Procedures and Top Down Design.

Top Down design is emphasized quite heavily in this course. We see examples of top down design through all parts of computer science. It is an important way to solve problems on computers by breaking down the problem to be solved into its components repeatedly until we finally reach the level of source code, in Pascal or any other modern computer language.

A key mechanism in developing programs using top down design is the idea of a subprogram. In Pascal subprograms come in the form of procedures and functions.

This part of the syllabus will highlight the following points:

1. A subprogram is a part of a program which has all the elements of a program itself.
2. That identifiers may have values through the entire course of a program or may be made local to a subprogram by using scoping rules.
3. That variables can be passed between a (sub)program to a subprogram allowing work to be divided up between the various subprograms and their results shared.

1 Why subprograms?

When designing programs in a top down manner it is necessary to have a mechanism to group actions together into subprograms.

The subprograms can be called on repeatedly by the main program to perform the set of actions that they are designed to do.

The subprogram unit that we study now is the procedure.

2 Procedures.

2.1 A motivating example.

The first example of how a subprogram helps in the top down design process comes from trying to draw a rectangle.

We want to be able to draw a rectangle that looks something like.

```
********************
* * * *
* * * *
* * * *
* * * *
********************
```

We design the program using a top down approach.

The following pseudocode is a first pass at our solution.

```
Draw a horizontal line 20 units long
Draw two vertical lines 18 spaces apart
Draw a horizontal line 20 units long.
```

The top down design of the program looks something like this.

```
MAIN
----------
```
Horizontal and Vertical will be two procedures used by the main part of the program to draw the rectangle.

2.2 The main part of the program.

The main part of our program may well look like

```
PROGRAM Rectangle(input,output);
VAR ....
BEGIN
  Horizontal;
  Verticals;
  Horizontal
END.
```

When the main program comes to the first line, ie Horizontal it starts the subprogram working and the subprogram, in this case the procedure is said to be activated.

The referral to a procedure by the main program is called a call.

2.3 Designing the subprograms.

Now how would we design a subprogram to design a horizontal line twenty units apart?

We do this in exactly the same way as we would when designing a normal program without subprograms. This is all very simple and much the same as before, except this time we call our subprogram a PROCEDURE instead of a PROGRAM and we end our procedures with a semi-colon instead of a full stop.

```
PROCEDURE Horizontal;
CONST Width = 20;
VAR J:integer
BEGIN
  FOR J := 1 TO Width DO
    Write('x');
    Writeln('');
END;
```

Similarly Verticals can be written as follows

```
PROCEDURE Verticals;
CONST Height = 6, Width := 20;
VAR K :integer;
BEGIN
  FOR K := 1 TO Height DO
    writeln('x','x':Width-1)
END;
```

As promised procedures look exactly like programs except that they are called procedures and not programs and end with a semicolon.

They have the same elements, ie an identifier, CONST and VAR declarations and a body.

Procedures need to precede the program body. They are declarations, just like VAR and CONST were, for the main program to use.
2.4 The final program.

So now with the procedures implemented our main program now can use them freely and it looks like.

```pascal
PROGRAM Rectangle(input,output);

PROCEDURE Horizontal;
CONST Width = 20;
VAR J:integer
BEGIN
  FOR J := 1 TO Width DO
    Write('x');
  Writeln('');
END ;

PROCEDURE Verticals;
CONST Height = 6, Width := 20;
VAR K :integer;
BEGIN
  FOR K := 1 TO Height DO
    Writeln('x','x':Width-1)
END;

BEGIN
  Horizontal();
  Verticals();
  Horizontal()
END.
```

In the above example the procedure `Horizontal` is activated twice and the procedure `Verticals` is activated once.

3 Global and Local Identifiers.

Let’s look at the variables in the program below.

```pascal
PROGRAM Rectangle(input,output);
VAR Height:integer;

PROCEDURE Horizontal;
CONST Width = 20;
VAR J:integer
BEGIN
  FOR J := 1 TO Width DO
    Write('x');
  Writeln('');
END ;

PROCEDURE Verticals;
CONST Width := 20;
VAR K :integer;
BEGIN
  FOR K := 1 TO Height DO
    Writeln('x','x':Width-1)
END;

BEGIN
  Height := 6;
```
3.1 Global Variables.

Height is a variable declared inside the main program but not in any of the subprograms. Variables like this are called **global variables**. They have the property that they may be used at any point in the program.

We set Height in the main program to be six and then when we activate `Verticals()` we use the global variable height to determine how long the vertical lines ought to be.

3.2 Local Variables.

The variables J and K are examples of **local** variables. They are only accessible within the procedure in which they are defined and any procedures defined within it. ie. If we tried to use J or K in the main part of the program we’d get a compile error saying that the variables J and K are undeclared.

We call the region in which a variable is defined a **scope**. We will discuss this in more detail later.

In the ANSI Pascal standard all loop control variables must be declared locally.

3.3 Local variables simplify programs.

This use of local variables allows us to simplify our programs once again. If we had twenty control variables in different loops in our program, each in a different sub program we needn’t chose separate names for each of them. We could just call the control variable control and redefine it local to each procedure.

eg. Counter is declared locally

```pascal
PROGRAM Rectangle(input, output);
VAR Height, Counter: integer;

PROCEDURE Horizontal;
CONST Width = 20;
VAR Counter: integer
BEGIN
    FOR Counter := 1 TO Width DO
        Write(‘x’);
    Writeln(’’);
END;

PROCEDURE Verticals;
CONST Width := 20;
VAR Counter : integer;
BEGIN
    FOR Counter := 1 TO Height DO
        Writeln(‘x’, ’x’: Width-1)
END;

BEGIN
    Counter := 6;
    Height := Counter;
    Horizontal();
    Verticals();
    Horizontal()
END.
```
Now *Counter* is declared in three places. Once in each of the procedures and once globally in the main program. Each *Counter* is actually different. i.e. The counter declared in `Horizontal()` could actually be called `HorizontalsCounter`, the counter declared in `Verticals()` could be called `VerticalsCounter` and the main program’s counter could be called `RectanglesCounter`. They each correspond to different locations in memory and hold different values. They are different variables they just share the same identifier. Infact the computer keeps track of which one is which by renaming the variables in a similar manner.

### 3.4 The most closely nested declaration rule.

So how do we decide which version of counter we’re using? We use the following rule.

**The variable used is always that of the most closely nested declaration of that variable.**

So the uses of counter in each of the procedures is of the Counter variables declared in the procedures. Counter at the program level only affects the value of `Height` and not of the Counter variables in the loop.

### 3.5 Side Effects.

Using global variables is a bad idea. Look at the following code.

```pascal
PROGRAM test;
VAR One:integer;

PROCEDURE Deep;
BEGIN
  One := 3;
END;

BEGIN
  One := 1;
  Deep();
  Writeln('The value of One is ',One)
END.
```

Because the procedure `Deep()` used no local variables the variable it refers to in `One` is the global variable `One`. So when we call `Deep()` from our main program it has the *side effect* of setting `One` to 3. In general when our programs become big it’s important to make sure we don’t create any unwanted side effects. For this reason we avoid global variables, or at least use them very sparingly.

### 4 Value Parameters.

Value parameters give us a way to pass information between subprograms.

### 4.1 Drawing a patterned rectangle.

We wish to design a program which when given three numbers will produce the following output.

```
xxxxxxx------
xxxxxxx------
xxxxxxx------
xxxxxxx------
xxxxxxx------
```

4.2 Designing the program.

We go through the familiar top down design steps.

The pseudocode is simply:

```
Read in number of X’s, number of -’s, number of lines.
FOR K := 1 TO Number of Lines DO
    Pattern
```

The top down design diagram looks like

```
Main
  |
Pattern
```

And Patterns pseudocode looks like

```
print the number of X’s supplied by main
print the number of -’s supplied by main
print a new line.
```

4.3 Using global variables to share information.

Consider this program which uses global variables to achieve this.

```
PROGRAM IntroProc;
VAR Outer, NumX, NumDash, NumLine:integer;

PROCEDURE Pattern;
VAR Xindex, Dindex:integer
BEGIN
    FOR Xindex := 1 TO NumX DO
        Write(’x’);
    FOR Dindex := 1 TO NumDash DO
        Write(’-’)
    Writeln(’’);
END;

BEGIN
    Writeln(’How many Xs, -s and lines ?);
    readln(NumX,NumDash,NumLine);
    FOR Outer := 1 TO NumLine DO
        Pattern()
END.
```

Our program would then print out patterns as described above. The variables LimitX and LimitD are global variables which are used to pass in values to the subprogram. In this case they control how many x’s and how many -’s we’re going to print.

As we’ve just discovered however, using globals is considered bad practice since it may introduce unanticipated and therefore unwanted side-effects. Since Pascal was designed to be a safe and structured programming language it seems natural for there to be a way of passing information between procedures without using global variables.
4.4 Value Parameters.

We rewrite our program using **Value Parameters** to pass this information into the procedure. Pascal’s parameter passing mechanism looks like this. When the main program activates the procedure Pattern it will call it with two values one for the number of X’s and one for the number of -’s.

4.4.1 The format of the activating call.

The format of the call looks like this:

```
Pattern(NumX, NumDash);
```

Here **NumX** and **NumDash** are parameters that will be passed into the procedure.

4.4.2 Parameterizing the subprogram.

To be able to accommodate, and make sense of the extra information being passed in we need to rewrite **Pattern** to expect the new value parameters.

This is done by changing the procedure header to:

```
PROCEDURE Pattern(LimitX, LimitD: integer);
```

And now we rewrite the **Pattern** procedure body to use **LimitX** and **LimitD** instead of **NumX** and **NumDash**.

```
PROCEDURE Pattern(LimitX, LimitD: integer);
VAR Xindex, Dindex: integer;
BEGIN
  FOR Xindex := 1 TO LimitX DO
    Write(‘x’);
  FOR Dindex := 1 TO LimitD DO
    Write(‘-’);
  Writeln(‘’);
END;
```

4.4.3 Actual and Formal Parameters.

We need to define some terms.

**Actual Parameters** The variables used in the procedure call are known as actual parameters. In our example they are **NumX** and **NumDash** in the line `Pattern(NumX, NumDash);` They are the actual values used by the subprogram when it is activated.

**Formal Parameters** The variables defined in the procedure header are known as formal parameters. These are variables local to the procedure.

For every actual parameter we need a corresponding formal parameter. When the procedure is activated the computer copies the values of each actual to it’s corresponding formal. If we have actuals **A1**, **A2**, **A3** ... they are copied to the formals **F1**, **F2**, **F3** respectively. Note **A1** and **F1** have to be of the same type for the copy work. You have to supply the actuals in the correct order, the compiler simply does a series of assignments which are like **F1 := A1, F2 := A2** ...

4.5 The final program.

By doing things this way we now have a new program where we don’t use global variables to pass information into a procedure.
PROGRAM IntroProc;
VAR Outer, NumX, NumDash, NumLine:integer;

PROCEDURE Pattern(LimitX,LimitD:integer);
VAR Xindex, Dindex:integer;
BEGIN
  FOR Xindex := 1 TO LimitX DO
    Write('x');
  FOR Dindex := 1 TO LimitD DO
    Write('-');
  Writeln('');
END;

BEGIN
  Writeln('How many Xs, -s and lines ?);
  readln(NumX,NumDash,NumLine);
  FOR Outer := 1 TO NumLine DO
    Pattern(NumX,NumDash);
END.

5 Designing Procedures.

Learning Top Down design is easy in principle but it’s actually not something you can get right until you’ve written quite a few complicated programs and read many examples of top down design. We’re not yet studying how to organise data so the design process is still relatively simple.

We now turn our attention to some examples of top down design and how to design procedures accordingly. None of this is gospel, it’s just a set of conventions that programmers use based on experience.

5.1 Drawing an income graph.

This program reads NumData pairs of data, consisting of a person’s ID and income, and makes a histogram (bar graph) of the income using bars of x’s such that one x represents $1000.

The pseudocode for this program is

FOR J := 1 to NumData DO
  BEGIN
    Read the ID and Income;
    Plot the income
  END.

Plot the income will become the procedure Plot. The top down diagram is

MAIN
    |
    |
Plot

It is often best to develop your program in stages. Once we've finalized the design we can use the following approach to building procedures.

1. Incorporate the statements that will eventually be a procedure into a separate program.
2. Once the program is tested, convert the appropriate statements into a procedure and test it from an appropriately written main program.
3. Alter the main program so that it will use the procedure as your goal dictates.
5.2 Forming the program

The program below will become the procedure plot in our histogram program eventually. We write it as a standalone program initially to test the behaviour of the procedure.

Since the program will only read one set of data we set NumData to 1 in the CONST definition. In the statement part we divide income by 1000 and assign the result to RndIncome, using it as the upper bound in a loop to print out a bar with RndIncome x’s in it.

```
PROGRAM Graph(input, output);
(* Makes a histogram of people’s income *)
CONST NumData = 1; Blank = ‘ ’;
VAR
  Income: real;
  RndIncome, K, ID: integer;
BEGIN
  writeln(‘Type ’, NumData, ‘ sets of: ID# followed by Income’);
  read(ID, Income);
  writeln;
  write(‘ID=’, ID:4, ‘Income=’, Income:6:0, Blank);
  RndIncome := round(Income) DIV 1000;
  FOR K:= 1 TO RndIncome DO
    write(‘x’);
  writeln;
  writeln(‘_________________________________’);
  writeln(‘Each X represents $1000’)
END.
```

Note: our program will draw the bars sideways!

5.3 Forming the Procedure.

We take the statements

```
write(‘ID=’, ID:4, ‘Income=’, Income:6:0, Blank);
RndIncome := round(Income) DIV 1000;
FOR K:= 1 TO RndIncome DO
  write(‘x’);
```

And incorporate them into a procedure. This means that ID and Income as parameters to the procedure.

The procedure heading looks like this:

```
PROCEDURE Plot(ID:integer; Income:real)
```

RndIncome and K can also be declared as local variables in Plot, so we move their declaration from the main program to the procedure.

In general we try to move as many variables into procedures as possible. Global variables are seldom really needed and can often cause problems as your program grows.

We get the program below.

```
PROGRAM Graph(input, output);
(* Makes a histogram of people’s income *)
```
VAR Income: real;
   J, ID:integer

PROCEDURE Plot(ID:integer;Income:real);
(* Produces a histogram of incomes *)
VAR RndIncome, K:integer;
BEGIN
   write('ID=',ID:4,'Income=',Income:6:0,Blank);
   RndIncome := round(Income) DIV 1000;

   FOR K:= 1 TO RndIncome DO
      write('x');
   END (*Plot*);
END (*Plot*);

BEGIN
   writeln('Type ', NumData, ' sets of: ID# followed by Income');
   read(ID, Income);
   writeln;
   Plot(ID,Income);
   writeln;
   writeln('_________________________________');
   writeln('Each X represents $1000');
END.

The compiler sets aside two sets of variables Income and ID one for the procedure and one for the main program. When the procedure is called the variables from the main program are copied into the variables for the procedure.

5.4 Altering the main program.

The point of transforming Plot into a procedure was to be able to call plot as many times as we wish.

We could now make the main program include a loop that read in input from the user and produced a series of bars. We could add a section like this into the main program.

FOR J:= 1 TO NumData DO
   BEGIN
      read(ID,Income);
      writeln;
      Plot(ID,Income);
   END (*for J*);

5.5 The basic histogram program.

The basic histogram program now looks like:

PROGRAM Graph(input, output);
(* Makes a histogram of people’s income *)

CONST NumData = 1; Blank = ‘ ’;
VAR
Income: real;

PROCEDURE Plot(ID:integer;Income:real);
(* Produces a histogram of incomes *)
VAR RndIncome, K:integer;
BEGIN
  write('ID=',ID:4,'Income=',Income:6:0,Blank);
  RndIncome := round(Income) DIV 1000;
  FOR K:= 1 TO RndIncome DO
    write('x');
END (*Plot*)

BEGIN
  writeln('Type ', NumData, ' sets of: ID# followed by Income');
  FOR J:= 1 TO NumData DO
    BEGIN
      read(ID,Income);
      writeln;
      Plot(ID,Income);
    END (*for J*);
  writeln;
  writeln('_________________________________');
  writeln('Each X represents $1000');
END.

6 Variable Parameters.

The flow of information so far has gone from the main program into it’s sub-programs. Actually, any program can pass values into its subprograms. You might think, from what we have seen so far, that the flow of information only goes inwards, this isn’t true. Sub-programs are often used to modify input data and pass back a result. We’ve already seen procedures like this. Read() and Readln() both read in a value from the keyboard and modify a variable.

We can write procedures that pass back information using variable parameters

To demonstrate the use of variable parameters we write a program that averages five numbers in the range 0 - 9. The input will an integer five characters long.

The pseudocode for our program is as follows:

Read number;
Sum := 0;
FOR index := 1 TO 5 DO
  BEGIN
    Remove leftmost digit (* procedure *)
    Sum := Sum + digit;
  END
Calculate average;
writeln(average);

To split a five digit integer like 12345 into individual values 1, 2, 3, 4, and 5 we make the following observation.

We get the left most digit of a:

five digit integer using N DIV 10000; four digit integer using N DIV 1000; three digit integer using N DIV 100; two digit integer using N DIV 10; one digit integer using N DIV 1;

Let’s call the number on the right hand side of the DIV operator the Power
We use the keyword VAR again in the procedure header. The VAR here means that the variables Grade, Power and Number are all modifyable by the subprogram. The subprogram is simply:

```pascal
PROCEDURE Remove(VAR Grade, Power, Number: integer);
(* Removes leading digit *)
BEGIN
  Grade := Number DIV Power;
  writeln('Mark is', Grade: 2);
  Number := Number MOD Power;
  Power := Power DIV 10;
END
```

With VAR infront of the variable declarations, the behaviour of an activation is equivalent to:

1. Copy in the variables from the main program to the subprogram.
2. Do the computation using the variables.
3. Copy out the variables back from the subprogram to the main program.

So for every activation of `Remove()`, the variables passed in, i.e. Grade, Number, and Power are modified. When the activation is over and the procedure is returned the new values for Grade, Number, and Power are copied back into the main program.

If we put `Remove()` into a loop and execute it 5 times we’ll effectivly split a five digit number into five seperate digits.

```pascal
PROGRAM ExamAverage(input, output);
(* Average 5 quiz marks *)
VAR Sum, Mark, Index, Power: integer;
  Number: longint;
  Average: real;
PROCEDURE Remove(VAR Grade, Power, Number: integer);
(* Removes leading digit *)
BEGIN
  Grade := Number DIV Power;
  writeln('Mark is', Grade: 2);
  Number := Number MOD Power;
  Power := Power DIV 10;
END
BEGIN
  writeln('Type 5 contiguous digits FOR quiz marks');
  readln(Number);
  (* First calculate 10 ** 4 *)
  Power := 10 * 10 * 10 * 10;
  Sum := 0;
  (* Now remove the leading digit and add TO sum *)
  FOR Index := 1 TO 5 DO
    BEGIN
      Remove(Mark, Power, Number);
      Sum := Sum + Mark;
    END;
  Average := Sum / 5;
  writeln('Quiz average is ', Average: 4: 1)
END.
```
Let's trace this through:

Take the initial value 34567

The variables on each pass through the loop become:

<table>
<thead>
<tr>
<th>Index</th>
<th>Mark</th>
<th>Power</th>
<th>Number</th>
<th>Sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>-</td>
<td>10000</td>
<td>34567</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>1000</td>
<td>4567</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>100</td>
<td>567</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>10</td>
<td>67</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>1</td>
<td>7</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>25</td>
</tr>
</tbody>
</table>

; - means uninitialized.

7 Variable vs Value Parameters.

Let's explore the differences between variable parameters and actual parameters carefully.

Actual Parameters: On activation these are copied into the called procedure from the calling procedure. This is also called call-by-value.

Variable Parameters: On activation these are copied into the called procedure from the calling procedure. The calling procedure then performs its computation and when it is finished copies its arguments back to the calling procedure. This is also called call-by-reference.

The following programs illustrate the differences.

```pascal
PROGRAM Test(input,output);
VAR A, B;
PROCEDURE Swap(C,D:integer);
VAR Temp:integer;
BEGIN
   Temp := C;
   C := D;
   D := Temp;
END;
BEGIN
   Writeln('Type in two integers a and b);
   Readln(A,B);
   Swap(A,B);
   Writeln('A and B are ',A:4,B:4)
END.
```

Does this work?

7.1 Why swap fails with value parameters.

Swap failed with value parameters. Why?

A and B are copied into C and D on activating Swap(A,B). Now the values of C and D are swapped correctly. Then the procedure exits and A and B are left unchanged. So Swap(A,B) doesn't do what it's expected to do.

7.2 Swap will work with VARiable parameters.

This version does work.
PROGRAM Test(input,output);
VAR A, B;

PROCEDURE Swap(VAR C,D:integer);
VAR Temp:integer;
BEGIN
  Temp := C;
  C := D;
  D := Temp;
END;

BEGIN
  Writeln('Type in two integers a and b);
  Readln(A,B);
  Swap(A,B);
  Writeln('A and B are ',A:4,B:4)
END.

Why? As before on activation Swap(A,B) copies the values of A and B into C and D respectively. The procedure then swaps the values of C and D. Now on exiting the procedures the values, which have been swapped, are copied back into A and B.

8 Another difference ... 

You can write an expression in the place of an actual for a value parameter. i.e. we could write Remove( 22*300 ). But we can't do the same for variable parameters. The reason for this is that a variable parameter must have a place to be copied back to. Expressions aren't variables, so they don't have places in memory for us to copy them back to.

9 Nesting Procedures and the Scope of Identifiers.

Any procedure can declare a set of subprograms within itself. We need to consider the behaviour of variables and understand where a variable declaration is valid.

The region of the program in which a variable declaration is valid is known as the scope of the variable.

Look at the program below:

PROGRAM Test(output);
VAR One : integer;

PROCEDURE Deep;
VAR Two : integer;

PROCEDURE Deeper
VAR Three:integer;
BEGIN
  Three := 3;
  writeln('Main, One :=',One);
  writeln('Deep, Two :=',Two);
  writeln('Deeper, Three :=',Three);
END;

BEGIN (*Deep*)
  Two := 2;
  writeln('Main, One := ', One);
  writeln('Deep, Two := ', Two);
  Deeper;
END;

BEGIN (* Main *)
  One := 1;
  writeln('Main, One := ', One);
  Deep;
END.

The scoping rule: A variable declared in a program is usable in that program and in any subprograms that it defines or in any subprograms of those subprograms.

So when we run the above program we get:

Main, One := 1
Main, One := 1
Deep, Two := 2
Main, One := 1
Deep, Two := 2
Deeper, Three := 3;

If in the main program body we’d decided to use the following line:

    writeln('Deep, Two := ', Two);

we’d get a compile time error. This is because the variable Two isn’t declared in the scope of main. Two is declared only within Deep and Deeper.

10 The Readln(), Read(), Writeln() and Write().

You should recognise by now that Readln(), Read(), Writeln(), and Write() are all procedures. They are known as standard procedures because all ANSI standard Pascal systems are expected to provide them for the programmer to use.

Standard procedures perform commonly used tasks and must behave in a manner specified exactly by the ANSI committee on Pascal.

11 Why we use subprograms.

Using subprograms makes planning, reading and writing a program easier. They mirror the top-down design of our programs making it easy for us to break up tasks into smaller more manageable pieces.