Stacks

- The first data structure we will deal with is a *stack*. Physically, a stack is nothing but a group of data. However, the operations on a stack can be done only in a specific order. Any new item to be added to the stack must be added at the top, and any item to be removed from the stack must be removed from the top.

- Consider a Pringles box. If you want to eat a piece of chips, which one can you take? The one on the top of the box. If you want to take a chips other than the one on the top, you must first of all take all the chips on top of that one out of the box, and put them back to the box, after taking the one you wish. If you want to put one more chips into the box, you must put it on the top.

- This example simulates very well how a stack works. A stack works in a Last In First Out (LIFO) strategy. The lastly inserted item should be taken first out of the stack.

- Consider another example. Assume that next to your house you have a very narrow area which is used as the parking lot. It is so narrow that, only one car can enter or leave that area at once. Thus, the cars must be parked one after another in a line. Which car can leave the parking lot without disturbing anybody? The last one that entered into the lot (LIFO). If any other car wants to leave the lot, all the other cars between that car and the entrance of the lot must be taken out one by one, and after that car leaves, they should again be taken in, in the reverse order they are taken out. Such a parking lot is another good example of a stack.

Definition and Declaration of Stacks

- A stack can be defined as a structure with two members:
    - an array to hold the data items
    - an integer value which shows the position of the top element

Example:

```
typedef struct
{
     int top;
     int data[50];
} stack_t;
...
stack_t s;
```

- `s` is a stack to store 50 integer numbers. Those numbers will be stored in the array `s.data`. `s.top` will contain the position (index) of the last element of `s.data` containing actual data.

- The data items in the array may be of any data type, such as integer, double, character, string. They may even be of a structure type, depending on what kind of data we want to store in our stack.

Example: Define a stack type and declare a stack to contain the names of at most 100 customers.

```
typedef struct
{
     int top;
     char data[100][40];
} name_stack_t;
...
name_stack_t customer;
```

- `customer` is a stack to store 100 strings corresponding to customer names. Those names will be stored in the rows of the array `customer.data`. `customer.top` will contain the position (index) of the last row of `customer.data` containing actual data.

Example: Define a stack type and declare a stack to contain the ID, name, registration date and department of at most 500 students.

- Our data is complex. It is easier if we define its structure first:

```
typedef struct
{
     int day, month, year;
} date_t;

typedef struct
{
     int  id;
     char name[40],
          dept[20];
     date_t regist_date;
} student_t;

typedef struct
{
     int top;
     student_t data[500];
} std_stack_t;
...
std_stack_t std;
```

- Notice that `s`, `customer`, and `std` are not arrays. Each one contains an array in itself, but they are not arrays, they are structures representing one stack.

- If we want to keep information about the students in three sections, in three separate stacks, we need to declare three stack variables as:

  ```
  std_stack_t sec1, sec2, sec3;
  ```

  or an array with three elements of the `std_stack_t` type as:

  ```
  std_stack_t sec[3];
  ```

- How can we reach to the registration year of the last student of the first section?

  ```
  sec[0].data[sec[0].top].regist_date.year
  ```

Stack Operations

- There are five main stack operations:

  1. Initialization of an empty stack
  2. Checking if the stack is empty
  3. Checking if the stack is full
  4. Inserting a new item onto the stack (PUSH)
  5. Removing an item from the stack (POP)

- Inserting a new data item onto a stack is called as *pushing*. Removing an item from the stack is called as *popping*. Both of these operations are based on knowing the value of `top`. Push operation will increment the `top`, and insert the new item into the top position. Similarly, Pop operation will remove the top element, and decrement the `top`. Push must check if the stack is full or not, before trying to insert the item, and pop must check whether the stack is empty or not, before trying to remove an item. Therefore, before defining the functions for push and pop operations, we need to define functions for the other operations.

- Assume that we have already made the following definitions:

  ```
  #define STACK_SIZE 20
  #define STACK_EMPTY '#'
  typedef struct
  {
       int  top;
       char data[STACK_SIZE];
  } stack_t;
  ```

- Initialization of an empty stack means initializing its top to -1, so that when push inserts the first data item onto the stack, it will become 0, representing the index of that item. Hence the function is so simple as:

```
/* sets top to -1 */
void initialize_s(stack_t *s)
{
     s->top = -1;
}
```

- Since the stack will change after the operation, it must be passed by reference (as a pointer).
- Checking if the stack is empty is very easy. We just need to check if its top is -1. Hence the function is

```
/* Checks if the stack is empty */
/* If top = -1, the stack is empty */
int is_empty_s(stack_t *s)
{
     if (s->top == -1)
         return 1;
     return 0;
}
```

- If the top of the stack shows the last element of the data array, it means that the stack is full. Therefore, to check if the stack is full, we just need to check if its top is equal to `STACK_SIZE - 1`. You may pass the size as a parameter, or assume that it is a global constant, as in our example. Hence the function is

```
/* Checks if stack is full */
/* If top reached STACK_SIZE-1, then the stack is full*/
int is_full_s(stack_t *s)
{
     if (s->top == STACK_SIZE - 1)
         return 1;
     return 0;
}
```

- Notice that, in the previous two functions although we would not perform any changes on the stack we used call by reference. Why?
- Now, we are ready to write functions for push and pop operations. As I said before, push operation will increment the `top`, and insert a new item into the top position. Therefore, `push` function needs the stack, and the item to be pushed as parameters. Since the stack will change after the operation, it must be passed by reference. The data type of the item must be the same as the type of the `data` array in the stack. To be able to insert the new item onto the stack, `push` must check if the stack is full or not. If the stack is full, it can give an error message. The function is:

```
/* Push function first checks if stack is full using is_full_s
   function. If full, a "FULL" message is printed, otherwise
   stack's top is incremented and the new item is put
   in the new top position of stack */
void push(stack_t *s, char item)
{
     if (is_full_s(s))
         printf("Error: Stack is full!\n");
     else
     {
         (s->top)++;
         s->data[s->top] = item;
     }
}
```

- Pop operation will remove the top element, and decrement the `top`. Therefore, `pop` function needs the stack as parameter. Since the stack will change after the operation, it must be passed by reference. The function will return the item at the top of the stack. Thus, the function's data type must be the same as the type of the `data` array in the stack. The function must check if the stack is empty or not. If the stack is empty, it can give an error message. The function is:

```
/* Pop function first checks if stack is empty. If empty,
   "EMPTY" message is printed, otherwise, the item at
   the top of the stack is put in a variable and the top
   of the stack is decremented */
char pop(stack_t *s)
{
     char item;
     if (is_empty_s(s))
     {
         printf("Error: Stack is empty!\n");
         item = STACK_EMPTY;
     }
     else
     {
         item = s->data[s->top];
         (s->top)--;
     }
     return (item);
}
```

- Since the above function must return a character in any situation, but there are no characters in the stack when it is empty, we assigned `item` to a constant `STACK_EMPTY` representing that no item could be popped. The value of that constant should be choosen as a value which does not appear within the stack.

- Since we finished to define the functions for the basic stack operations, now we can use them in some examples.

Example: Given 5 characters, put them into a stack.

- First of all, we must initialize our stack as empty. Then, we need to read the input data using a loop. We will read each character into a variable, then push this variable onto the stack:

```
char ch;
int k;
stack_t s;
initialize_s(&s);
for (k = 1; k <= 5; k++)
{
     ch = getchar();
     push(&s, ch);
}
```

- Assuming our input data is "**KONYA**", trace and show the stack representation in the memory.

Example: Output the data in the stack.

- When outputting, there is an important decision to make: Do you want the data to remain in the stack, or do you want to empty the stack as you output? If you want to empty the stack each time you output the value, you need to pop it, and don't forget that this value will be lost after outputting.

```
while (!is_empty_s(s))
{
     ch = pop(&s);
     putchar(ch);
}
```

- After the loop terminates, what will you see on the screen? Trace and show the stack representation in the memory.

- Notice that, the output is just in the reverse order of the input data.

- Since we popped the stack and then output the value, don't forget that the stack is empty after the loop terminates. Thus, we destroyed the stack.

- If we don't want to destroy the stack, we can define the above loop as a function, using the stack as a value parameter, so that, when we call the function the actual stack will not be affected from the changes done in the function.

```
void display_s(stack_t s)
{
     while (!is_empty_s(s))
         putchar(pop(&s));
     printf("\n");
}
...
display_s(s);
```

- Notice that, stacks are very suitable if you want to reverse something, because the items inserted in a stack can only be removed in the reverse order.