Stacks

The stack is a very common data structure used in programs. Stacks are useful in situations requiring the retention and recall of information in a specific order—namely, when information is removed in a last-in, first-out order (LIFO for short). Removing tennis balls from a can is LIFO. In programming, stacks are useful when a program must postpone obligations, which must later be fulfilled in reverse order from that in which they were incurred.

Information is kept in order in a stack. Additional information may be inserted on the stack, but only at the front or top end. Thus the information at the top of the stack always corresponds to the most recently incurred obligation. Information may be recalled from the stack, but only by removing the current top piece of information. The stack acts like a pile of playing cards to which we can only add a card at the top and from which we can only remove the top card.

The stack, with the operations of front-end insertion and deletion, initialization, and an empty check, is a data abstraction. Stacks used to be called "pushdown stacks"; insertion of data onto the stack was referred to as "pushing" the stack, and deleting data was known as "popping" the stack. So far we have written programs that treat the stack with the four operations as a data abstraction. Stacks will be used frequently in algorithms developed throughout the rest of this book, and it is now time to consider this implementation. Of the many implementations possible for the stack, the following subsections illustrate the three most straightforward.

![Stack of coins, Stack of books, Computer stack](image)

**FIGURE 3-1** Stack
Here we present the idea of stack implementation, based on arrays. We assume that stack's capacity is limited to a certain value and overfilling the stack will cause an error. Though, using the ideas from dynamic arrays implementation, this limitation can be easily avoided.
In spite of capacity limitation, array-based implementation is widely applied in practice. In a number of cases, required stack capacity is known in advance and allocated space exactly satisfies the requirements of a particular task. In other cases, stack's capacity is just intended to be "big enough".

Implementation

Implementation of array-based stack is very simple. It uses top variable to point to the topmost stack's element in the array.

```c
struct stack  {
    int s[size]; //array to hold stack elements
    int top; //variable to point the top of stack
}st;
```

Source code to implement stack using arrays and some of its operations:

```c
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define size 5
/* stack structure*/
struct stack  {
    int s[size];
    int top;
}st;

int stfull() {
    if(st.top>=size-1)
        return 1;
    else
        return 0;
}

void push(int item) {
    st.top++;
    st.s[st.top] =item;
}

int stempty() {
    if(st.top==-1)
        return 1;
    else
```
return 0;
);
//-------------------------------------------------------
int pop()
{
    int item;
    item=st.s[st.top];
    st.top--;
    return(item);
}
//-------------------------------------------------------
void display()
{
    int i;
    if(stempty())
        printf("n Stack Is Empty!");
    else
    {
        for(i=st.top;i>=0;i--)
            printf("n%d",st.s[i]);
    }
}
//-------------------------------------------------------
void main(void)
{
    int item,choice;
    char ans;
    st.top=-1;
    clrscr();
    printf("n Implementation Of Stack");
    do
    {
        printf("n Main Menu");
        printf("n Enter Your Choice");
        scanf("%d",&choice);
        switch(choice)
        {
        case 1:
            printf("n Enter The item to be pushed");
            scanf("%d",&item);
            if(stfull())
                printf("n Stack is Full!");
            else
                push(item);
            break;
        case 2:
if(stempty())
    printf("n Empty stack!Underflow !!");
else
    {
    item=pop();
    printf("n The popped element is \%d",item);
    }
    break;
case 3:
    display();
    break;
case 4:
    exit(0);
}
printf("n Do You want To Continue?");
ans=getche();
while(ans == 'Y' || ans == 'y');
getch();
}

Explanation:

Step 1 : Declare One Stack Structure

#define size 5

struct stack
{
    int s[size];
    int top;
}st;

1. We have created ‘stack’ structure.
2. We have array of elements having size ‘size’
3. To keep track of Topmost element we have declared top as structure member.
```c
void push(int item) {
    st.top++;
    st.s[st.top] = item;
}
```

**Step 2: Push/Pop Operation**

While pushing remember one thing in mind that we are incrementing the top and then adding element. and while removing or poping the element, we are firstly removing the element and then decrementing the top.

```c
int pop() {
    int item;
    item = st.s[st.top];
    st.top--;
    return (item);
}
```