Lesson 11: The Record Data Structure

In a nutshell, this lesson will cover:

What are Records

A record is a special type of data structure which, unlike arrays, collects a number of fields of different data types which define a particular structure such as a book, product, person and many others. The programmer defines the data structure under the ***Type*** user definition.

Let us see how we can define the properties of a book using a record data structure. Note that a record data structure gets close to the object entity used in ***Object Oriented Programming***. The properties of a book which we will be including are its title, author, unique ISBN number and its price. This is how it is defined in Pascal:

```
Type
         Str25    = String[25];
         TBookRec =
                            Record
                                     Title, Author, ISBN : Str25;
                                     Price : Real;
                            End;

Var
    myBookRec : TBookRec;
```

That's it! Note that I could have used this structure as a variable by declaring it under **Var**. When type-defining it, you can declare as many *TBookRec* variables as you wish.The entities of the record (title, author...) are called fields and these will be accessed through the instance of `myBookRec` variable. As you might have noticed, the declaration of typical record data structure starts with the keyword **`Record`** and always ends with the **`End`** keyword (note that there is no **`Begin`** keyword).

Now, you will see how we access the fields of the record by assigning them values and retrieve them back later in the following small program.

```
Type
         Str25    = String[25];
         TBookRec =
                  Record
                           Title, Author, ISBN : Str25;
                           Price : Real;
                  End;

Var
    myBookRec : TBookRec;

Begin
         myBookRec.Title  := 'Some Book';
         myBookRec.Author := 'Victor John Saliba';
         myBookRec.ISBN   := '0-12-345678-9';
         myBookRec.Price  := 25.5;

         Writeln('Here are the book details:');
         Writeln;
         Writeln('Title:  ', myBookRec.Title);
         Writeln('Author: ', myBookRec.Author);
         Writeln('ISBN:   ', myBookRec.ISBN);
         Writeln('Price:  ', myBookRec.Price);
         Readln;
End.
```

The 'With' Keyword

The `With` keyword is used in conjunction with recors to make it easier to access fields. Although it helps with quick field evaluation, it is rarely used since it has a lack of readability and increases ambiguity when coming to distinguish between a variable having the same name as one of the field names of a record within a **with** statement.

Taking a snippet from the previous example, this is how the code above transforms when incorprating the `With` keyword.

```
With myBookRec do
Begin
         Title  := 'Some Book';
         Author := 'Victor John Saliba';
         ISBN   := '0-12-345678-9';
         Price  := 25.5;
End;
```

Note how the fields in the Begin..End code block all belong to the Record `TBookRec` data structure.

Passing Records as Arguments

It may become very useful when records are required to be passed through arguments and this will be demonstrated shortly. I will use the same data structure, pass it by reference as a parameter and return the value back through the parameter also.

```
Type
         Str25    = String[25];
         TBookRec =
                            Record
                                     Title, Author, ISBN : Str25;
                                     Price : Real;
                            End;

Procedure EnterNewBook(var newBook : TBookRec);
Begin
         Writeln('Please enter the book details: ');
         Write('Book Name: ');
         Readln(newBook.Title);
         Write('Author: ');
         Readln(newBook.Author);
         Write('ISBN: ');
         Readln(newBook.ISBN);
         Write('Price: ');
         Readln(newBook.Price);
End;

Procedure DisplayBookDetails(myBookRec : TBookRec);
Begin
         Writeln('Here are the book details:');
         Writeln;
         Writeln('Title:  ', myBookRec.Title);
         Writeln('Author: ', myBookRec.Author);
         Writeln('ISBN:   ', myBookRec.ISBN);
         Writeln('Price:  ', myBookRec.Price);
End;

Var
         bookRec : TBookRec;

Begin
         EnterNewBook(bookRec);
         Writeln('Thanks for entering the book details');
         DisplayBookDetails(bookRec);
         Readln;
End.
```

Arrays of Records

Records may be stored in arrays and this will become very useful and it is not that difficult to manage. We will use an array of records to store a number of different books and by using this example, it will be immensely indicative to learn how to use them.

In the following example I will use the procedures above to store 10 different books from input and then output only one chosen record to display it back to screen in order to demonstrate how to access a record from an array.

```
Type
         Str25    = String[25];
         TBookRec =
                            Record
                                     Title, Author, ISBN : Str25;
                                     Price : Real;
                            End;

Procedure EnterNewBook(var newBook : TBookRec);
Begin
         Writeln('Please enter the book details: ');
         Write('Book Name: ');
         Readln(newBook.Title);
         Write('Author: ');
         Readln(newBook.Author);
         Write('ISBN: ');
         Readln(newBook.ISBN);
         Write('Price: ');
         Readln(newBook.Price);
End;

Var
         bookRecArray : Array[1..10] of TBookRec;
         i            : 1..10;

Begin
         For i := 1 to 10 do
                  EnterNewBook(bookRecArray[i]);

         Writeln('Thanks for entering the book details');
         Write('Now choose a record to display from 1 to 10: ');
         Readln(i);
         Writeln('Here are the book details of record #',i,':');
         Writeln;
         Writeln('Title:  ', bookRecArray[i].Title);
         Writeln('Author: ', bookRecArray[i].Author);
         Writeln('ISBN:   ', bookRecArray[i].ISBN);
         Writeln('Price:  ', bookRecArray[i].Price);
         Readln;
End.
```

Note that you can also use arrays of records. Be careful how to place the square brackets of the array. Also you can embed records within records. Dot operator will be required to access deeper records.

Binary Files and Records

Records can also be stored into files and this could be done by using binary files. I will demonstrate storing records into files by continuing from the previous example. Using binary files could be very handy, fast and more reliable over text files. You can't afford storing hundreths of files by using text files since it becomes confusing and even slower for the computer to process and read/write from/to the file.

In the following example I will use a file of the book record I have created and then store as many books as I want in the file using the binary file system. Watch carefully how I will create the file of record and how I will perform the file I/O for the binary file system. Also, I will make use of special built in functions that help me position the file pointer to the record I want.

Note that with binary files, only Read and Write are allowed to read/write from/to a file.

```
Type
         Str25     = String[25];
         TBookRec =
                           Record
                                    Title, Author, ISBN : Str25;
                                    Price : Real;
                           End;

Procedure EnterNewBook(var newBook : TBookRec);
Begin
         Writeln('Please enter the book details: ');
         Write('Book Name: ');
         Readln(newBook.Title);
         Write('Author: ');
         Readln(newBook.Author);
         Write('ISBN: ');
         Readln(newBook.ISBN);
         Write('Price: ');
         Readln(newBook.Price);
End;

Var
         bookRecArray : Array[1..10] of TBookRec;
         tempBookRec  : TBookRec;
         bookRecFile  : File of TBookRec;
         i            : 1..10;

Begin
         Assign(bookRecFile, 'bookrec.dat');
         ReWrite(bookRecFile);

         For i := 1 to 10 do
         Begin
                  EnterNewBook(bookRecArray[i]);
                  { bookRecArray[i] now contains the book details }
                  Write(bookRecFile, bookRecArray[i]);
         End;

         Close(bookRecFile);
         Writeln('Thanks for entering the book details.');
         Writeln('They are saved in a file!');
         Write('Now choose a record to display from 1 to 10: ');
         Readln(i);
         ReSet(bookRecFile);
         Seek(bookRecFile, i-1);
         Read(bookRecFile, tempBookRec);
         Close(bookRecFile);
         Writeln('Here are the book details of record #',i,':');
         Writeln;
         Writeln('Title:  ', tempBookRec.Title);
         Writeln('Author: ', tempBookRec.Author);
         Writeln('ISBN:   ', tempBookRec.ISBN);
         Writeln('Price:  ', tempBookRec.Price);
         Readln;
End.
```

The example program above demonstrated the use of the seek function. It's role is to place the file pointer to the desired position. The first component of the file is marked as 0. So you have to keep in mind that if you have a counter starting from 1, you have to decrement it by 1 to obtain the actual record you want.

The seek function is very important and has an important role in binary file system. Here are some uses of the function and how it can be used effectively to obtain a particular position of the file.

Special Uses of the *Seek* Function

Seek the first record of the file

```
Seek(myFile, 0);
```

Seek the last record of the file

```
Seek(myFile, FileSize(myFile)-1);
```

Seek one position beyond the last record

```
Seek(myFile, FileSize(myFile));
```

Seek to the next position of the file from the current

```
Seek(myFile, FilePos(myFile)+1);
```

When trying to access from a file position that is beyonf the file limits, a runtime error is automatically raised. Try to avoid this type of error. This may be caused because you might have looped through the file and kept on looping beyond its limits. Note that Seek(myFile, -1) is a typical runtime error becuase -1 position does not exist. 0 is the least and the first record in the file. Note that ***FilePos*** is also very useful and it returns the current positon of the file. Please note that ***FileSize*** returns the number of components in the specified file and not the size in Bytes. If the file is empty, 0 is the returned value. On the other hand, if the file contains 5 records (ie. 0 to 4), 5 is returned.

The structure of a binary file is just like blocks being stored contiguosly in a line. Think of boxes being placed one adjacent the other and each one of them has data. There is a space between this boxes that indicates the file positon and we can easily depict this fact below.

0	**BookRec0**	1	**BookRec1**	2	**BookRec2**	3	**BookRec3**	4	**BookRec4**
		^							

The first row is the actual file showing the indexes of each record block and the second row shows the file pointer ie. the file position. The current file position shown in the illustration above is relevant to ***Seek(myFile, 1)***. Now you have been assured that the number 1 record of a file is not the first record of the file. After the last record, there is an EOF marker that indicates the end of the file and it is not legal to go beyond this point except for only one position to allow appending ie. ***Seek(myFile, FileSize(myFile))***.