, constants, functions and procedures.

  Take a look at the code first...  It is a simple database program...
  Try to figure out how things work... If you don't understand a keyword
  or something else, use the help-pages... ;-)
}
```

```
Program Example4;

Uses Crt;                          { use the CRT unit for screen output }

Const NoName         = 'No name';  { define some handy constants }
      NoAge          = -1;
      MaxNrOfPersons = 10;

Type ANameType = String[ 80 ];     { define the types for a person }
     AnAgeType = Integer;

     APerson   = Record            { and define the persons record type }
                   TheName : ANameType;
                   TheAge  : AnAgeType;
                   IsEmptyRecord : Boolean;
                 End; { Record }

Var AllThePersons : Array[ 1 .. MaxNrOfPersons ] of APerson;

{ - - - - - - - - ALL FUNCTIONS - - - - - - - - - - - - - - - - }
{ Returns true if record of nr RecNr in database is empty...  }

Function IsRecordEmpty( RecNr : Integer ) : Boolean;
 Begin
      If( ( RecNr <= MaxNrOfPersons ) And ( RecNr > 0 ) ) Then  { if RecNr within range }
       IsRecordEmpty := AllThePersons[ RecNr ].IsEmptyRecord    { get the flag from record }
      Else
       IsRecordEmpty := FALSE;                                  { else default to false }
 End; { Function }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Returns the number of an empty entry in the database...      }

Function GetEmptyRecNr : Integer;
Var Count : Integer;
 Begin
      For Count := 1 To MaxNrOfPersons Do     { loop over all entries }
       If( IsRecordEmpty( Count ) ) Then      { if we found an empty one }
        Begin
             GetEmptyRecNr := Count;          { then stop the search... }
             Exit;
        End; { If Then }

       GetEmptyRecNr := -1;                   { else default to -1 }
 End; { Function }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Returns the number of free entries in the database...        }

Function NrOfFreeEntries : Integer;
Var Count1, Count2 : Integer;
 Begin
      Count1 := 0;                { start counting empty records }

      For Count2 := 1 To MaxNrOfPersons Do
       If( IsRecordEmpty( Count2 ) ) Then
        Inc( Count1 );

      NrOfFreeEntries := Count1;
 End; { Function }

{ - - - - - - - - ALL PROCEDURES - - - - - - - - - - - - - - - - }
```

```
{ Clear all entries in table...                              }

Procedure SetAllEntriesEmpty;
Var Count : Integer;
 Begin
      For Count := 1 To MaxNrOfPersons Do          { loop over all entries and set them emtpy }
       Begin
            AllThePersons[ Count ].IsEmptyRecord := TRUE;
            AllThePersons[ Count ].TheName := NoName;
            AllThePersons[ Count ].TheAge := NoAge;
       End; { For Do }
 End; { Procedure }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Prints the contents of a given person...                   }

Procedure PrintTheRecord( Entry : APerson );
 Begin
      Writeln( 'NAME: ', Entry.TheName, ' and AGE: ', Entry.TheAge );
 End; { Procedure }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Prints out the contents of a given entry...                }
{ This procedure uses the routines available from CRT unit. }

Procedure PrintEntryNr( RecNr : Integer );
Var OldColor : Word;
 Begin
      OldColor := TextAttr;                { get old colors }

      TextColor( YELLOW );                 { set default colors }
      Write( 'REC #: ', RecNr, ' ==> ' );

      If( IsRecordEmpty( RecNr ) ) Then    { if the given rec is empty }
       Begin
            TextColor( RED );
            Writeln( 'EMPTY !!!' );        { then say so... }
       End { If Then }
      Else
       Begin                               { else print out the requested result }
            TextColor( WHITE );
            PrintTheRecord( AllThePersons[ RecNr ] );
       End; { If Then Else }

      TextAttr := OldColor;               { restore the old colors... }
 End; { Procedure }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Start filling in the entries until user stops or table is }
{  filled...                                                 }

Procedure StartEnteringEntries;
Var TheNameEntered : String;
    TheAgeEntered : Integer;
    EmptyRecNr    : Integer;
 Begin
      Writeln( 'Simple database program for ages of persons...' );
      Writeln;
      Writeln( 'Start entering data until database is full or an empty name is given...' );

      While( NrOfFreeEntries > 0 ) Do      { as long as the database is not full... }
       Begin
            Write( 'THE NAME: ' );         { ask for a name }
            Readln( TheNameEntered );
            If( TheNameEntered = '' ) Then  { stop the while do loop if '' entered... }
             Exit;
            Write( 'THE AGE: ' );          { else ask the age }
            Readln( TheAgeEntered );
            If( TheAgeEntered >= 0 ) Then   { and if correct then enter it }
             Begin                         { in the database in an empty spot... }
                  EmptyRecNr := GetEmptyRecNr;
                  AllThePersons[ EmptyRecNr ].TheName := TheNameEntered;
                  AllThePersons[ EmptyRecNr ].TheAge := TheAgeEntered;
                  AllThePersons[ EmptyRecNr ].IsEmptyRecord := FALSE;
             End { If Then }
            Else                           { else give an error message and start again }
             Writeln( 'AGE MUST BE 0 OR BIGGER !!!' );
       End; { While Do }

     If( NrOfFreeEntries = 0 ) Then        { specify why the while do loop stopped }
      Writeln( 'DATABASE IS FULL !!!' )    { either the database is full }
```

```
     Else
      Writeln( 'USER STOPPED WITH INPUT !!!' );   { or the user stopped filling in the data }
 End; { Procedure }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Just dump the complete database to screen...                            }

Procedure DumpTheCompleteDataBase;
Var Count : Integer;
 Begin
      Writeln( 'DUMPING COMPLETE DATABASE ...' );
      For Count := 1 To MaxNrOfPersons Do   { just print all entries }
       PrintEntryNr( Count );
 End; { Procedure }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Dump only filled parts to screen...                                     }

Procedure DumpFilledEntries;
Var Count : Integer;
 Begin
      Writeln( 'DUMPING ONLY FILLED PART OF DATABASE ...' );
      For Count := 1 To MaxNrOfPersons Do      { only print filled entries }
       If( Not IsRecordEmpty( Count ) ) Then
        PrintEntryNr( Count );
 End; { Procedure }

{ - - - - - - - - - START MAIN PROGRAM - - - - - - - - - - - }

 Begin { Main program }
     ClrScr;                  { use CRT again to clear the screen }
     SetAllEntriesEmpty;      { first clear all entries in database }
     StartEnteringEntries;    { start filling the database }
     DumpTheCompleteDataBase; { dump the lot to screen }
     DumpFilledEntries;       { only dump filled entries }
 End.  { Program }
```

{ In this example code two major things are new. First of all, it demonstrates the use of the CRT unit. The second thing, it demonstrates the use of 'Types'.

What new things the Crt unit will give is easy to find out by using the help pages. Try to find out what routines are present and what variables are defined or declared in this unit. Also try to figure out what other units are available in your PASCAL version.

The other new thing, types, is somewhat more interesting. But before we go into the theory of data abstraction, go to the 'Type' keyword and get the help pages. Try to find out what 'ordinal' types are. The other types listed in the help-pages will be the subject in future examples. Let's concentrate on the 'normal' stuff first...

As you might have read, the ordinal types are the types that hold numerical or logical values or perhaps one or more characters. The new type introduced in this example is the 'Record' type.

Have a look at the 'APerson' type definition in the top part of the code. This part of the code defines a new type called 'APerson' which is a combined type containing a name field, an age field and an empty flag field which are of the types string[80], integer and boolean resp. By defining the record type 'APerson' one can use the trick of data abstraction. This means throughout the code one can speak of an APerson type and actually declare arrays or variables of it. Look at the global Var statement. An array of a specified size is declared of the new type.

This means memory will be allocated for this array of the size of the number Of elements times the number of bytes a record holds. Because the array is of the type APerson, using the indexes of the array notation an APerson type

will be the result. To bad you cannot do much with it as such, but one can surely do something with it's fields...

This is done in the procedures and functions of this example. You might have noticed the 'AllThePersons[RecNr].IsEmptyRecord' expression somewhere. What does it mean? Well, of the array AllThePersons, the RecNr entry is addressed. This entry of course is of the type APerson. It is a record. The next thing what happens is that the 'IsEmptyRecord' field will be used. The dot in the expression enables you to specify the record field. So now you can define or declare records and structures of data types and access them... You can also create arrays of them...

The technique of using records and combining different types into a new type is called data abstraction! You can regard the new types as 'things' without making to much trouble of what actually is in it in higher levels of the code when using the Top-Down point-of-view...

EXERCISE:

1) To demonstrate this, alter the program in such a way that not only the age of APerson can be used but i.e. a telephone number as well... You might use a String[15] field for this because a telephone number not only contains digits but separators and other stuff as well.

Have a look at the PrintTheRecord procedure. You see that user defined types can also be used in parameter list of subroutines. Only restriction exists with functions. Functions cannot return a record based type as a result. This is obvious when considering the following piece of code:

```
If( FindFirstRecord >= LastRecord ) Then
 Begin
     ....
 End;
```

With FindFirstRecord and LastRecord being two functions returning a similar record type. What is the meaning of the = operator then? Should it compare the contents or the memory locations of both records? And even if it is clear it should check the contents, how is this done? Which field needs to be compared to what and in what order? The relational operators are therefore only defined for ordinal types!

A simple trick to circumvent this problem is to make a function returning a boolean and accepting both things to compare with each other in a way that is suitable for the type used.

EXERCISE:

2) Try to make a function called 'ArePersonsTheSame' which compares two APerson records and return TRUE if they have the same names. Ages do not matter in this function! Use this function to loop over the database when entering new data to check whether the user does not enter the data of the same person twice in the table.

You can see, even in this example, the use of functions, constants, procedures and a 'neat' programming style helps you to solve more complex tasks...

```
A BIT BIGGER EXERCISE: Development...

Extend the program with a 'DeletePerson' procedure that 'empties' the entry of
a person with a given name. This means of course you can use the search trick
from the previous exercise. If you did things clever enough you already made a
'SearchPerson' function returning the array index of a person or -1 if not
found kind-a-like function which uses the 'ArePersonsTheSame' function in it's
turn to do the comparisons. If not, do this then... It will make things
easier. With this you'll learn to think ahead and program in such a way that
big problems are broken down into smaller problems and more easily to
maintain. The Top-Down or Down-Top method, dependent on whether you start with
the bigger complex tasks or the simple functions first!

Personally I use a mixture you might call 'Down-Top-Down' model. Start with
the very first things like defining data types and 'easy' work routines for
the types. Then start with the big problem and reduce it to smaller things and
eventually end-up using the 'work routines' already made again. Try this
technique of solving problems as well!

After having extended the program with this 'DeletePerson' thing, build a
procedure that gives the user a simple menu with choices like 'enter data',
'dump data', 'delete person' etc. You might actually use the 'Case' statement
for this which I find very suitable. Try to use colours in the menu as well.

Having done this, build a 'ComparePerson(Person1,Person2)' function which
returns -1 if  the name of Person1 is lower in order than the name of Person2,
0 if the names are the same and +1 if Person2 name is lower in order than the
name of Person1. This function is a sort of combination of the >, <, <>, and
= operators. Perhaps you where smart and you already made the
'ArePersonsTheSame' function like this. Thinking ahead again... Good for you!

Now you have a more general comparison function, you can start sorting the
array of persons based on the names of the persons. Try to think of a sorting
algorithm and implement it in a procedure called 'SortTheDatabase'. Also
extend your menu routine with this new  option. Things are going to look more
complete now aren't they ??

The last thing of course, just to finish of the program, is write a
'SaveDataToFile' procedure that writes the complete database into a typed
file. Look in the help-pages for examples on how to use files in PASCAL... You
might look for the words 'Assign', 'Write', 'Close', 'Reset', 'ReWrite' etc.
Also look for the word 'File'. Study the examples given there very well!
You'll need the knowledge later on.

Also make a routine called 'ReadDataFromFile' to read in the data from a given
filename. Extend the menu of course... Now the program is more or less
complete, start playing around with it and try to find any bugs... Fix them if
you find any! Believe me, you probably made some bugs but not to worry, this
happens all the time and is part of development!

You actually learnt quite a lot of programming in PASCAL and in general now.
The examples became more and more complex, you also learnt to extend your
knowledge yourself by using the help-pages.

The following examples will be documented less than usual when PASCAL's
concerned and concentrate on general programming techniques, debugging,
algorithms etc. }
```

The fifth example…

Pointers, heap usage and memory leaks…

```
{ The fifth example...

  In the previous example you actually made a full-blown program. Although
  the program works, still some things could be improved. One of the things
  is the problem with the table sizes. The program makes use of a fixed table
  size (database). What if the user wants to enter more than the default 10
  elements? Things go wrong... Seriously wrong! A simple trick is to increase
  the corresponding constant to a size that's surely enough. That'll work but
  the table will consume quite a lot of memory even if it isn't used completely.

  The memory usage might become so big that things don't fit anymore. The
  'neatest' way of dealing with situations like this is to have a program that
  uses just the right amount of memory it needs at a given point. But this means
  that the program should be able of extending or shrinking the arrays or memory
  usage at 'run-time'. Using static global data tables means fixed sizes
  defined at compile-time.

  So another programming technique should be used and explored for this. The
  technique I'm talking about is called 'dynamical memory usage'. Like the name
  says, things can grow or shrink in size as the program runs. Dynamical, so to
  speak. That's a good idea but how are we going to implement this? For this one
  needs a new data type called 'pointers' and that's what this example is about.

  Not only pointers will be addressed but also the concept of 'heap' usage. The
  heap is the rest of the memory the computer has that is not filled with code,
  stack or static data. As the program runs, heap memory will be 'allocated' and
  'freed' when needed. Dynamical stuff thus... But we do not know yet where on
  the computer a block of memory is free or how to make the program work with
  blocks of memory that can change shape or location. The answer for this is
  pointers again... Pointers and heap usage are linked together like bread and
  butter!

  Lets look at the code first...
}
```

```
Program Example5;

Type ADataType = Record                      { just an example data type }
                     DataField1 : Integer;
                     DataField2 : Real;
                     DataField3 : String[ 30 ];
                End; { Record }

     PADataType = ^ADataType;                   { make new type which is a pointer to the thing... }

Var  PointerOne,                                { global variables used }
     PointerTwo,
     PointerThree : PADataType;

     StaticDataOne,
     StaticDataTwo : ADataType;

     PointerToInteger : ^Integer;
     PointerToReal    : ^Real;
     PointerToString  : ^String;

{ - - - - - - - - - - - PROCEDURES HERE - - - - - - - - - - }
{ Just to print the given datatype...                       }

Procedure PrintDataType( TheData : ADataType );
 Begin
```

```
      Writeln( ' Field1: ', TheData.DataField1,
               ' Field2: ', TheData.DataField2,
               ' Field3: ', TheData.DataField3 );
 End; { Procedure }

{ - - - - - - - - - - - - MAIN PROGRAM - - - - - - - - - - - - }

 Begin
       StaticDataOne.DataField1 := 10;            { just init the static record with some data }
       StaticDataTwo.DataField1 := 20;
       StaticDataOne.DataField2 := 0.36827;
       StaticDataTwo.DataField2 := 0.2310;
       StaticDataOne.DataField3 := 'one';
       StaticDataTwo.DataField3 := 'two';

       PrintDataType( StaticDataOne );            { dump the results }
       PrintDataType( StaticDataTwo );

       PointerOne := Addr( StaticDataOne );       { setup the pointers to the data recs }
       PointerTwo := Addr( StaticDataTwo );       { each pointer now contains the memory locations of the data
types }

       PointerThree := PointerOne;                { set 3rd pointer to first datatype }
       PointerThree^.DataField1 := 300;

       PrintDataType( StaticDataOne );            { dump the results }
       PrintDataType( StaticDataTwo );

       PointerThree := PointerOne;                { now let it point to 2nd type }
       PointerThree^.DataField2 := 0.25252525252;

       PrintDataType( StaticDataOne );            { dump the results }
       PrintDataType( StaticDataTwo );

       PointerToInteger := Addr( StaticDataOne.DataField1 );   { and setup other pointers }
       PointerToReal    := Addr( StaticDataTwo.DataField2 );
       PointerToString  := Addr( PointerThree^.DataField3 );

       PointerToInteger^ := 8;                   { start altering data }
       PointerToReal^    := 0.1111111;
       PointerToString^  := 'hi hi';

       PrintDataType( StaticDataOne );           { print out some stuff }
       PrintDataType( StaticDataTwo );
       PrintDataType( PointerThree^ );

       PointerToInteger := Nil;                  { and set pointers to nil }
       PointerToReal    := Nil;
       PointerToString  := Nil;
       PointerThree     := Nil;
       PointerTwo       := Nil;
       PointerOne       := Nil;

       { Now show some dynamical stuff }

       New( PointerToInteger );                    { allocate memory for an integer and point pointer to it }
       Writeln( PointerToInteger^ );               { show what piece of mem contains }
       PointerToInteger^ := 10101;                 { alter contents }
       Writeln( PointerToInteger^ );               { show new value then }
       Dispose( PointerToInteger );                { free the memory allocated }
       Writeln( PointerToInteger^ );               { but pointer still points to that spot and contents hasn't
changed }
       New( PointerToReal );                       { allocate memory for a real and point the pointer to it }
       New( PointerToInteger );                    { allocate memory for an integer and point the pointer to it }
       Dispose( PointerToReal );                   { free the memory allocated for the real }
       Writeln( PointerToInteger^ );               { show contents of new integer memory, different because
                                                   {  the memory was at a different location }
       Dispose( PointerToInteger );                { free the memory }
 End. { Program }
```

{ Let's analyse the program in a bit more detail...

  The first few line shouldn't be a problem. Just a new data type is defined.
  The next statement in the Type clause is new. It contains a ^ character.
  PASCAL interprets this as 'a pointer to an xxx'. So the ^ADataType means 'a
  pointer to an ADataType'. Because it is written in the Type clause we have

```
defined a new type that is actually a pointer to a ADataType.

But what the hell are pointers? Well, as you know, a normal variable can
contain some value. An integer can contain numbers, a string characters, user
defined types like records can hold whatever you put in them. What about
pointers to data types then, what do they contain?

Well, a pointer to an ADataType contains the memory location of such an
ADataType. It does not contain the data itself, but where in memory an
ADataType is located. It's an address. A location in memory. That's quite
different! Make sure you understand the difference!

Because pointers only contain the location of a given type, it can point to
different instances (variables) of the same type. Just fill the pointer with
the address of the new or other data type. This is done in the lines after the
first two calls to the PrintDataType procedure. It uses the Addr() function.
Get the help-pages of this thing... In the examples of these help-pages, the
use of the type 'Pointer' is demonstrated.

As I said before, normally you use a pointer to a specified type. But when you
want to use addresses of things regardless their type, you should use
'Pointer'. Look at the help-pages of this type. A good example is given
there...

Let's continue with the code. The next line will point the 3rd pointer
to the same thing as the first pointer is pointing to. So now one can access
the static data from type 1 through the type itself, the first pointer
pointing to it and the third pointer.

Because pointers just contain memory locations, the compiler needs to know
what to do when your program tries to access stuff. There are two different
situations when working with pointers: Is the program trying to alter the
contents of the pointer itself, when i.e. changing the address it points to,
or the actual data the pointer is pointing to?

To make the difference PASCAL uses the ^ symbol again. As soon as this ^ thing
is present at the end of a pointer identifier, the compiler knows you are
trying to alter the data the pointer is pointing to and not just the pointer
contents (address of the type with some memory associated to it) itself.

To demonstrate the difference, the example uses a few of these statements.
Look at them again and try to understand what's going on. Is the pointer
being changed or the data it points to?

On to the next thing... After a while when using pointers you don't need
them anymore in the program. But as is demonstrated in the example they still
point to memory locations whether they are used, allocated, freed or whatever.
To make sure a pointer doesn't point to a memory spot that does not contain
valid or allocated data, a good programmer assigns the value 'Nil' to the
pointer. This will make the pointer point to nothing at all!

Beware! When a pointer is used but not initialised, it points to a random
memory location. So before use, either assign 'Nil' or a valid address to
it... Go to the help-pages of the 'Nil' keyword and read them.

The basics and some of the strengths of pointers have been demonstrated
now... But in the beginning of this example I was talking about dynamical
```

```
  memory usage. Well, this is demonstrated in the second part of the example.
  Read it carefully and read the help-pages of the 'New' and 'Dispose'
  procedures. Then look at the example again. You can skip the parts about
  'Objects'... Maybe in a latter stage we'll think about them. Not for now...

  The example continues. Because I put in plenty of remarks in this part of the
  code, no problem is expected when analysing this... Do you understand what's
  going on?

  One major thing to remind is the disposure of everything you have ever
  allocated in the program. If a program allocates memory but doesn't free it
  eventually, the program will 'leak memory' and after a while the memory has
  been allocated completely so the program crashes... Make sure you free
  everything you ever allocated after use! Also never let pointers 'dangle'...
  Give them a usable memory address or nil. This enables you to do checks
  whether the thingy points to correct data or crap...

  EXERCISE:

  1) Try to modify the program so it will allocate a chunk of memory for an
     ADataType record dynamically through the use of PointerThree. Fiddle about
     with the data for a while and then free the memory.

  Although things are getting better and better, there's still a thing that
  might go wrong. What happens if the program uses so much memory that it ran
  out of free memory? What happens then if you try to allocate more? Things go
  really wrong so PASCAL has implemented the 'Nil' trick again. If trying to
  allocate memory and after the call to 'New' the pointer contains 'Nil', a
  memory error has occurred. This means no more heap memory is available and if
  the program doesn't use that much memory at a given point in time, a memory
  leak is likely to be present...

  Another trick to check for memory leaks is to print out the amount of free
  memory before you start allocating stuff and after disposing it again. Use the
  MaxAvail and MemAvail functions for this. Find out in what unit they are
  defined...

  EXERCISE:

  2) Implement the memory leak check trick with MemAvail in the program example.
     Also implement the checks for 'Nil' after the call to 'New'.

  Okay, now we know how to allocate and free memory, use pointers to these
  chunks of memory to access the data and safely manage the heap a program has
  access to. Still we haven't solved the database problem discussed in the
  beginning. For this we'll use a 'linked list' and that's going to be the
  subject of the next example.

}
```

The sixth example…

An unit, linked-lists, sets and procedural-types…

```
{ The sixth example...

  In this example we'll try to solve the static database problem with a
  'linked list'. To demonstrate the use of a linked list, look at the
  code first... But because a linked list is a very general thing one
  might use quite a lot, the list will be implemented in an unit. You'll
  notice the only difference between making a program and an unit are a
  few statements at the very start of the code...

}
```

```
Unit LnkLst;

Interface                       { the public part of a unit }

Type    PAListNode = ^AListNode;
        AListNode = Record
                        TheData         : Pointer;
                        TheNextNode,
                        ThePreviousNode : PAlistNode;
                End; { Record }

        LoopThroughAction = ( LoopStop, ContinueLoop, LoopForwards,
                              LoopBackwards, ChangeLoopDirection );
        LoopMessages      = ( LastNodeOfList, FirstNodeOfList,
                              AnyNodeOfList, OnlyNodeOfList );
        SortMethods       = ( SortAscendingForWards, SortAscendingBackWards,
                              SortDescendingForWards, SortDescendingBackWards );

        AllocateFunc    = Function : Pointer;
        DeAllocateProc  = Procedure( DataItem : Pointer );
        LoopThroughFunc = Function( DataItem : Pointer; Message : LoopMessages ) : LoopThroughAction;
        ComparisonFunc  = Function( DataItem1, DataItem2 : Pointer ) : Integer;

Function  AllocateNewNode( TheFunc : AllocateFunc ) : PAListNode;
Function  GetNodesData( TheNode : PAListNode ) : Pointer;
Function  SetNodesData( TheNode : PAListNode; Data : Pointer ) : Pointer;
Function  GetNextNodeInList( TheList : PAListNode ) : PAListNode;
Function  GetPreviousNodeInList( TheList : PAListNode ) : PAListNode;
Function  LoopThroughListOfNodes( TheList : PAListNode; StrtDirection : LoopThroughAction;
                                  TheFunc : LoopThroughFunc ) : PAListNode;
Function  GetFirstNodeOfList( TheList : PAlistNode ) : PAListNode;
Function  GetLastNodeOfList( TheList : PAlistNode ) : PAListNode;
Function  GetNumberOfNodesInList( TheList : PAListNode ) : Integer;
Procedure InsertTheNodeInList( TheNode : PAListNode; Var TheList : PAListNode );
Procedure DeleteTheNodeFromList( TheNode : PAListNode );
Procedure DeAllocateNode( Var TheNode : PAListNode; TheProc : DeAllocateProc );
Procedure DeAllocateCompleteList( Var TheList : PAListNode; TheProc : DeAllocateProc );
Procedure SwapDataOfNodes( Node1, Node2 : PAListNode );
Procedure SortNodesOfList( TheList : PAListNode; Method : SortMethods; TheFunc : ComparisonFunc );

Implementation                  { the private part of the unit }

{ - - - - - - - - - - - - - GLOBAL VARIABLES HERE - - - - - - - - }

Var _A_Nr_Of_Calls_Counter_Ptr : ^Integer;

{ - - - - - - ALL FUNCTIONS HERE - - - - - - - - - - - - - - - - }
{ Allocates a new node and calls user supplied AllocateFunc... }

Function AllocateNewNode( TheFunc : AllocateFunc ) : PAListNode;
Var TheNewNode : PAListNode;
 Begin
      New( TheNewNode );              { allocate memory }
      If( TheNewNode = Nil ) Then
       Begin
           Writeln( '!!! ERROR ALLOCATING NODE !!!' );
           Halt( 1 );
       End; { If Then }

      TheNewNode^.TheData := TheFunc;      { call user supplied func }
```

```
      TheNewNode^.TheNextNode := Nil;
      TheNewNode^.ThePreviousNode := Nil;

      AllocateNewNode := TheNewNode;       { and return new node }
 End; { Function }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Returns a pointer to the data item of a node...                        }

Function GetNodesData( TheNode : PAListNode ) : Pointer;
 Begin
      If( TheNode <> Nil ) Then
       GetNodesData := TheNode^.TheData
      Else
       GetNodesData := Nil;
 End; { Function }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Sets a datapointer in a given node, returns old pointer...  }

Function SetNodesData( TheNode : PAListNode; Data : Pointer ) : Pointer;
 Begin
      SetNodesData := GetNodesData( TheNode );
      If( TheNode <> Nil ) Then
       TheNode^.TheData := Data;
 End; { Function }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Get pointer to next node in list or nil...                             }

Function GetNextNodeInList( TheList : PAListNode ) : PAListNode;
 Begin
      If( TheList <> Nil ) Then
         GetNextNodeInList := TheList^.TheNextNode
      Else
       GetNextNodeInList := Nil;
 End; { Function }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Get pointer to previous node in list or nil...                         }

Function GetPreviousNodeInList( TheList : PAListNode ) : PAListNode;
 Begin
      If( TheList <> Nil ) Then
         GetPreviousNodeInList := TheList^.ThePreviousNode
      Else
       GetPreviousNodeInList := Nil;
 End; { Function }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ A general looping function to run through a linked list...  }
{ The start will be the node given as the list. The direction }
{  will be of Direction and for each element a call to the    }
{  supplied LoopFunc will be made... The return value of this }
{  supplied function will define how the loop continues...    }
{ The result returned of this function will be a pointer to   }
{  the last node being accessed by supplied LoopFunc or Nil.  }

Function LoopThroughListOfNodes( TheList : PAListNode; StrtDirection : LoopThroughAction;
                                 TheFunc : LoopThroughFunc ) : PAListNode;
Var CurrentNode : PAListNode;
    Direction   : LoopThroughAction;
    Message     : LoopMessages;
 Begin

      If( ( TheList = Nil ) Or ( ( StrtDirection <> LoopBackWards ) And
          ( StrtDirection <> LoopForwards ) ) ) Then
       Begin
            LoopThroughListOfNodes := Nil;
            Exit;
       End; { If Then }

      CurrentNode := TheList;   { set start of loop }
      Direction := StrtDirection;

      While( Direction <> LoopStop ) Do
       Begin
            Message := AnyNodeOfList;                { figure out the current position for the message }
                                                     { passed to the user supplied function }
            If( CurrentNode^.TheNextNode = Nil ) Then
             Message := LastNodeOfList;
```

```
            If( CurrentNode^.ThePreviousNode = Nil ) Then
             Message := FirstNodeOfList;
            If( ( CurrentNode^.ThePreviousNode = Nil ) And ( CurrentNode^.TheNextNode = Nil ) ) Then
             Message := OnlyNodeOfList;

            StrtDirection := TheFunc( CurrentNode^.TheData, Message );  { call the user func }

            Case( StrtDirection ) Of                                  { change direction if needed }
                ChangeLoopDirection: If( Direction = LoopForwards ) Then
                                      StrtDirection := LoopBackWards
                                     Else
                                      StrtDirection := LoopForwards;
                ContinueLoop:        StrtDirection := Direction;       { or keep original direction }
            End; { Case }

            Direction := StrtDirection;      { make it the current direction }

            Case( Direction ) Of             { and get the correct new node then }
                LoopForwards:  CurrentNode := GetNextNodeInList( CurrentNode );
                LoopBackWards: CurrentNode := GetPreviousNodeInList( CurrentNode );
            End; { Case }

            If( CurrentNode = Nil ) Then      { if we hit dirt, stop looping }
             Direction := LoopStop;
       End; { While Do }

      LoopThroughListOfNodes := CurrentNode;         { return last accessed node or Nil if hit the dirt }
 End; { Function }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Locates the first node of the list given a 'middle' node...                     }
{ The function starts walking through the list in BackWards direction            }
{  and returns Nil or the last node encountered...                                }

Function _TheUsedLoopFuncForFinds( Data : Pointer; Message : LoopMessages ) : LoopThroughAction; Far;
 Begin
      If( Message <> AnyNodeOfList ) Then           { only stop looping at the start or end of the list }
        _TheUsedLoopFuncForFinds := LoopStop        { because of this, any loop maynot start at the end or }
      Else                                          { beginning or things wil stop at the first call! }
        _TheUsedLoopFuncForFinds := ContinueLoop;

      { also increase counter needed if present }

      If( _A_Nr_Of_Calls_Counter_Ptr <> Nil ) Then
       Inc( _A_Nr_Of_Calls_Counter_Ptr^ );
 End; { Sub Function }

Function GetFirstNodeOfList( TheList : PAlistNode ) : PAListNode;
Var TmpPtr : ^Integer;
 Begin
      GetFirstNodeOfList := TheList;
      If( TheList = Nil ) Then                      { nothing to do }
       Exit;
      If( TheList^.ThePreviousNode = Nil ) Then     { if first is given then return }
       Exit;
      TheList := TheList^.ThePreviousNode;          { start with a 'middle' one }
      TmpPtr := _A_Nr_Of_Calls_Counter_Ptr;         { save old counter pointer }
      _A_Nr_Of_Calls_Counter_Ptr := Nil;            { deactivate counter }
      GetFirstNodeOfList := LoopThroughListOfNodes( TheList, LoopBackWards, _TheUsedLoopFuncForFinds );
      _A_Nr_Of_Calls_Counter_Ptr := TmpPtr          { restore old counter pointer }
 End; { Function }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Locates the last node of the list given a 'middle' node...                      }
{ Same trick is used like the find of the first node but we start                 }
{  looping in the forwards direction now...                                       }

Function GetLastNodeOfList( TheList : PAlistNode ) : PAListNode;
Var TmpPtr : ^Integer;
 Begin
      GetLastNodeOfList := TheList;
      If( TheList = Nil ) Then                      { nothing to do }
       Exit;
      If( TheList^.TheNextNode = Nil ) Then         { if last is given then return }
       Exit;
      TheList := TheList^.TheNextNode;              { start with a 'middle' one ... }
      TmpPtr := _A_Nr_Of_Calls_Counter_Ptr;         { save old counter pointer }
      _A_Nr_Of_Calls_Counter_Ptr := Nil;            { deactivate counter }
       GetLastNodeOfList := LoopThroughListOfNodes( TheList, LoopForWards, _TheUsedLoopFuncForFinds );
      _A_Nr_Of_Calls_Counter_Ptr := TmpPtr;            { deactivate counter }
 End; { Function }
```

```
{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Counts the number of nodes present in a list...                              }

Function GetNumberOfNodesInList( TheList : PAListNode ) : Integer;
Var SavePtr   : ^Integer;
    Counter   : Integer;
 Begin
      If( TheList = Nil ) Then
       Begin
           GetNumberOfNodesInList := 0;
           Exit;
       End; { If Then }
      SavePtr := _A_Nr_Of_Calls_Counter_Ptr;          { save old counter addr }
      _A_Nr_Of_Calls_Counter_Ptr := Addr( Counter );  { set new counter addr }
      TheList := GetFirstNodeOfList( TheList )^.TheNextNode;    { locate beginning of list }
      Counter := 1;                                   { set counter to 1 and count with loopthrough }
      TheList := LoopThroughListOfNodes( TheList, LoopForWards, _TheUsedLoopFuncForFinds );
      _A_Nr_Of_Calls_Counter_Ptr := SavePtr;          { restore old counter addr }
      GetNumberOfNodesInList := Counter;              { return nr counted }
 End; { Function }

{ - - - - - - - - - ALL PROCEDURES HERE - - - - - - - - - - - - - - - - - - - }
{ Inserts a given Node into the linked list location specified...              }
{ The routine will put the new node just infront of the location...            }
{ If a list pointer of Nil is passed through the inserted node will            }
{  become the base of the list...                                              }

Procedure InsertTheNodeInList( TheNode : PAListNode; Var TheList : PAListNode );
 Begin
      If( TheNode = Nil ) Then                    { check if node given is not allready part of list }
       Exit;
      If( TheNode^.TheNextNode <> Nil ) Then
       Exit;
      If( TheNode^.ThePreviousNode <> Nil ) Then
       Exit;

      TheNode^.TheNextNode := TheList;

      If( TheList <> Nil ) Then
       Begin
            TheNode^.ThePreviousNode := TheList^.ThePreviousNode;
            TheList^.ThePreviousNode := TheNode;
            If( TheNode^.ThePreviousNode <> Nil ) Then
             TheNode^.ThePreviousNode^.TheNextNode := TheNode;
       End { If Then }
      Else
       Begin
            TheNode^.ThePreviousNode := Nil;
            TheList := TheNode;
       End; { If Then Else }
 End; { Procedure }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Deletes a given Node from the linked list it is in...                        }
{ The procedure doesnot release the memory used by the data and                }
{  the node itself! Use the DeAllocateNode for this...                         }
{ This routine just breaks the links of pointers in a list...                  }

Procedure DeleteTheNodeFromList( TheNode : PAListNode );
 Begin
      If( TheNode <> Nil ) Then
       Begin
            { relink the list first }
            If( TheNode^.TheNextNode <> Nil ) Then
             TheNode^.TheNextNode^.ThePreviousNode := TheNode^.ThePreviousNode;

            If( TheNode^.ThePreviousNode <> Nil ) Then
             TheNode^.ThePreviousNode^.TheNextNode := TheNode^.TheNextNode;

            { then break links from this node... }
            TheNode^.ThePreviousNode := Nil;
            TheNode^.TheNextNode     := Nil;
       End; { If Then }
 End; { Procedure }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Deallocates a node and calls user supplied proc for the data...              }
{ This procedure also removes the node from the list if it is part             }
{  of a list automatically...                                                  }
```

```
Procedure DeAllocateNode( Var TheNode : PAListNode; TheProc : DeAllocateProc );
Var TmpNode : PAListNode;
 Begin
      If( TheNode <> Nil ) Then
       Begin
            TmpNode := GetNextNodeInList( TheNode );  { save a link to the rest }
            If( TmpNode = Nil ) Then
             TmpNode := GetPreviousNodeInList( TheNode );

            DeleteTheNodeFromList( TheNode );    { first remove the links of this node if it is linked in a list }
            TheProc( TheNode^.TheData ); { call supplied proc to deallocate data }
            Dispose( TheNode );          { and remove memory for the node }
            TheNode := TmpNode;          { and set pointer given to valid value }
       End; { If Then }
 End; { Procedure }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Deallocates the complete list node for node...                        }

Procedure DeAllocateCompleteList( Var TheList : PAListNode; TheProc : DeAllocateProc );
 Begin
       While( TheList <> Nil ) Do
        DeAllocateNode( TheList, TheProc );
 End; { Procedure }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Swaps the data of two nodes...                                        }

Procedure SwapDataOfNodes( Node1, Node2 : PAListNode );
Var TmpPtr : Pointer;
 Begin
      If( ( Node1 <> Nil ) And ( Node2 <> Nil ) ) Then
       Begin
            TmpPtr := Node2^.TheData;
            Node2^.TheData := Node1^.TheData;
            Node1^.TheData := TmpPtr;
       End; { If Then }
 End; { Procedure }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Sorts the list of nodes with a given comparison func...               }
{ The TheFunc comparison function returns -1, 0, or 1 based on the  }
{  comparison of the two data pointers given.                           }
{ The TheFunc returns -1 if Data2 is 'bigger' than Data1, 0 if they }
{  are equally 'big', 1 if Data1 is 'bigger' than Data2.                }
{ The list will be sorted according to the method given:                }
{  SortAscendingForWards: sort list from small to big elements      }
{   going forwards in the list from node given.                         }
{  SortDescendingForWards: sort list from big to small elements     }
{   going forwards in the list from node given.                         }
{  SortAscendingBackWards: sort list from small to big elements     }
{   going backwards in the list from node given.                        }
{  SortDescendingBackWards: sort list from big to small elements    }
{   going backwards in the list from node given.                        }

Procedure SortNodesOfList( TheList : PAListNode; Method : SortMethods; TheFunc : ComparisonFunc );

  Function _FindSwapNodeInList( TheSubList : PAListNode ) : PAListNode;
  Var CurrentNode : PAListNode;
      SwapNodeSoFar : PAListNode;
   Begin
        SwapNodeSoFar := TheSubList;                    { start at given node }

        If( Method In [ SortAscendingForWards, SortDescendingForWards ] ) Then
         CurrentNode := GetNextNodeInList( TheSubList )      { get next node }
        Else
         CurrentNode := GetPreviousNodeInList( TheSubList ); { get previous node }

        While( CurrentNode <> Nil ) Do                      { if nil then done else loop }
         Begin
              If( Method In [ SortAscendingForWards, SortAscendingBackWards ] ) Then
               Begin                                        { and check if current is smaller }
                    If( TheFunc( GetNodesData( CurrentNode ), GetNodesData( SwapNodeSoFar ) ) < 0 ) Then
                     SwapNodeSoFar := CurrentNode;                      { if so then found new smaller }
               End
              Else                                          { or check if current is bigger }
               If( TheFunc( GetNodesData( CurrentNode ), GetNodesData( SwapNodeSoFar ) ) > 0 ) Then
                SwapNodeSoFar := CurrentNode;                       { if so then found new bigger }

              If( Method In [ SortAscendingForWards, SortDescendingForWards ] ) Then
               CurrentNode := GetNextNodeInList( CurrentNode )      { get next node }
```

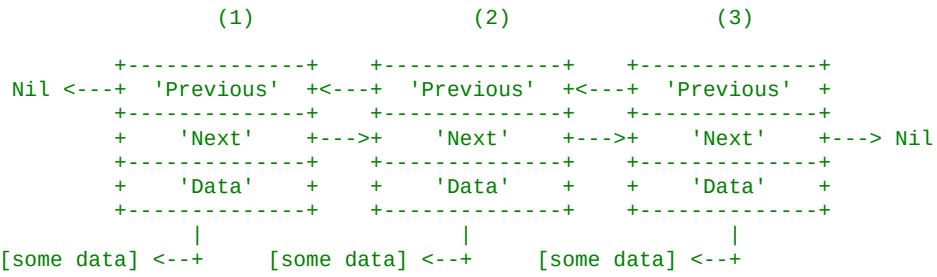
```
              Else
               CurrentNode := GetPreviousNodeInList( CurrentNode ); { get previous node }
          End; { While Do }
         _FindSwapNodeInList := SwapNodeSoFar;          { return best found so far... }
   End; { Sub Function }

 { -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  }

 Begin
      While( TheList <> Nil ) Do
       Begin
            SwapDataOfNodes( TheList, _FindSwapNodeInList( TheList ) );    { swap found with current one }
            If( Method In [ SortAscendingForWards, SortDescendingForWards ] ) Then
             TheList := GetNextNodeInList( TheList )                       { get next node }
            Else
             TheList := GetPreviousNodeInList( TheList );                  { else get previous node }
       End; { While Do }                                                   { list is now sorted... }
 End; { Procedure }

{ - - - - - - UNIT INIT STARTS HERE - - - - - - - - - - - - }

 Begin
      _A_Nr_Of_Calls_Counter_Ptr := Nil;      { init counter pointer to nil }
 End. { Unit }
```

{ As you can see, the difference between a program and an unit is small.
  The only extra statements are the 'Interface', 'Unit' and 'Implementation'
  statements. Surely these statements speak for them selves but never the
  less, look at the corresponding help-pages.

  Okay, now you know how to make an unit. Now the question arises 'why should
  you bother making units?'. To answer this question you should have a closer
  look at this unit. Let's analyse the code then...

  First of all, in this unit a type is defined. The type 'ALinkNode' is a
  record containing three pointers. Two pointers point to the type itself but
  the third pointer is declared with the 'Pointer' statement. Perhaps you recall
  that these pointers are just normal pointer that can point to anything you
  want. That is one of the tricks you'll make use of in more programs. With
  these pointers you can handle anything you like!

  Back to the two other pointers. As the names of the record type suggest these
  will be used to make a linked list of 'ALinkNode' types. But what actually is
  such a linked list? Have a look at the pictogram below:



  In this pictogram three nodes are 'allocated' and placed somewhere in memory.
  For convenience they are numbered 1..3. Each of these nodes has a 'Data'
  pointer pointing to some chunk of memory containing the data you like. That's
  why you should use type-less pointers for these. Now the node can point to any
  data you like.

  The name linked list is also apparent now. All the nodes are linked together
  by use of the other two pointers. By using 'Nil' one can signal the ending
  nodes of a linked list. Using the 'Previous' pointers one can move 'Downwards'

or 'Left' in the list to the previous node. Using the 'Next' pointer... Well, you can guess.

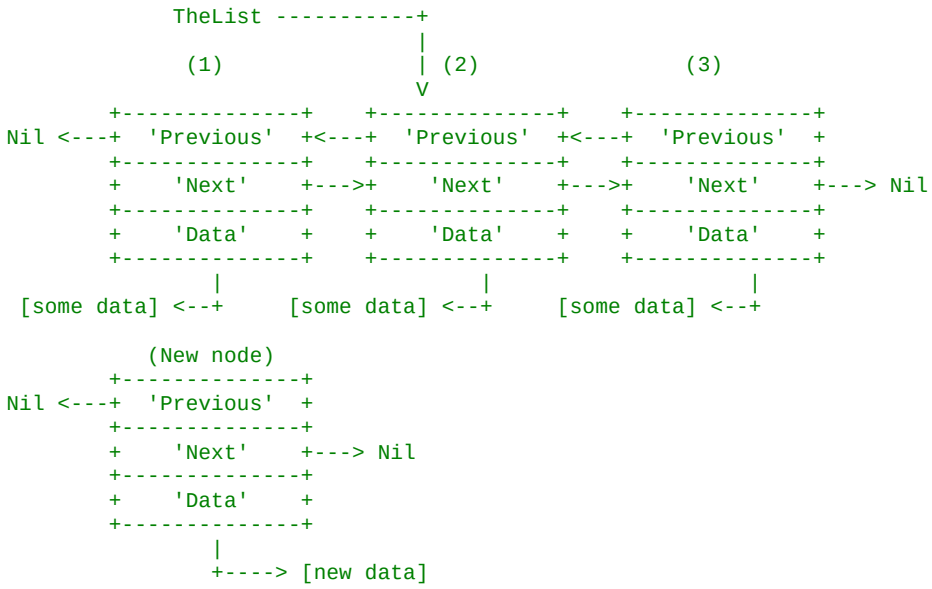
The list is set-up and ready to be used as soon as we know where in memory a node is. The rest we can find through the links. But one must always have at least one pointer in the code pointing to a single node of the list. If this is not the case, the whole list is lost and the memory it uses plus the data is un-findable... Keep this in mind! It is very important!

But let's assume, we are experts in coding linked lists and we have a pointer to the 2nd node. Because we were the experts we called this pointer 'TheList'. With this pointer and the right code, we can get anything from the list.

You now understand how to, 'in principle', walk through a list and we got the pointer to a node of the list. Things would be very shameful if that was all we could do so more must be possible. One of the major advantages of such a list is it's dynamical nature. Surely it should be dynamical because we are trying to solve the static data problem with such a list.

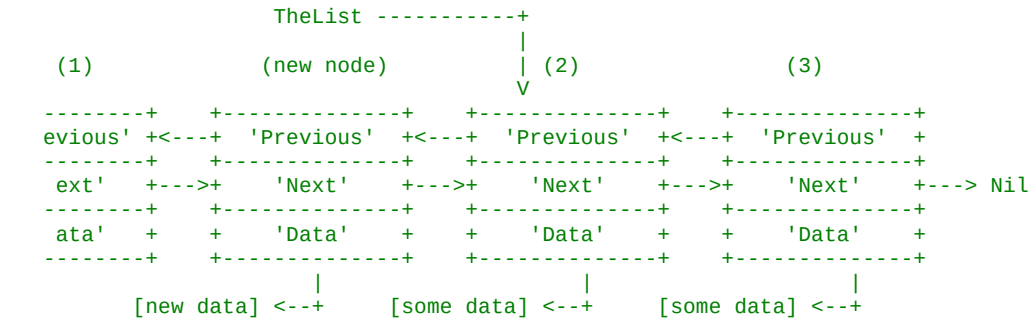
How come it is dynamic then? Well, let's suppose we allocated a new node and set the data pointer correctly to a new data entry or chunk of memory. We can actually extend the list just by manipulating the 'Next' and 'Previous' pointers of a node.

The situation:



What if we fill the 'Previous' and 'Next' pointers of the new node with the contents of the 'Previous' pointer of the TheList node and the address of the TheList node resp. We are half way of inserting the new node in the list... The rest that remains to be done is to actually get the pointers of nodes 1 and 2 (TheList). Point the 'Next' pointer of node 1 to the new node and point the 'Previous' pointer of the TheList node also to the new node.

We now have the situation:



Of course with the 'Previous' pointer of node 1 still 'Nil' to signal the end of the list.

You see, it is possible to 'insert' new nodes. The reverse process is also possible (take a look at the 'DeleteTheNodeFromList' procedure) and the whole list is dynamical. Have a look at the 'InsertTheNodeInList' procedure and make sure the above explained method is being used...

Of course some extra routines handling nodes or lists of nodes would be welcome in an unit. That's why the unit is not just composed of the routines mentioned so far. The rest of the unit is code to run through a list of nodes, sort them, swap the data of two nodes etc. Have a very good look at the LoopThroughList function. By making it so general as I did, I'm able of using it for several things and 're-use' code without actually doing the work more than once...

Besides the idea of a linked list, this unit is also a good example of a general unit. Through the use of typed procedures and functions like 'ComparisonFunc' the unit doesn't care what data the list will hold, the user will supply the correct procedures or functions as parameters when needed. Look up in the help-pages the information of 'procedural-types'. It is a useful tool for a programmer.

Another new thing introduced in this unit is the 'Set' of things... Have a look at the 'SortMethods' type definition. A variable of the type 'SortMethod' can only be one of the four specified items of the set. One can actually make statements like:

```
If( Method In [ ..... ] ) Then
 ....;
```

This is used in the 'LoopThroughList' function and the 'SortNodesOfList' procedure.

QUESTIONS:

1) Why did I implement a pointer to the counter instead of just using an Integer itself? What consequences does this have for the code and what are the advantages? Give an example of a situation explaining why this trick is needed!
2) How would a line of code look like if I wanted to insert a new data item into a list? Try to use the 'AllocateANewNode' function for this.
3) What happens if to press 'Run' now? How come it gives this message?

```
  EXERCISES:

  1) Rename this unit file to 'LnkLst.pas' and compile that file to
     'lnklst.tpu' and do the next example first to get used to the use
     of this unit...
  2) Get your version of the database program you made and put in a
     correct 'Uses' statement so it will use this new unit.
  3) Rewrite your code now so it uses the routines of this unit.
     A problem may arise when loading or writing a database from or
     to a file. The file may contain just one or perhaps 10 records.
     The best option is to use the 'IOResult' method... Look in the
     help-pages for the example of this keyword... Now make the read
     from file routines in such a way it keeps on reading data until
     an IOResult 'error' is encountered... This means the end of the
     file was reached and all data was read... The other trick is to
     determine the file size in bytes and divide by the nr of bytes
     an APerson record holds. Now you now the number of records in the
     file and you can start reading them with a For loop... Even a
     better trick is to use un-typed files (lookup the 'File' keyword)
     and when dumping the data to file, first write an Integer with the
     number of records following and then all the records. Perhaps
     you should use the 'BlockWrite' and 'BlockRead' procedures with
     this... Plenty of options, so choose a variant you didn't think
     of yourself...

   Because you already did the next example and I'm out of examples, it
   is now time to start working with FORTRAN 90. Get the handbook of
   FORTRAN 90 and try to find the statements used so far in PASCAL. You'll
   notice PASCAL and FORTRAN don't seem to different... but... there's a
   catch... What is it? Try writing some of the first exercises in FORTRAN
   90. This will be a good experience for you... PASCAL is finished!
}
```

The seventh example…

Building, making and debugging with the Borland Turbo Pascal 7.00 IDE…

```
{ Finally the last example...

  Have a look at the code first and see how the linked list unit is
  used to create a list of 150 records, each containing an integer.
  This list will be initialised with random numbers, printed to
  screen and then sorted. After the sort, the list is printed again
  and deallocated.
}
```

```
Program TestList;

Uses Crt, LnkLst;

Type ARec = Record                          { a simple record containing an integer }
              Value : Integer;
            End; { Record }
     PArec = ^ARec;

{ - - - - - - - - - - - - - GLOBAL VARIABLES - - - - - - - }

Var TheList, Node : PAListNode;    { a general pointer to the list and a node }
    Looper  : Integer;             { a loop variable }

{ - - - - - - - - - - ALL FUNCTIONS - - - - - - - - - - - - - }
{ Allocate function... Must be far or else won't work...    }

Function Allocate : Pointer; Far;
 Begin
       Allocate := New( PARec );          { easy huh ? }
 End; { Function }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ A comparison function... Also far !                        }

Function Compare( P1, P2 : Pointer ) : Integer; Far;
 Begin
      If( PARec( P1 )^.Value > PARec( P2 )^.Value ) Then    { just like the unit describes it... }
       Compare := 1;
      If( PARec( P1 )^.Value < PARec( P2 )^.Value ) Then
       Compare := -1;
      If( PARec( P1 )^.Value = PARec( P2 )^.Value ) Then
       Compare := 0;
 End; { Function }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ A looper function... Also far !                            }

Function PrintItem( Data : Pointer; Message : LoopMessages ) : LoopThroughAction; Far;
Var TheInt : Integer;
 Begin
      TheInt := PARec( Data )^.Value;               { get the data from pointed record into temp var }
      Write( 'DATA: "', TheInt, '" ' );             { print it and continue with the rest... }
      PrintItem := ContinueLoop;
 End; { Function }

{ - - - - - - - - - - ALL PROCEDURES - - - - - - - - - - - - - }
{ The deallocation procedure. Must be FAR !!!               }

Procedure DeAllocate( P : Pointer ); Far;
 Begin
      Dispose( PARec( P ) );         { oh oh oh, how complicated... }
 End; { Procedure }

{ - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - }
{ Just a simple print routine for a list...                  }

Procedure PrintTheList( TheList : PAListNode );
 Begin                                          { yep, also very complicated !!! }
      TheList := LoopThroughListOfNodes( GetFirstNodeOfList( TheList ), LoopForwards, PrintItem );
 End;

{ - - - - - - - - - MAIN PROGRAM HERE - - - - - - - - - - - - }
```

```
Begin
     Randomize;
     Writeln( 'MEM:', MaxAvail );            { a check for memory leaks... }
     For Looper := 1 To 150 Do
      Begin
           Node := AllocateNewNode( Allocate );          { allocate a new node with a record in it... }
           InsertTheNodeInList( Node, TheList );         { insert it in the list }
           PARec( GetNodesData( Node ) )^.Value := Random( 100 ); { oh, right, also give the data some value... }
      End; { For Do }

     Writeln( GetNumberOfNodesInList( TheList ), ' NODES IN LIST !!!' ); { just a simple check... }
     PrintTheList( TheList );
     Writeln( 'SORTING ... press the enter key to continue...' );       { realy needs explaining huh? }
     Readln;

     SortNodesOfList( GetFirstNodeOfList( TheList ), SortAscendingForwards, Compare );
     PrintTheList( TheList );
     DeAllocateCompleteList( TheList, DeAllocate );
     Writeln( 'MEM:', MaxAvail );      { the final leakage check... }
End. { Program }
```

{ Well, the code is very simple and straight forwards... No need to explain things that much. Why the hell then make this example? Well, you nearly learnt the most common tricks of the book. There's still one thing we actually didn't pay any attention too.

Debugging... Why debugging? Well, perhaps you already found out that no program you write runs as it should the very first time. You nearly always make a little mistake or a bug. The trick is to find them in a fast and structured way.

The best method for this is not to write the complete program at once, but to build it routine for routine and check each one of them as soon as you think it is finished. Try to test them and make sure they do what they should. This way you can eliminate routines from the complete code which has been validated already. The bug must be in the part you have just written...

Of course, knowing where more or less the bug is doesn't help fixing it. Sometimes it would be nice if you were able to follow the code while it was running step by step and look at the contents of variables along the way. This is possible with the debugger. Because using debuggers is as general as programming, we'll try and use the internal debugger of Turbo PASCAL to follow the code of this example and the unit(s) it uses...

First activate the debugger. Check under the option menu for the compiler options. Make sure all the checkboxes are checked except the 'Open Parameters' one. Also when viewing the options get the help of each item...

Now we have nearly activated all debug options in the compiler. But the internal debugger itself still has to be turned on. Do this by checking the 'Integrated...' option in the options of the debugger menu...

So, having activated the 'internal' debugger and set the correct compiler options the code needs to be recompiled with these options. Choose the Compile option of the compile menu. Now the currently viewed code is recompiled. This obviously means that the unit this code uses is still compiled with the older options.

To make sure everything this code uses is recompiled with the new options, do a 'build'. This option is also found under the compile menu. The difference between a Compile and a Build is that a Build recompiles anything the current code or 'primary file' uses. Any unit or includes... Normally you wouldn't do

this because it only takes more compilation time and if nothing has changed in the sources, it shouldn't do more than needed. So a compile is compile the currently viewed file and build is do the whole lot all over again.

The thing in the middle is called a 'Make'. Also present in the compile menu, this command only re-compiles those files changed since the last time things where compiled. Of course this also is over anything the primary file or currently viewed code is using...

Do a build now to make sure everything is done over and correct now...

Having done that, we are ready to use the debug facilities. Get the debug menu and have a look at the possible options you have access to... Let's explain in short what it all means...

A breakpoint is a marker in the code. This marker can be put on any line of the code and as soon as during a run this marker is encountered the program stops and gives you all the debug capabilities it offers. To set a 'simple' breakpoint go to the line where you want to place one and press CTRL-F8.

Go to the line with the first Writeln statement of the main program. Put a simple breakpoint there and start running the code. As you can see, the line is inverted to signal the current run position. Now press F8 to do a single step. The code is run for only that single line. Look at the user screen (CTRL-F5) and make sure the writeln statement went okay...

With the single step you can step through the code line by line. As soon as a function or procedure is encountered, the routine is called and executed and the next line will be stepped to. Try to step through the first 5 iterations of the For loop.

Things would be nice if we could actually see what value Looper had each time. Of course the debugger can do this for you. Try the 'add watch' option in the debug menu and make a watch for the 'Looper' variable. You'll find a new window called 'Watches' is created on the desktop. In this window the chosen variables are being 'watched'. Try and step through another couple of iterations of the for loop and pay attention to the 'watch' you just set... Neat huh? A very useful tool in debugging complex code!

But sometimes the code will be so complex you also want to know what all the functions and subroutines do that are called. To step through a piece code and to all the routines being called, try to use the 'Trace into' option from the run menu. Or just press F7 each time. Try this for a single iteration of the for loop. Make sure you end up in the for loop again before we continue with the lesson ! Enjoy... Oh, try to make some watches along the route through the code...

Okay, you now actually have seen the trick of using typed-procedures in action! Cool huh? Makes life easier for programmers...

Now we have made sure the for loop behaves as expected and contains no bugs, we can run the code to the next statement after the loop. There are two methods for doing this. The first is using a breakpoint again. This is easy and you should try this yourself. Of course after this you shouldn't single step or trace into anymore but just run the code. It will stop as soon as the breakpoint is hit. Have a look at the watch of 'Looper' now... Is the value as expected?

```
  But as I mentioned, there's another trick of running to a specified point
  without using a breakpoint. To demonstrate this, put your cursor at the next
  writeln statement just after printing a list. Use the 'Go to cursor' option
  from the run menu or press F4. The code will run up to the cursor now. Check
  the watch again! Make sure things are okay and don't forget to check the user
  screen as well! Is it okay ? Then this means the print routine worked and
  contains no bugs.

  Now trace into the 'SortListOfNodes' routine in such a way you are actually
  executing the code of the sub function present in the 'SortListOfNodes'
  routine. Now use the 'Call stack' option of the debug menu. What do you see?

  You see the calling sequence of the code. The top routine is the one you're
  currently in, the middle one is the one that called the routine you're in
  etc... Helps a lot to figure out where you are and how the code is designed!

  Now create a watch for the Method parameter and try to verify the contents.
  Then trace into through the sorting procedure and take a close look at the
  Call stack window...

  Okay, we had enough of this. It seems to work and everything is okay. Go to
  the main program window, set the cursor on the following PrintList call and
  press F4. Look at the call stack and the watch window... All Okay?!?!

  The rest of the debugging is left as an exercise for you... Trace into the
  PrintList procedure, set a breakpoint in the PrintItem's write statement with
  a counter value of 10 (first set the breakpoint, then get the breakpoint
  window and set the number of counts...). Run the code, check the value of the
  data currently passed into the function and look at the Call stack... All
  seems okay and no bugs are present ??

  Then press CTRL-F2 to reset the code and the debugger and fiddle around with
  some breakpoints and watches. Try to trace into the 'InsertNodeInList'
  routine...

  Having done all this, you are now able of debugging a code... You can continue
  with the exercises of the previous example... You'll find out the use of the
  debugger in that 'bigger' lesson !
}
```