Lesson 14: Article 2: The Stack Data Structure

An array is a data structure which stores data items in a pre-allocated space. It allows you to store data anywhere within its range. On the other hand, the Stack Data Structure does not allow you to store data where you want i.e. order is important. It is based on a LIFO basis - the last to arrive, the first to be served.

The stack is a simple array which stores data items using an index which points to the last element that has been inserted. If data is requested from the stack, the last element that has been stored is 'poped' out of the array and returned. When popping occurs, the last element is returned and discarded from the stack with the top pointer being decremented by 1. Obviously, if the stack is empty, nothing is returned and the top pointer will remain untouched.

One can always add elements to the stack by pushin data items at the end of the stack indicated by the top pointer. After each added item, the top pointer is increased by 1, however items won't be added and top pointer won't be increased by 1 if the stack is already full. The stack rejects any more items to be added when full until at least 1 item is poped.

Functions involved in operating the Stack Data Structure

The fundamental operations performed when working with stacks are listed below.

[Assuming *DataItem* is a pre-declared user data type. It maybe anything like an integer, a string or any other data object.]

Function	**Description**
Procedure InitStack;	Initialises the stack data structure.
Function IsEmpty : **Boolean**;	Returns true if the stack is empty.
Function IsFull : **Boolean**;	Returns true if the stack is full.
Function Pop : **DataItem**;	If list is empty nothing is returned and top pointer remains the same. Otherwise, it acquires the last data item from the list and returns it. The top pointer is decreased by 1.
Procedure Push(item : **DataItem**);	If the stack is not full, the item passed as argument is added to the end of the list indicated by the top pointer and latter is increased by 1.
Function GetSize : **Integer**;	Returns the top pointer.

Implementation

An implementation of the fundamental functions are demonstrated below.
First I'll show you how to declare a stack data structure using an array and its top pointer. These will be global variables in your program as long as you decide to use them specifically in some procedure or function.

```
Const
          STACK_SIZE = 100;


Var
          myStack : Array[1..STACK_SIZE] of DataItem;
          topPointer : Integer;
```

The initialisation is to be called before any stack operation.

```
Procedure InitStack;
Begin
          topPointer := 0;
End;
```

We now implemement the *IsEmpty()* and *IsFull()* functions.

```
Function IsEmpty : Boolean;
Begin
          IsEmpty := False;
          If (topPointer = 0) Then
                    IsEmpty := true;
End;
```

```
Function IsFull : Boolean;
Begin
          IsFull := False;
          If ((topPointer + 1) = STACK_SIZE) Then
                    IsFull := True;
End;
```

Here are the implementations of the Pop() and Push() functions and making use of the functions that we have just implemented.

```
Function Pop : DataItem;

Begin
          Pop := nil;

          If not IsEmpty Then
          Begin
                    Pop := myStack[topPointer];
                    topPointer := topPointer - 1;
          End;
End;
```

```
Procedure Push(item : DataItem);
Begin
          If not IsFull Then
          Begin
                    myStack[topPointer+1] := item;
                    topPointer := topPointer + 1;
          End;
End;
```

Finally, we implement the utility function *GetSize()*. Although one can access the current size of the stack using the global variable topPointer, it is of good practice to make use of functions instead of global variables.

```
Function GetSize : Integer;
Begin
          GetSize := topPointer;
End;
```

That's the story of implementing a Stack Data Structure. It is a very useful utility that is not provided ready made with a programming language. A stack has wide range of uses such as in parsing a command line calculator - matching brackets. The stack is also used inevitably by your operating system in executing programs. When a function is called, the location in memory of the embedding function is pushed onto the stack for later reference, so that the operating system 'remembers' where it has broken execution of the previous function. When the function continues execution, it is poped off the stack.