The FOR–DO Loop and the Ordinal Types

One of keys to making a programming language versatile and powerful is to provide mechanisms for repeating a section of instructions. This should be clear intuitively. Computers are suited to running long explicit sequences of instructions and often, as we’ve seen in our development of programs so far, we need a mechanism to repeat a set of instructions over and over.

The FOR–DO loop in Pascal is one of a few mechanisms provided by the language for repeating these sequences of instructions. In studying this elementary, but powerful, form of looping we will see that integers can be used to control the number of times we move through a loop. In fact we’ll see that any type that has discrete values, called an ordinal type, can be used to decide how many times a loop is executed. This is where our programming becomes more complex and our view of programs becomes a little more sophisticated.

1 The FOR–DO loop.

The general form of a FOR–DO loop is

```pascal
FOR controlVariable := lowValue TO highValue DO
  statement or compound statement
```

Let’s explore this in detail:

- FOR controlVariable := lowValue TO highValue DO is known as the loop header. It has the following parts.
  - A control variable, this variable keeps track of how many times the loop has been executed. This variable is often called the Index.
  - A pair of values one lower than the other, which together tell you how many times the program needs to go through a loop. The control variable is initially set to the low value and increases by some increment to the high value with each pass through the loop.

- The loop header is followed by either a single statement or by a compound statement. So loop bodies can have the form

  a statement;

  or

  BEGIN ... statement(s) ... END

  Each pass through the loop is called an iteration.

1.1 A FOR–DO loop with a single statement.

Here is a trivial example of a FOR–DO loop.

```pascal
PROGRAM SillyLoop (output);
VAR Index:integer;
BEGIN
  FOR Index := 1 TO 5 DO
    Writeln('This is silly five times.');
    Writeln('Done looping.');
END.
```

Which prints,

```
This is silly five times.
This is silly five times.
This is silly five times.
This is silly five times.
This is silly five times.
Done looping.
```

1.2 How did it do that?

Let’s turn our attention to how the FOR–DO loop achieved this.

Initially we started our program by declaring a variable called Index of type integer. At the time we reach the loop header we initialize the variable to the lower value, in this case 1, and the loop then checks to see if the value is the same as the higher value, 5. This check determines whether we should execute the loop or not. Since the value of Index isn’t greater than 5 we move into the body of the loop and execute the Writeln().

Once we’ve finished the body of the loop the value of Index is incremented by 1 and Index now has the value 2. Now when we do the check we see that the value of Index is still not greater than 5 so we move back into the loop.

1.2.1 The implicit update rule.

You can think of the body loop as having an implicit statement at the bottom which says

```pascal
ControlVariable := ControlVariable + 1;
```

ie. In the loop structure you have by default the update line and so you don’t have to explicitly program in something like.
FOR ControlVariable := Lower TO Higher DO
BEGIN
  ControlVariable := ControlVariable + 1
END

1.3 A FOR–DO loop with a compound statement.

If we’d had a compound statement for the loop body we’d execute the entire compound statement. Consider the next example

PROGRAM MoreSillyLooping (output);
VAR Index:integer;
BEGIN
  FOR Index := 1 TO 5 DO
  BEGIN
    Writeln('This isn’t silly five times.');
    Writeln('It’s very silly!');
    Writeln('Done looping.');
  END (* FOR *);
END.

Which will produce the output.

This isn’t silly five times.
It’s very silly!
This isn’t silly five times.
It’s very silly!
This isn’t silly five times.
It’s very silly!
This isn’t silly five times.
It’s very silly!
This isn’t silly five times.
It’s very silly!
Done looping.

1.3.1 Indentation alone does not affect the scope!

If we’d just written

PROGRAM MoreSillyLooping (output);
VAR Index:integer;
BEGIN
  FOR Index := 1 TO 5 DO
  BEGIN
    Writeln('This isn’t silly five times.');
    Writeln('It’s very silly!');
    Writeln('Done looping.');
  END;
END.

we’d have only executed the first Writeln() statement 5 times but not the second one. All the instructions that need to be looped through need to be gathered together in a compound statement.

The second Writeln() is actually at the same level as the third Writeln().

2 Using the control variable in the body of the loop.

We can use the control variable in the body of the loop too. This program tells you what line it’s printing.

PROGRAM WhichLineIsThis(output);
VAR Index:integer;
BEGIN
  FOR Index := 1 TO 5 DO
  BEGIN
    Writeln('We’re now on line ’,Index);
  END (* FOR *);
END.

Which will produce

We’re now on line 1
We’re now on line 2
We’re now on line 3
We’re now on line 4
We’re now on line 5

There are many situations where it is desirable to use the control variables in a loop.

2.1 An example using using a control variable in a loop.

To demonstrate the use of a control variable in a loop let’s write a simple program to create a quick table for calculating tips at a resteraunt.

We start by calculating a price for a meal in terms of the control variable and then working out what the tip should be. For good measure we can add the two together to get the total.

The control variable only goes up in increments of one. Lets assume that the price of a meal goes up in 50 cent increments in our table too.

We want our table to look like this.

<table>
<thead>
<tr>
<th>Price</th>
<th>Tip</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.50</td>
<td>0.08</td>
<td>0.58</td>
</tr>
<tr>
<td>1.00</td>
<td>0.15</td>
<td>1.15</td>
</tr>
<tr>
<td>1.50</td>
<td>0.23</td>
<td>1.73</td>
</tr>
</tbody>
</table>
We start by considering how to calculate the price of a meal.

The price of a meal increases by 50 cents each time. If the control variable increases by 1 each time it should be obvious that we only need to multiply the control variable by 50 cents, (or 0.5 dollars) to get the price of a meal. From that we can work out the tip as being 15 percent of the meal price, and the total as being the tip plus the total.

\[
\begin{align*}
\text{meal} &= 0.5 \times \text{Index} \\
\text{tip} &= \text{meal} \times 0.15 \\
\text{total} &= \text{meal} + \text{tip}
\end{align*}
\]

We want to go up to meals costing 20 dollars, so we need to have \(20 / 0.50\) iterations, ie. 40 steps through the loop.

```
PROGRAM TipChart(output);
VAR Index:integer;
Meal, Tip, Total:real;
BEGIN
  Writeln('Meal ':7,' Tip ':7,'Price':7);
  Writeln('------+-----+------');
  FOR Index := 1 TO 40 DO
    BEGIN
      Meal := Index * 0.50; (* Note: Mixed mode *)
      Tip := Meal * 0.15;
      Total := Meal + Tip;
      Writeln(Meal:6:2,Tip:6:2,Total:6:2)
    END
  END.
```

Which gives us the output we wanted.

### 2.2 A second example. Using control variables in a loop.

A second, somewhat simpler, example is how to produce the following output on the top of your screen. The two lines are intended to be a ruler that tells you what columns you’re printing in. The numbers are read from top to bottom by column.

```
00000000001111111111...77
01234567890123456789...89
```

We can set up a loop to count from 0 to 79. But how do we print the numbers one on top of the other?

The answer is we use two loops; one to generate the first line, and another to generate the second line.

```
PROGRAM ScreenRuler(output);
VAR Index,Temp:integer;
BEGIN
  FOR Index := 0 TO 79 DO
    BEGIN
      Temp := Index DIV 10;
      write(Temp:1);
    END
  Writeln('');
  (* Note: Index has been reset to Zero *)
  FOR Index := 0 TO 79 DO
    BEGIN
      Temp := Index DIV 10;
      write(Temp:1);
    END
END.
```

### 3 Using the variable limits for control.

We can actually change the limits of the variables too. Our loops don’t have to start at 1. We saw that in the ScreenRuler example where we started from 0.

Let’s go back to the Tip table example. No meal costs 50 cents, (and if it did we wouldn’t be tipping 15 percent for it!) So clearly we want to be able to start at a reasonable price. Let’s say at $3.00 for a breakfast special. $3 represents starting our loop at count 6.

```
PROGRAM TipChart(input,output);
CONST Low = 7;
VAR Index:integer;
Meal, Tip, Total:real;
BEGIN
  Writeln('Meal ':7,' Tip ':7,'Price':7);
  Writeln('------+-----+------');
  (* Now start iterating from Low *)
  FOR Index := Low TO 40 DO
    BEGIN
      Meal := Index * 0.50; (* Note: Mixed mode *)
      Tip := Meal * 0.15;
      Total := Meal + Tip;
      Writeln(Meal:6:2,Tip:6:2,Total:6:2)
    END
  END.
```

How do we generate the first line? We turn back to our integer operators for a hint. The numbers are nothing but 0 through 79 divided by 10. Remember we’re using DIV the integer divider so 0 to 9 DIV 10 is 0, 10-19 DIV 10 is 1, ...

The second column should be obvious by now, we want the remainders of division by 10 so we use MOD instead of div.

Now the program is easy to write.

```
PROGRAM ScreenRuler(output);
VAR Index,Temp:integer;
BEGIN
  FOR Index := 0 TO 79 DO
    BEGIN
      Temp := Index DIV 10;
      write(Temp:1);
    END
  Writeln('');
  (* Note: Index has been reset to Zero *)
  FOR Index := 0 TO 79 DO
    BEGIN
      Temp := Index DIV 10;
      write(Temp:1);
    END
END.
```
3.1 We can use a variable to set the limits too.

The value of Low doesn’t need to be hard coded into our program. We could actually prompt the user for input and work out what the minimum value should be.

We could ask the user to directly figure out what the low value on the loop needs to be but in general that’s a bad idea. We like to keep the user of a program blissfully unaware of the way the program works. This is called abstraction. To the user the program should be a black box into which he has no insight. All the user needs to do is supply input and read output in a way that is meaningful to his problem, but not necessarily to the way we, the programmers, have implemented the solution.

For this reason we ask the user for input in terms of real currency, ie fifty cent units, the program we write depends on integers to control the number of iterations through the loop.

We need to do a little extra work to make this conversion from the real world to our programs view of the values.

The conversion is simple. Read in a real variable, for the minimum cost of a meal and then convert it into low using a little mixed mode arithmetic.

```pascal
VAR Min:real;
Low:integer;
Low := (Min / 0.50) DIV 1;
```

This changes our real input into the beginning Index variable which is an Integer.

Min/0.50 gives us the number of 50 cent units needed. This is still a real number.

To get the correct integer result we divide, using DIV by 1. Low is an integer too.

So our program becomes.

```pascal
PROGRAM TipChart(input,output);
VAR Low, Index:integer;
Min, Meal, Tip, Total:real;
BEGIN
  Writeln('Enter the cheapest meal you’ll tip \ for (in 50 units).');
  Readln(Min);
  Writeln('Enter the most expensive meal you’ll tip \ for (in 50 units).');
  Readln(Max);
  Low := (Min / 0.50) DIV 1;
  High := (Max / 0.50) DIV 1;
  Writeln('Meal ':7,' Tip ':7,'Price':7);
  Writeln('------+-----+------');
  (* Now start iterating from Low *)
  FOR Index := Low TO High DO
    BEGIN
      Meal := Index * 0.50; (* Note: Mixed mode *)
      Tip := Meal * 0.15;
      Total := Meal + Tip;
      Writeln(Meal:6:2,Tip:6:2,Total:6:2)
    END
  END.
END.
```

3.2 We can change both the limit variables.

We could expand our example to go to the highest meal cost for an individual as well.

```pascal
PROGRAM TipChart(input,output);
VAR Low, High, Index:integer;
Min, Max, Meal, Tip, Total:real;
BEGIN
  Writeln('Enter the cheapest meal you’ll tip \ for (in 50 units).');
  Readln(Min);
  Writeln('Enter the most expensive meal you’ll tip \ for (in 50 units).');
  Readln(Max);
  Low := (Min / 0.50) DIV 1;
  High := (Max / 0.50) DIV 1;
  Writeln('Meal ':7,' Tip ':7,'Price':7);
  Writeln('------+-----+------');
  (* Now start iterating from Low *)
  FOR Index := Low TO High DO
    BEGIN
      Meal := Index * 0.50; (* Note: Mixed mode *)
      Tip := Meal * 0.15;
      Total := Meal + Tip;
      Writeln(Meal:6:2,Tip:6:2,Total:6:2)
    END
  END.
END.
```

4 Ordinal Types.

Mathematically speaking an ordinal number indicates a position in a sequence. Thus first, second and third are ordinal numbers.
4.1 Integers can have ordinal values.

We can assign ordinal numbers to integers. Zero could be the first ordinal in a sequence, One could be the second... The key to having an ordinal relationship is being able to order numbers by determining what number comes next. eg the successor of 1 is 2 in the integers, the successor of 2 is 3.

4.2 Real numbers cannot have ordinal values.

Real numbers can’t have ordinals because it’s impossible to tell what the successor to a real number is. What is the successor to 2.3? Is it 2.31? Or is it 2.301? Or does 2.3001 sound more convincing? The answer is there is no ordinal relationship here. There is no successor to a real number. In fact between any two real numbers there exists an infinite amount of other real numbers.

4.3 Characters can have ordinal values.

The Pascal compiler associates an integer - called the ordinal position with each character available. The table contains all the codes for the characters, displayable and non displayable.

Different compilers may use different tables. Most compilers use ASCII code but some like IBM mainframe compilers use other standard tables like EBCDIC.

The ASCII table assigns an ordinal to the character ‘B’ that is one higher than the ordingal for ‘A’. This allows us to view ‘B’, as the successor of ‘A’. Look at the table below. Each successor character has an ordinal number which is higher than its predecessor. The ordering in this table is known as the collating sequence.
The ASCII values, in decimal.

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

+-------+-------+-------+-------+-------+-------+-------+-------+
4.4 Finding the ordinal value of a character using ord().

To find the **ordinal value of a character** we use the predefined Pascal function `ord()`.

eg. 
```
VAR Code:integer;
  ...
Code := ord('B');
```

This will give Code the value 66 by looking up the ASCII table. Note: the character is in quotes as always, if it weren't we'd be referring to a variable.

The **ordinal values of an integer** is the same as the integer's values. ie. the `ord(8)` is 8, `ord(-8)` is -8 and `ord(0)` is 0.

The **ordinal value of a boolean value** is 1 for `ord(true)` and 0 for `ord(false)`.

4.5 Finding the successor or predecessor of an ordinal value using succ() or pred().

As mentioned before variables of ordinal types possess a successor and predecessor relation. Pascal has two other standard functions called `succ()` and `pred()` which when applied to an ordinal value yield the successor and the predecessor of that value. eg. `succ('a')` yields 'b'. If we chose the last ordinal value, in the case of the booleans, `TRUE`, then we get an execution error when we try to evaluate `succ('TRUE')`.

4.6 The inverse of an ordinal value, using chr().

The inverse of the ordinal function for characters is the function `chr()`. If we give `chr()` an ordinal the associated character is printed. eg. `chr(66)` is 'B'. If you try to use `chr()` on a non existant ordinal value then you'll get a run time error. eg. `chr(-4)` is an error.

4.7 An example of ordinals being used.

Here is a simple program that demonstrates some of the ideas we've seen above.

```
PROGRAM OrdinalStuff(input, output);
VAR ch:character;
  int:integer;
  bool:boolean;
BEGIN
  Writeln('Type a character');
  Readln(ch);
  Writeln('Ch=',ch,' succ=',succ(ch),' pred=',pred(ch),' ord=',ord(ch));
  Writeln(ch,' is the same as ',succ(pred(ch)));
  Writeln(ch,' is the same as ',chr(ord(ch)));
  Writeln('Type an integer');
  Readln(int);
  Writeln('int=',int,' succ=',succ(int),' pred=',pred(int),' ord=',ord(int));
  Writeln(int,' is the same as ',succ(pred(int)));
  Writeln(int,' is the same as ',ord(chr(int)));
  bool := TRUE;
  (* This contains an error *)
  Writeln('int=',int,' succ=',succ(int),' pred=',pred(int))
```

5 Using ordinals for the control variable the loop.

The main purpose of learning about ordinals was to be able to use them as control variables of a loop. Looping through the letters of the alphabet may be a natural way to solve some problem. Suppose we wanted to write a program to type out the letter A through Z. We could do this by writing the following program.

```
PROGRAM Alphabet(output);
VAR ch:char;
BEGIN
  FOR ch := 'A' TO 'Z' DO
    Write(ch);
  Writeln('.');
END.
```

Note: In this case the update rule, to keep track of where we are in the loop is simply `ch := succ(ch)`.

This is certainly simpler than

```
PROGRAM HardAlphabet(output);
VAR num:integer;
BEGIN
  FOR num := ord('A') TO ord('Z') DO
    Write(chr(num));
  Writeln('.');
END.
```

or even

```
PROGRAM TheGeekWay(output);
VAR num:integer;
BEGIN
  FOR num := 65 TO 90 DO
    Write(chr(num));
  Writeln('.');
END.
```

where one would have to remember the ASCII values for the characters.
5.1 Solving a problem using ordinals to drive a loop.

Let’s try a simple program to figure out the distance between two letters in the alphabet.

The solution to this problem is essentially,

- We read in a pair of letters.
- We iterate through a loop from the lower letter to the higher letter keeping track of how many times we’ve executed the loop.
- We then print the count.

The program is easy to write:

```pascal
PROGRAM AlphaDist(input,output);
VAR Counter:integer;
Control, Lower, Higher:character;
BEGIN
  Writeln('Type in two letters ');
  Readln(Lower,Higher);
  Count := 0;
  FOR Control := Lower TO Higher DO
    Count := Count + 1;
  Writeln('The distance is ',Count);
END.
```

The use of Count here is worth noting. When a variable is used like this it’s called a counter, or sometimes an accumulator. The accumulator builds up the result of the computation in the body of the loop without actually being involved in the control of the loop. Accumulators are easy to spot, as they are often different from the control variable and are initialized outside the loop.

There is of course a quicker way to do this without using a loop at all. Can you figure it out?

6 DOWNTO, or counting down through a loop.

We’ve seen loops that count up, how about loops that count down?

The keyword DOWNTO replaces the TO in the header of a FOR-DO loop when we wish to count downwards. Now FOR-DO loops have the following form:

```pascal
FOR controlVariable := highValue DOWNTO lowValue DO
  statement or compound statement
```

So when Nasa calls you up and asks you to write a Pascal program to launch the Space Shuttle you’d use a FOR-DO loop with the DOWNTO form as follows.

```pascal
PROGRAM Countdown(output);
VAR control:integer;
BEGIN
  FOR control := 10 DOWNTO 0 DO
    Write(control, ' ');
  Writeln('Lift off!');
END.
```

Just remember the initial value needs to be higher than the final value when you’re counting down.

7 Buffered input with the FOR-DO loop.

Sometimes processing input character by character is a method for solving a problem. Before we look at such a problem let’s see how to read in from the buffer character by character.

We can read in from a buffer using a loop. After we read in a sequence of characters from the keyboard we can use the Read() procedure.

Suppose we typed in the following sequence of characters.

```
  asdfg[CR]
```

the buffer would look like

```
  |a|s|d|f|g|[CR]
```

We can read in and print out the string using the program below.

```pascal
PROGRAM ReadBuf(input,output);
CONST NumInt := 5;
VAR K:integer;
Letter:character;
BEGIN
  FOR K := 1 TO NumInt DO
    BEGIN
      Read(Letter);
      Write(Letter);
    END (* For K *);
  Writeln;
END.
```
An example using a loop to read a buffer.

As you'll recall from the first lecture a byte is a series of 8 bits in memory. It is a binary number of length 8. Suppose we wanted to convert a byte into its decimal value, how would we do this?

A simple algorithm is take the first bit and multiply it by 128, then take the second bit and multiply it by 64, take the next bit and multiply it by 32, the next bit by 16, then 8, 4, 2 and finally 1. Add the sum of these products and you'll have the decimal form.

Our solution uses an accumulator, Value, to keep track of the value of the byte computed through the iterations of the loop, before we enter the loop for the first time it is set to zero.

We also have a variable bitVal which will decrease by half each time we pass through the loop. Initially it is 128, but decreases to 1 by the time we've completed 7 iterations. This gives us the values of each bit and how much we need to multiply 0 or 1 by before we add it to our accumulator.

So our pseudocode for the loop looks like this.

```
FOR each of the 8 bits downwards and DO
    BEGIN
        read the character signifying a bit in.
        convert the bit to the integer values 0 or 1.
        multiply the bit value by the current multiplier
        and accumulate the result.
        set the current multiplier for the next pass
        through the loop.
    END
```

Which when we set up our program exactly becomes.

```
PROGRAM ByteConverter(input,output);
VAR bitVal, thisbit, Value,Count:integer;
ch:character;
BEGIN
    (* Initialize values *)
    Value := 0; bitVal := 128;
    Writeln('Type in a byte');
    (* Loop through the 8 positions of a byte *)
    FOR Count := 7 DOWNTO 0 DO
        BEGIN
            read(ch); (* Read the character *)
            thisbit := ord(ch) - ord('0');
            Value := (bitVal * thisbit) + Value;
            bitVal := bitVal DIV 2;
        END;
    Writeln('The decimal value is ',Value)
END.
```

The lines

```
thisbit := ord(ch) - ord('0');
Value := (bitVal * thisbit) + Value;
```

are tricky. `ord(ch)` gives you the ordinal value of the character being read. If it's a '1' then the ordinal value is 49, if it's a '0' then it's 48. We convert the letter '1' to the value 1 by subtracting the ord('0') from ord('1'). If we typed in '1' we'd get 49 - 48 which is 1. If we'd typed in '0' we'd get 48 - 48 = 0.

We multiply the value of bitVal by thisbit to get the decimal value of that bit and add it to the value we've built up on previous iterations through the loop. Remember Value is initially zero.

The line

```
bitVal := bitVal DIV 2;
```

sets the value of bitVal to 128, 64, 32, 16, 8, 4, 2, 1 through the course of the iterations.

Reading in decimal numbers.

If we had to read in characters that represent numbers in base ten we'd still use the line

```
thisbit := ord(ch) - ord('0');
```

which for the character '7' would give thisbit a value of 7. The only thing we'd need to do is make sure that the value of hibyte in the calculation below starts as a power of ten and is divided by ten on every iteration

```
bitVal := bitVal DIV 10;
```

Extra credit!

Try this stuff and if you do well I'll shove it in your grade somehow.

**Question 1.** What would happen in the ByteConverter program if we'd input.

1. 00010001
2. 0001 0001
for your convenience I’ve listed the program again.

PROGRAM ByteConverter(input,output);
VAR bitVal, this-bit, Value, Count:integer;
    ch:character;
BEGIN
  (* Initialize values *)
  Value := 0; bitVal := 128;
  Writeln(‘Type in a byte’);

  (* Loop through the 8 positions of a byte *)
  FOR Count := 7 DOWNTO 0 DO
    BEGIN
      read(ch); (* Read the character *)
      thisbit := ord(ch) - ord(‘0’);
      Value := (bitVal * this-bit) + Value;
      bitVal := bitVal DIV 2;
    END (* FOR Count *)
  Writeln(‘The decimal value is ’,Value)
END.

Question 2. If you had written the ByteConverter in the following way would it work? If so why? If not why not, how would you fix it?

PROGRAM ByteConverter(input,output);
VAR bitVal, this-bit, Value, Count:integer;
    ch:character;
BEGIN
  (* Initialize values *)
  Value := 0; bitVal := 128;
  Writeln(‘Type in a byte’);

  (* Loop through the 8 positions of a byte *)
  FOR Count := 7 DOWNTO 0 DO
    BEGIN
      bitVal := bitVal DIV 2;
      read(ch); (* Read the character *)
      thisbit := ord(ch) - ord(‘0’);
      Value := (bitVal * this-bit) + Value;
    END (* FOR Count *)
  Writeln(‘The decimal value is ’,Value)
END.

Question 3.

IMPRESS ME! Can you write, using ONLY THE FEATURES OF PASCAL we’ve learned so far, a program that converts decimal values in the range 0 to 255, into it’s binary form? Use a loop!

9 Nested loops.

We’ve seen how to use a loop and how to use a loops limits to control the number of iterations performed. We can place one loop inside another, this is called nesting. This allows us to repeat more than one operation in a loop body. This is a very useful feature and many programs use it.

9.1 A simple nested loop.

Let’s say we want to print a blank form with spaces for appointments during the day. We want our appointment book to go from 9:00 to 5:00 with 15 minute slots. For simplicity we’ll use military time.

Our appointment form should look like:

```
9:00 | 9:15 | 9:30 | 9:45 
10:00 | 10:15 |
```

We see that the hours need to count from 9 to 16 and the minutes from 00 to 45 for every hour counted.

Our program needs to repeat, for every hour, the code for the minutes. This is where we will nest the second loop.

The program below should give us the output we wanted.

```
PROGRAM ApptBook(output);
VAR Hours, Index, Minutes :integer;
BEGIN
  FOR Hours := 9 TO 16 DO
    BEGIN
      Write(Hours:2,’ ‘);
      FOR Index := 0 TO 3 DO
        BEGIN
          Minutes := Rep * 15;
          Writeln(Minutes)
        END (* FOR Index *)
    END (* FOR Hours *)
END. (* PROGRAM *)
```

The inner loop is iterated fully for every iteration of the outer loop. So if there are M iterations of the outer loop and N iterations in the inner loop the statements in the inner loop will iterate M * N times.
9.2 Another simple nested loop program.

We can use nested loops to generate a multiplication table. A multiplication table has a set of indices on both its sides, the indices go from 0 - N and 0 - M respectively. A multiplication table should look like this:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>0</td>
<td>M</td>
<td>2M</td>
<td>3M</td>
<td>4M</td>
</tr>
</tbody>
</table>

The program to generate this table is simple.

```pascal
PROGRAM MultTable(input,output);
VAR M,N:integer;
Mul,I,J:integer;
BEGIN
Write('Please give me two integers and hit return');
Readln(M,N);
FOR I := 0 TO M DO
BEGIN
  FOR J := 0 TO N DO
  BEGIN
    Mul := I * J;
    Write(Mul:6);
  END (* FOR J *);
  Writeln('');
END (* FOR I *);
END.
```

The line number, or the present iteration number, that we’re on.

9.3 Using a loop to control another loops iteration.

Here is a somewhat more sophisticated idea. Suppose we want to draw the following triangle on a page.

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

We see that on line 1 we need to print out one '*', on line 2 we need to print out two '*'s etc... The amount of '*'s we print is dependent on the number of the line we’re currently on.

Setting up the program is simple have two loops, one that goes from one to six, and a second loop that goes from 1 to

```pascal
PROGRAM Triangle(output);
VAR i,j:integer;
BEGIN
  FOR i := 1 TO 6 DO
  BEGIN
    FOR j := 1 TO i DO
    BEGIN
      Write('*');
    END (* FOR j *);
    Writeln('');
  END (* FOR i *);
END.
```

This is what is important about the program above.

- The inner loop loops from $j = 1$ to $i$ where $i$ is set by the loop that surrounds it.
- The control of the enclosing loop acts as the limit for the control variable of the inner loop.
- The `Write('*')` was the only statement in the inner loop so we didn’t need to place it in a `BEGIN ... END` block.