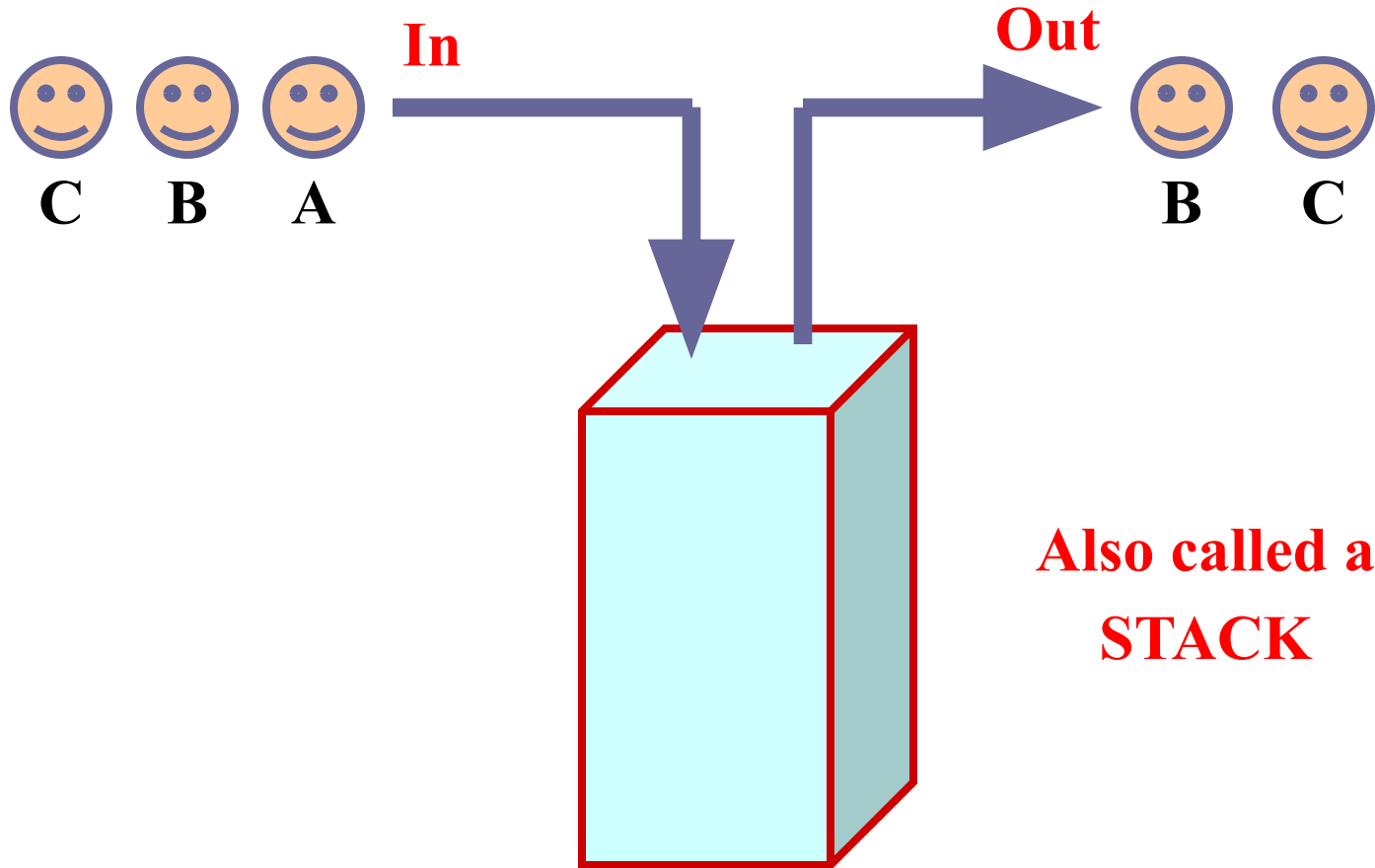
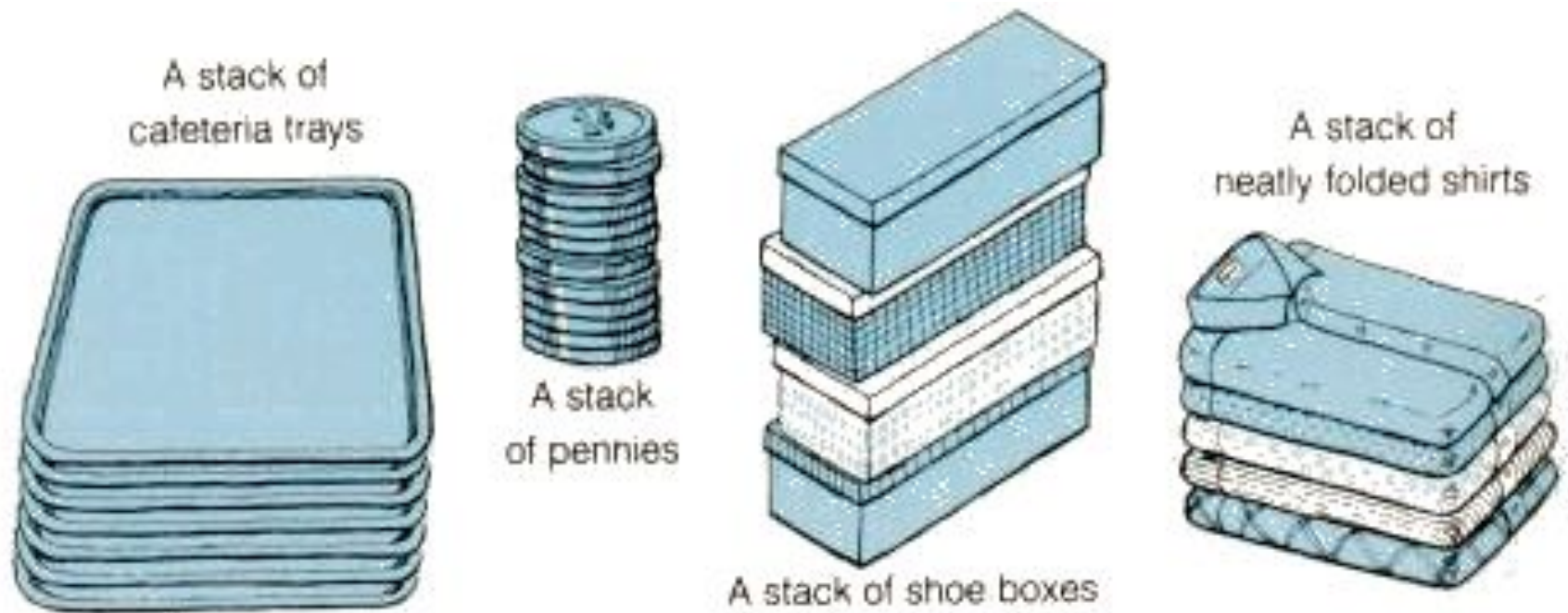


**Stack**

# A Last-in First-out (LIFO) List

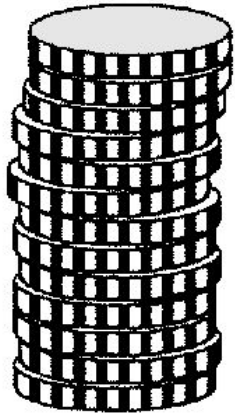


# Stacks in Our Life

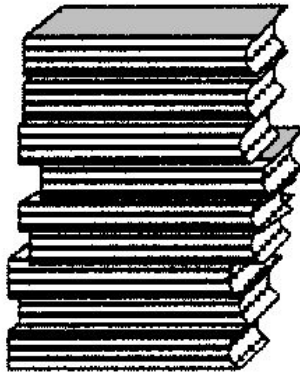


- A stack is an Abstract Data Type (ADT), commonly used in most programming languages.

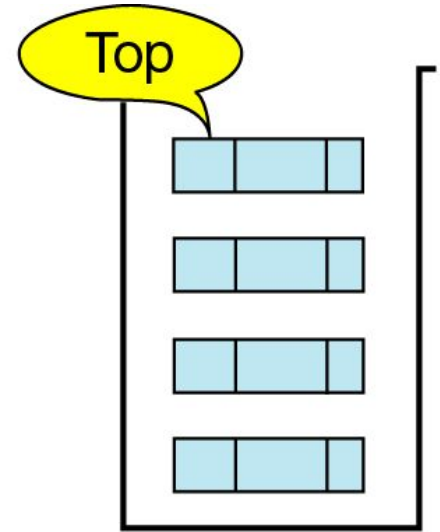
# More Stacks



Stack of coins



Stack of books

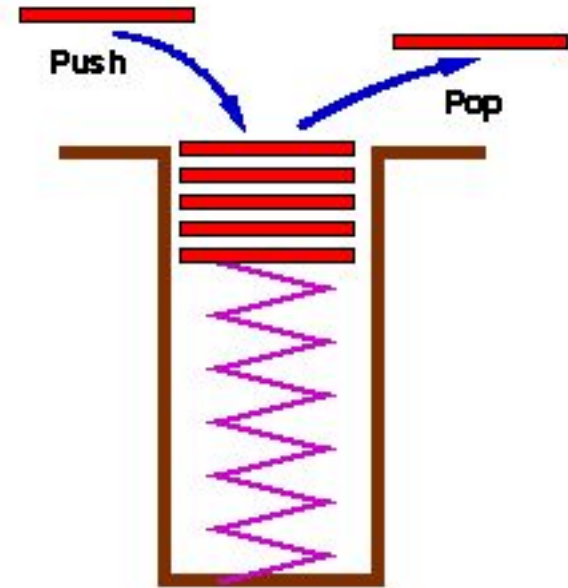


Computer stack

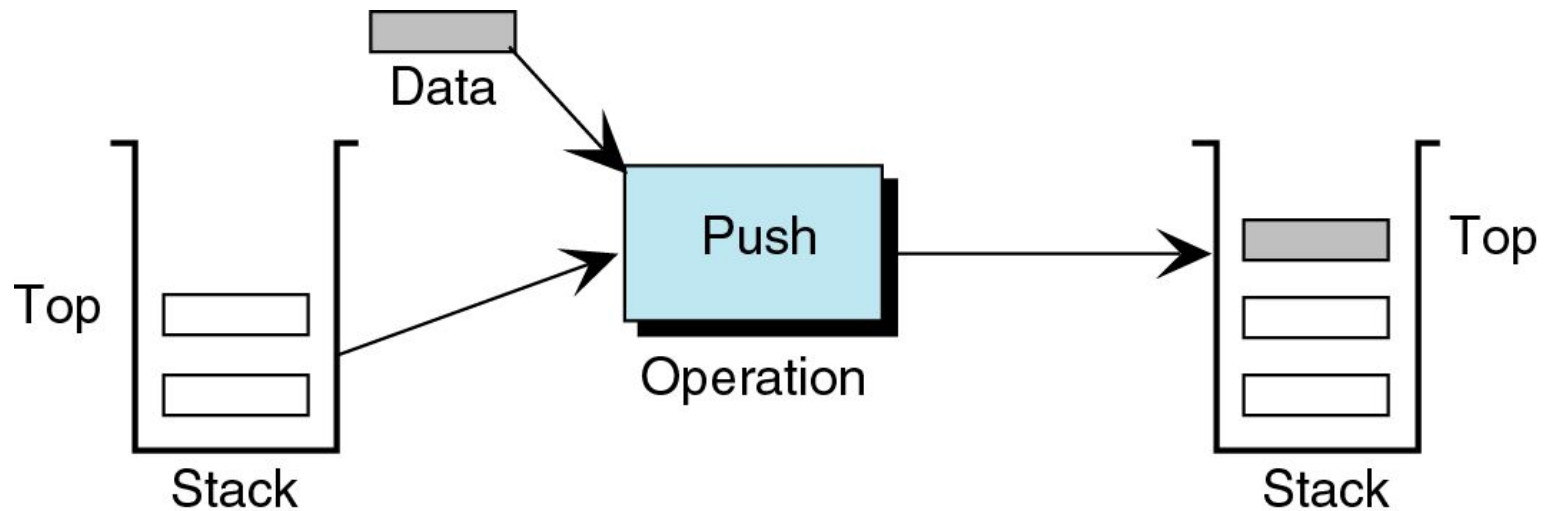
- A stack is a LIFO structure: Last In First Out

# Basic Operations with Stacks

- Push
  - Add an item
    - Overflow
- Pop
  - Remove an item
    - Underflow
- Stack Top
  - What's on the Top
    - Could be empty

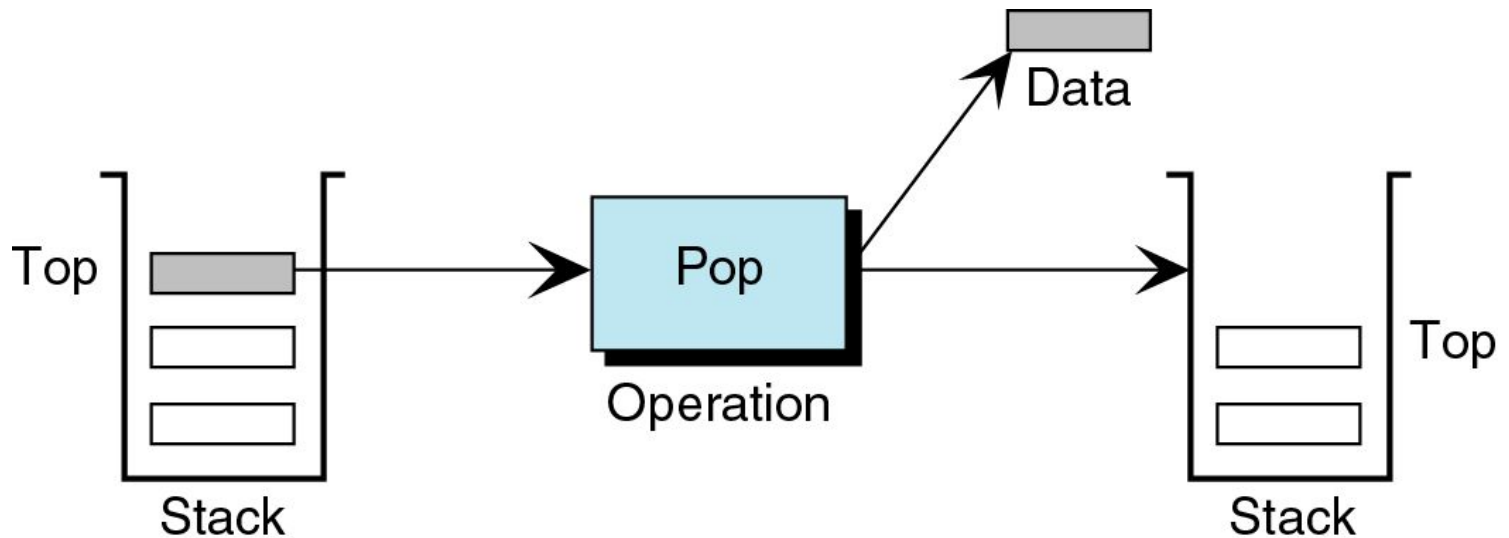


# Push



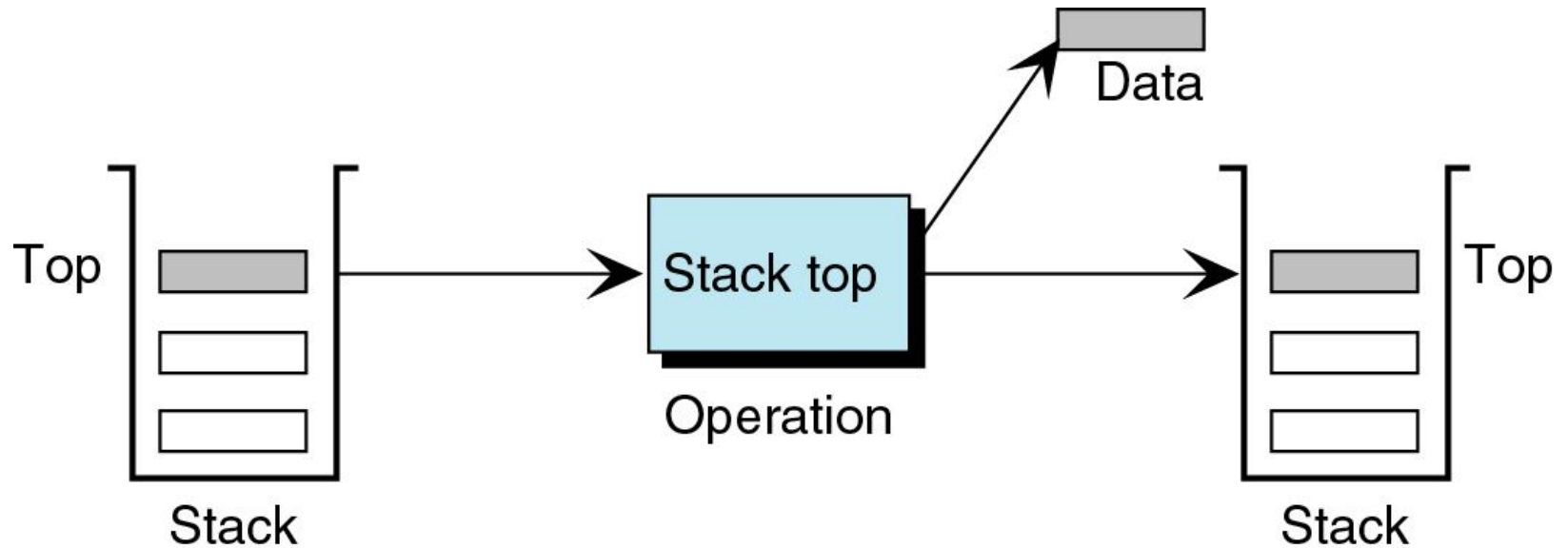
- Adds new data element to the top of the stack

# Pop



- Removes a data element from the top of the stack

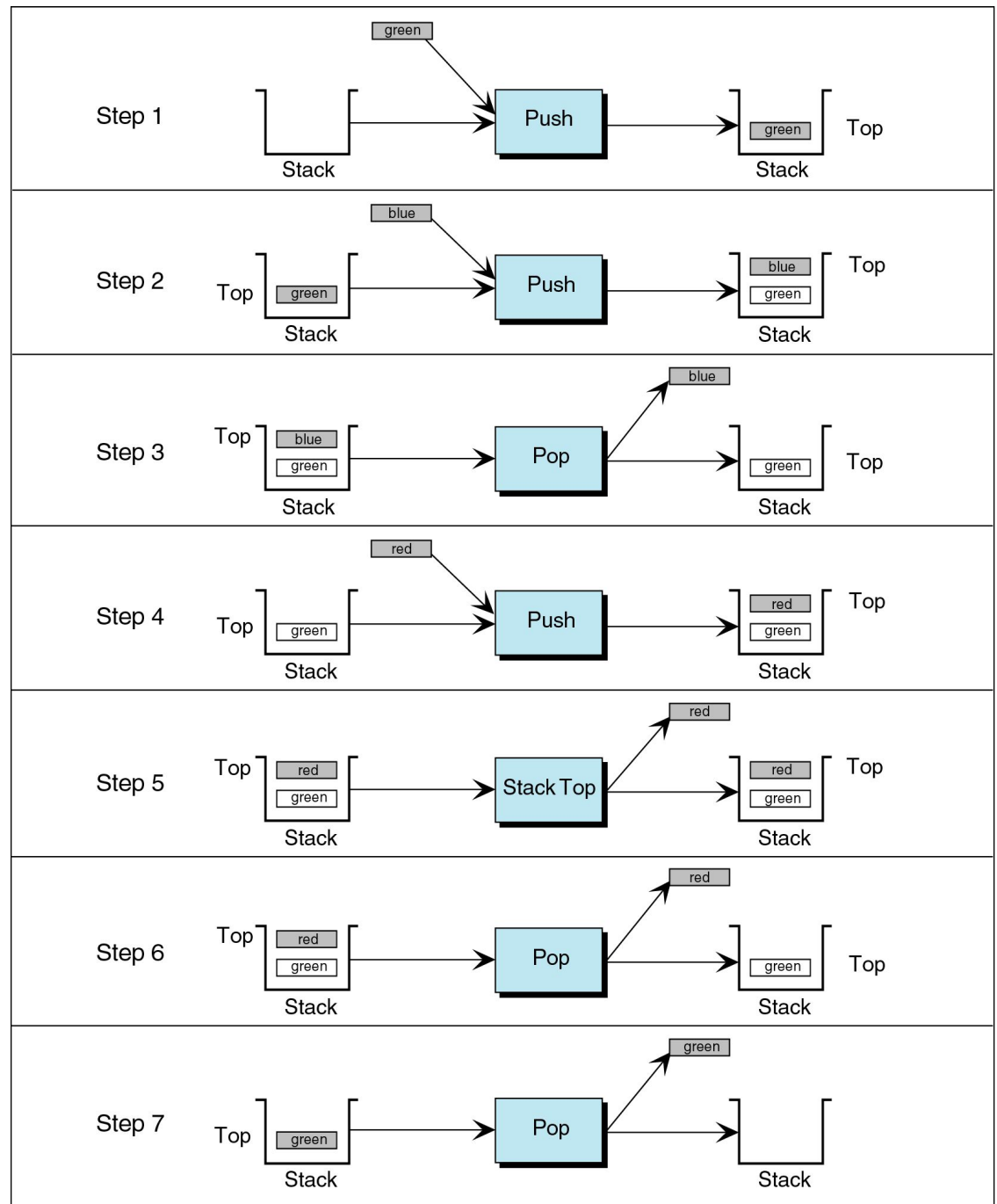
# Stack Top

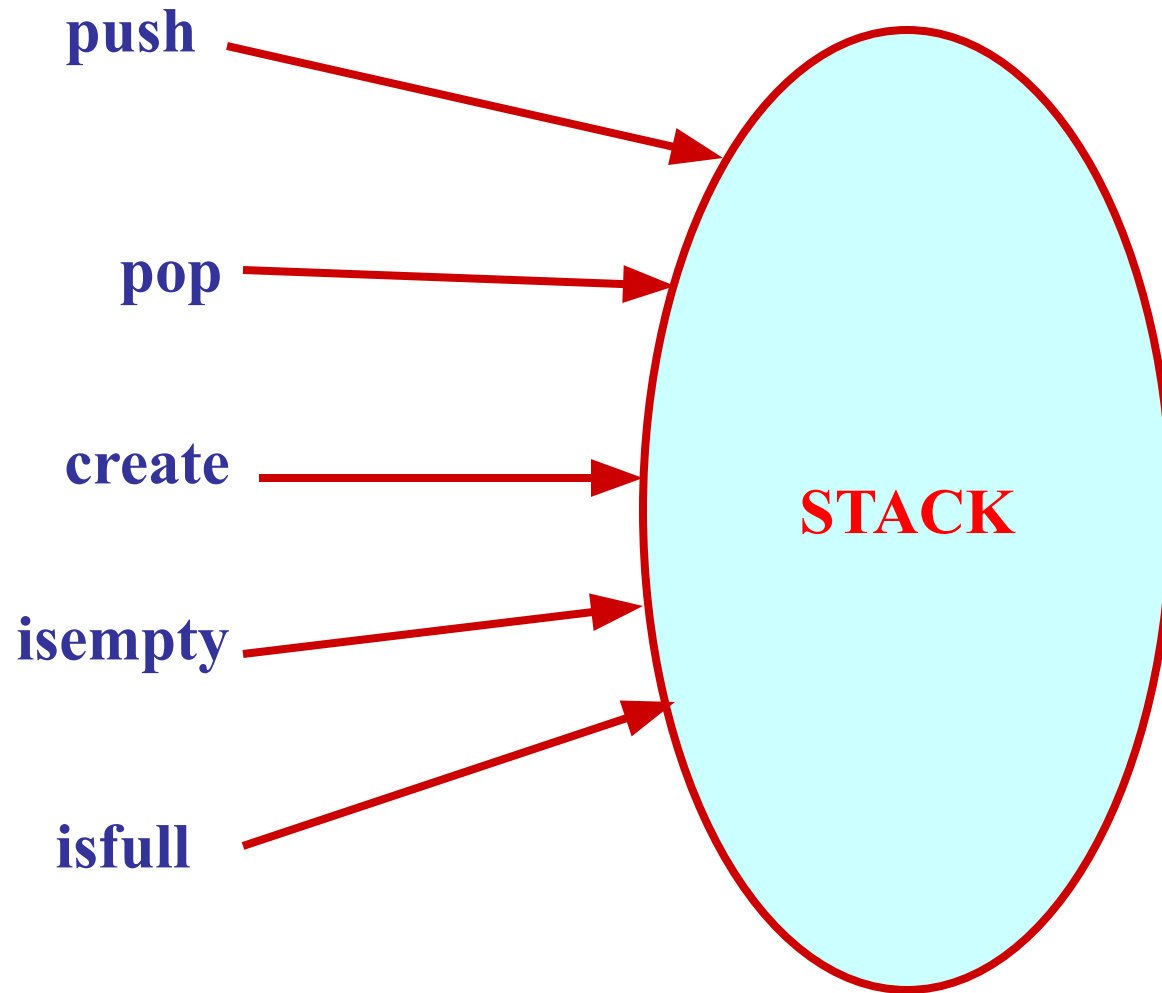


- Checks the top element. Stack is not changed



When an element is “pushed” onto the stack, it becomes the first item that will be “popped” out of the stack. In order to reach the oldest entered item, you must pop all the previous items.





# Assume:: stack (LIFO) contains integer elements

```
void push (stack s, int element);
```

```
/* Insert an element in the stack */
```

```
int pop (stack s);
```

```
/* Remove and return the top element */
```

```
void create (stack s);
```

```
/* Create a new stack */
```

```
int isempty (stack s);
```

```
/* Check if stack is empty */
```

```
int isfull (stack s);
```

```
/* Check if stack is full */
```

- We shall look into two different implementations of stack:
  - Using **arrays** (static implementation)
  - Using **linked list** (dynamic implementation)

# Stack Implementation

# Stack Implementation Using Arrays

- Basic idea.
  - Declare an array of fixed size (which determines the maximum size of the stack).
  - Keep a variable which always points to the “**top**” of the stack.

# Underlying Mechanics of Stacks

Initially, a pointer (`top`) is set to keep the track of the topmost item in the stack. The stack is initialized to `-1`.

Then, a check is performed to determine if the stack is empty by comparing `top` to `-1`.

As elements are added to the stack, the position of `top` is updated.

As soon as elements are popped or deleted, the topmost element is removed and the position of `top` is updated.

This sample program presents the user with four options:

1. Push the element
2. Pop the element
3. Show
4. End

It waits for the user to input a number.

- If the user selects 1, the program handles a `push()`. First, it checks to see if `top` is equivalent to `SIZE - 1`. If true, "Overflow!!" is displayed. Otherwise, the user is asked to provide the new element to add to the stack.
- If the user selects 2, the program handles a `pop()`. First, it checks to see if `top` is equivalent to `-1`. If true, "Underflow!!" is displayed. Otherwise, the topmost element is removed and the program outputs the resulting stack.
- If the user selects 3, the program handles a `show()`. First, it checks to see if `top` is equivalent to `-1`. If true, "Underflow!!" is displayed. Otherwise, the program outputs the resulting stack.
- If the user selects 4, the program exits.



# Declaration

```
#define MAXSIZE 100
```

```
struct lifo {  
    int st[MAXSIZE];  
    int top;  
};
```

```
typedef struct lifo stack;
```

# Stack Creation

```
void create (stack s)
{
    s.top = 0;    /* Points to last element
pushed in */
}
```

# Pushing an element into the stack

```
void push (stack s, int element)
{
    if (s.top == (MAXSIZE-1))
    {
        printf ("\n Stack overflow");
        break;
    }
    else
    {
        s.top ++;
        s.st [s.top] = element;
    }
}
```

# Removing/Popping an element from the stack

```
int pop (stack s)
{
    if (s.top == 0)
    {
        printf ("\n Stack underflow");
        break;
    }
    else
    {
        return (s.st [s.top --]);
    }
}
```

## Checking for stack full / empty

```
int  isempty (stack s)
{
    if (s.top == 0)  return 1;
    else  return (0);
}
```

```
int  isfull (stack s)
{
    if (s.top == (MAXSIZE - 1))  return 1;
    else  return (0);
}
```

# Structure and Memory Allocation in C

# Concept of Structure

Structure in C is a user-defined data type that enables us to store the collection of different data types. Each element of a structure is called a member. The **struct** keyword is used to define the structure.

## Syntax of Structure:

```
struct structure_name
```

```
{
```

```
    data_type
```

```
    member1;
```

```
    data_type
```

```
    member2;
```

```
    .
```

```
    .
```

```
    data_type memberN;
```

```
};
```

```
struct employee
```

```
{
```

```
    int id;
```

```
    char name[20];
```

```
    float salary;
```

```
};
```

## Declaring structure variable

We can declare a variable for the structure so that we can access the member of the structure easily. There are two ways to declare structure variable:

1. By **struct** keyword within main() function
2. By declaring a variable at the time of defining the structure.



To declare the structure variable by struct keyword, it should be declared within the main function.

```
struct
```

```
employee
```

```
{ int id;
```

```
  char name[50];
```

```
  float salary;
```

```
};
```

```
void main()
```

```
{
```

```
  struct employee
```

```
  e1,e2;
```

```
}
```

To declare variable at the time of defining the structure.

```
struct employee  
{ int id;  
  char name[50];  
  float salary;  
}e1,e2;
```

**Accessing members of the structure**

By **.** (member or dot operator)

Eg. e1.id=101

## C program to print Student information using structure

```
#include<stdio.h>
#include <string.h> struct
student
{   int id;
    char name[50]; char
    branch[50];
};
void main()
{   struct student    st1;
    scanf("%d %s %s",&st1.id ,st1.name,st1.branch);
    printf("%d %s %s",st1.id ,st1.name,st1.branch);

}
```

## Array of Structures in C

An array of structures in C can be defined as the collection of multiple structure variables where each variable contains information about different entities. The array of structures in C are used to store information about multiple entities of different data types. The array of structures is also known as the collection of structures.

## C program to print Student information using array of structure

```
#include<stdio.h>
#include <string.h>
struct student
{
int rollno;
char name[10];
};
int main()
{
int i;
struct student st[4];
printf("Enter Records of 4
students"); for(i=0;i<4;i++)
{
printf("\nEnter Rollno:");
scanf("%d",&st[i].rollno);
printf("\nEnter Name:");
scanf("%s",st[i].name);
}
```

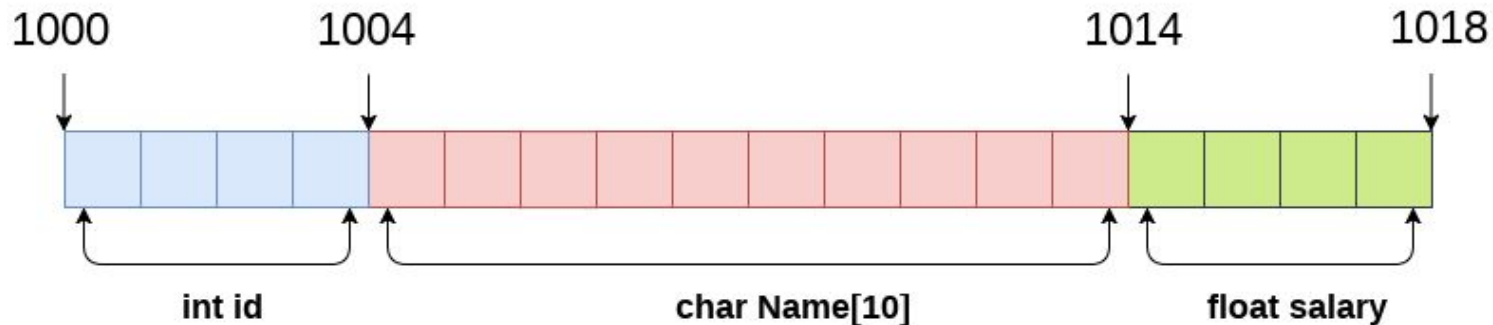
```
printf("\nStudent Information List:");
for(i=0;i<4;i++)
{
printf("\nRollno:%d, Name:%s",st[i].rollno,st[i].name);
}
return 0;
}
```

Id	Name	Branch
1	shailesh	EX
2	Amar	ECS
3	Pratik	Comp
4	Mugda	ECS

## Examples for Practice

- ▶ A Hospital needs to maintain details of patients. Details to be maintained are First name, Middle name, Surname, Date of Birth, Disease. Write a program which will print the list of all patients with given disease.
- ▶ Define a structure called Player with data members Player name, team name, batting average Create array of objects, store information about players, Sort and display information of players in descending order of batting average.

# Memory Allocation in Structure



```
struct Employee
{
    int id;
    char Name[10];
    float salary;
} emp;
```

`sizeof (emp) = 4 + 10 + 4 = 18 bytes`

where;  
`sizeof (int) = 4 byte`  
`sizeof (char) = 1 byte`  
`sizeof (float) = 4 byte`



# Dynamic Memory Allocation in C.

**C Dynamic Memory Allocation** can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.

C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under **<stdlib.h>** header file to facilitate dynamic memory allocation in C programming. They are:

- malloc()
- calloc()
- free()
- realloc()



## C malloc() method

The “malloc” or “memory allocation” method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn't initialize memory at execution time so that it has initialized each block with the default garbage value initially.

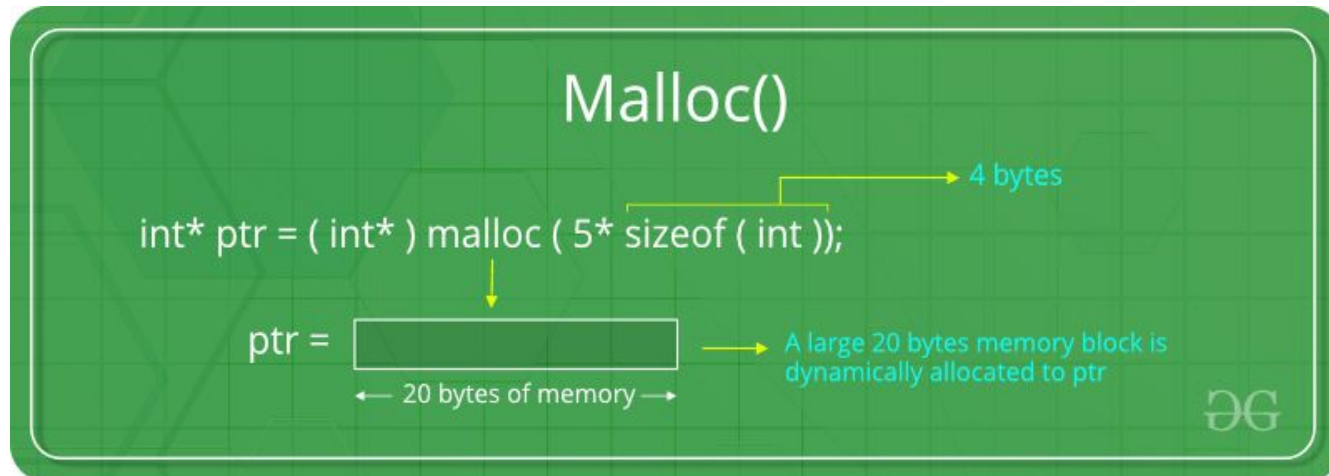
### Syntax:

`ptr = (cast-type*) malloc(byte-size)`

For Example:

***ptr = (int\*) malloc(100 \* sizeof(int));***

*Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.*



# calloc() method

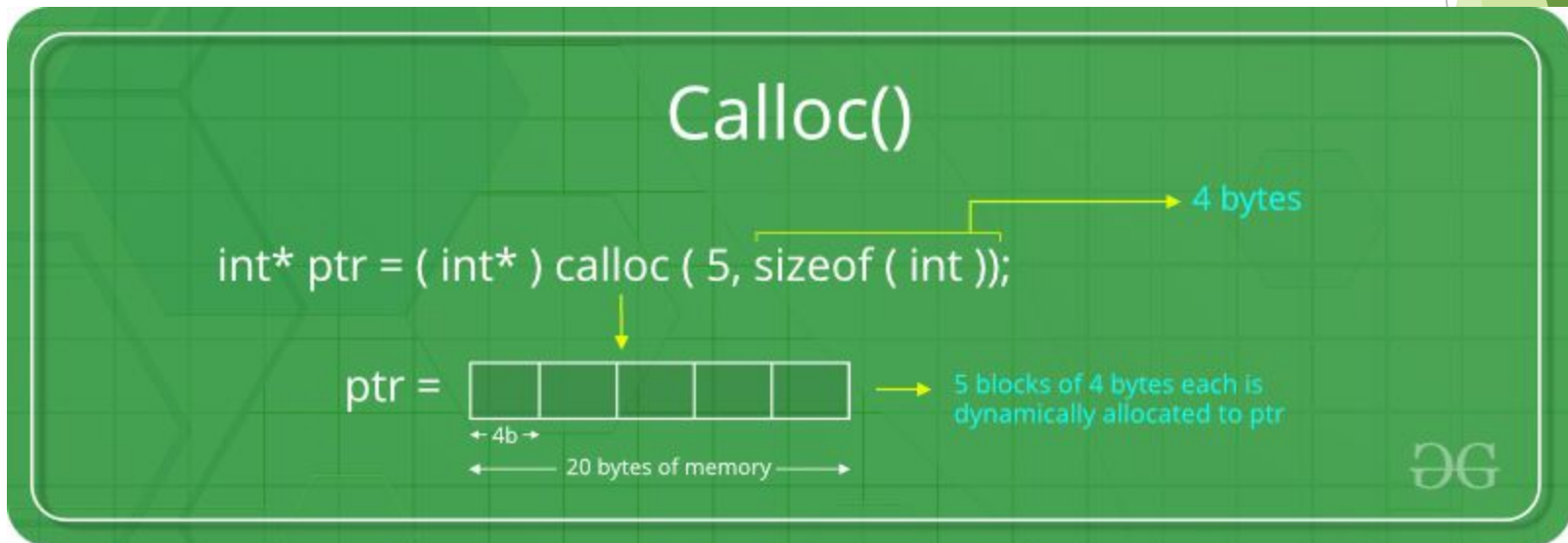
“calloc” or “contiguous allocation” method in C is used to dynamically allocate the **specified number of blocks of memory of the specified type**. it is very much similar to malloc() but has two different points and these are:

It initializes each block with a default value ‘0’.

It has two parameters or arguments as compare to malloc().

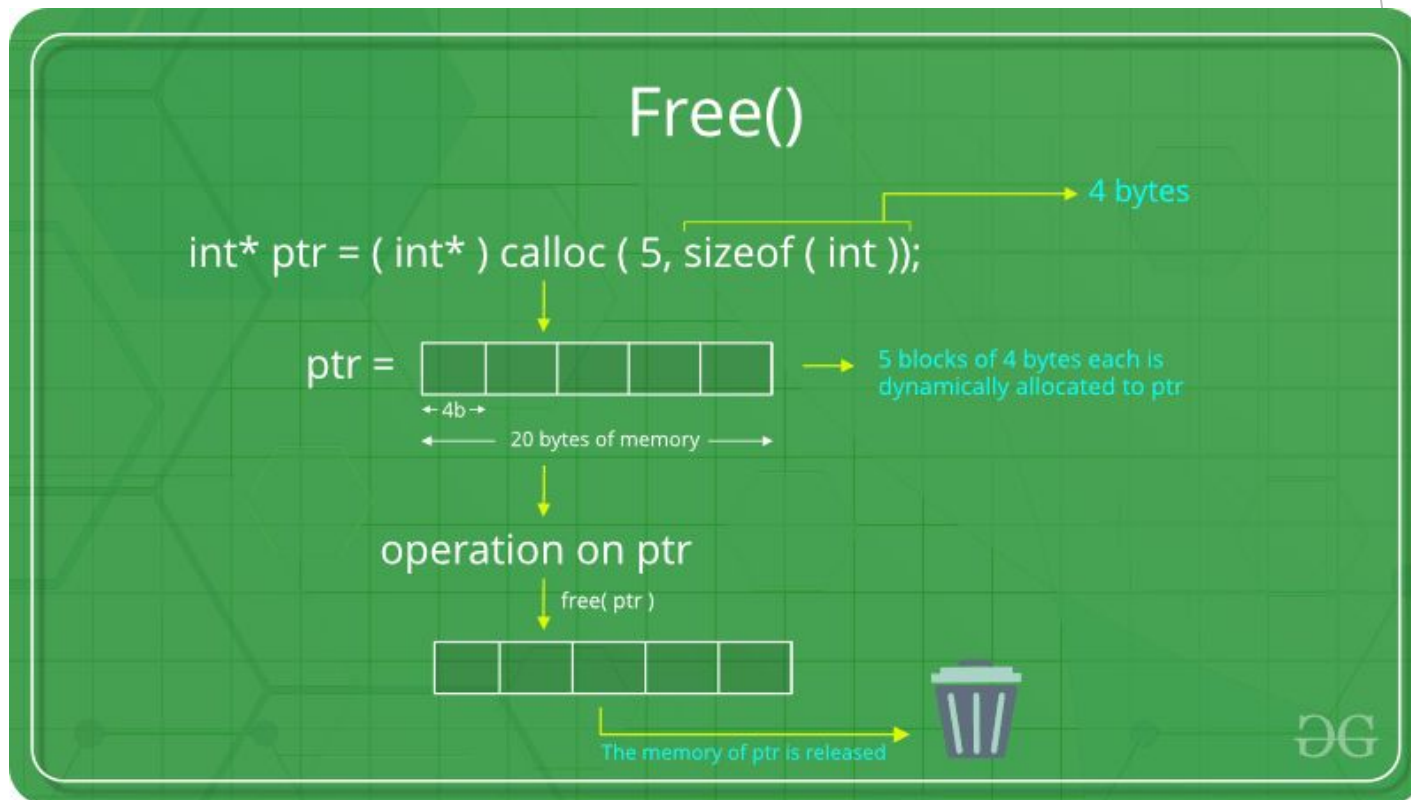
```
ptr = (cast-type*)calloc(n, element-size);
```

here, n is the no. of elements and element-size is the size of each element.



## C free() method

“free” method in C is used to dynamically **deallocate** the **memory**. The memory allocated using functions malloc() and calloc() is not deallocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

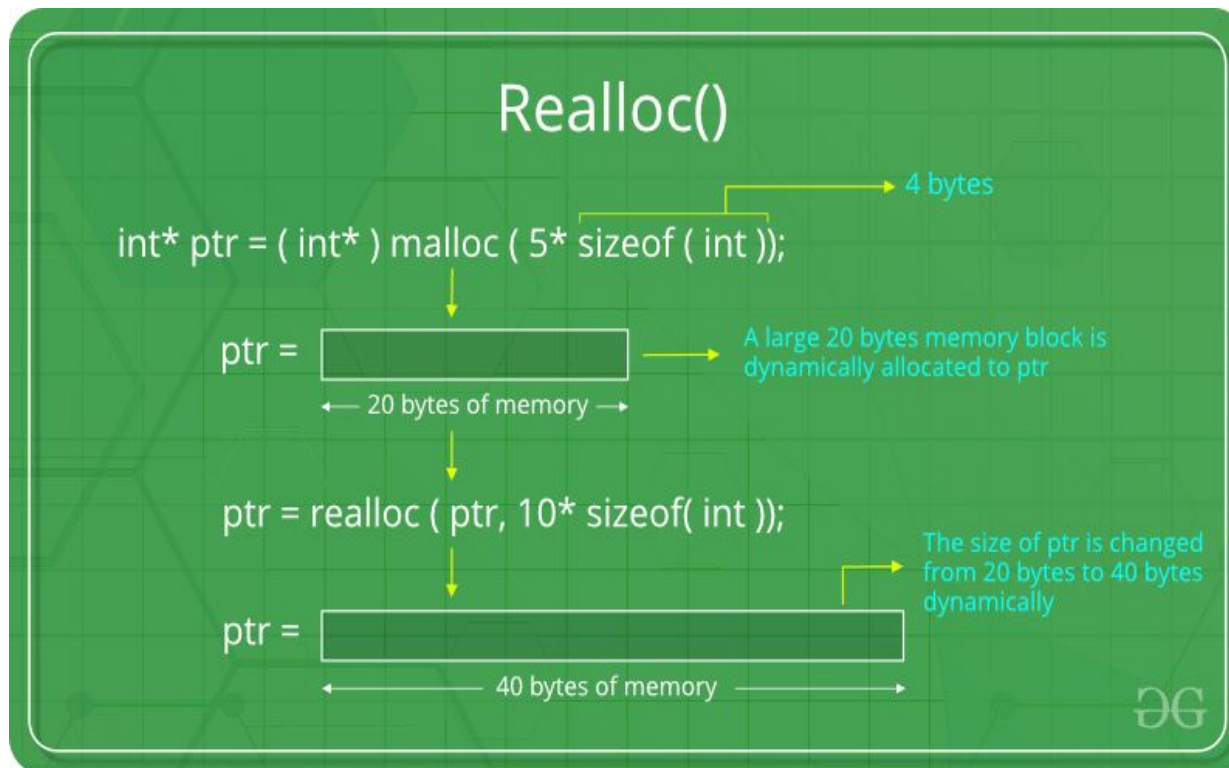


## C realloc() method

“realloc” or “re-allocation” method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to dynamically re-allocate memory. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

```
ptr = realloc(ptr, newSize);
```

where ptr is reallocated with new size 'newSize'.



# Stack Implementation Using Linked List

- In the linked list implementation, we would:
  - Maintain the stack as a linked list.
  - A pointer variable top points to the start of the list.
  - The first element of the linked list is considered as the stack top.

Code: <https://www.scaler.com/topics/c/stack-using-linked-list-in-c/>

<https://www.codesdope.com/blog/article/making-a-stack-using-linked-list-in-c/>  
(commented)

# Contd.

- Basic concept:
  - Insertion (push) and deletion (pop) operations take place at one end of the list only.
  - For stack creation / push operation
    - Required to call malloc function
  - How to check stack underflow?
    - Simply check if top points to NULL.
  - How to check overflow?
    - Check if malloc\* returns -1.

*\*The malloc() function stands for memory allocation, that allocate a block of memory dynamically. It reserves the memory space for a specified size and returns the null pointer, which points to the memory location.*

# Sample Usage

```
stack A, B;
```

```
create (A);   create (B);
```

```
push (A, 10); push (A, 20); push (A, 30);
```

```
push (B, 5);   push (B, 25); push (B, 10);
```

```
printf (“\n%d %d %d”, pop(A), pop(A), pop(A));
```

```
printf (“\n%d %d %d”, pop(B), pop(B), pop(B));
```

```
if (not isfull (A))
```

```
    push (A, 50);
```

```
if (not isempty (A))
```

```
    k = pop (A);
```

30	20	10
10	25	5

# Declaration

```
#define MAXSIZE 100

struct lifo
{
    int st[MAXSIZE];
    int top;
};
typedef struct lifo
    stack;
stack s;
```

ARRAY

```
struct lifo
{
    int value;
    struct lifo *next;
};
typedef struct lifo
    stack;

stack *top;
```

LINKED LIST



# Stack Creation

```
void create (stack *s)
{
    s->top = -1;

    /* s->top points to
       last element
       pushed in;
       initially -1 */
}
```

**ARRAY**

```
void create (stack **top)
{
    *top = NULL;

    /* top points to NULL,
       indicating empty
       stack */
}
```

**LINKED LIST**

# Pushing an element into stack

```
void push (stack *s, int element)
{
    if (s->top == (MAXSIZE-1))
    {
        printf ("\n Stack
overflow");
        exit(-1);
    }
    else
    {
        s->top++;
        s->st[s->top] = element;
    }
}
```

**ARRAY**

```
void push (stack **top, int element)
{
    stack *new;

    new = (stack *)malloc
(sizeof(stack));
    if (new == NULL)
    {
        printf ("\n Stack is full");
        exit(-1);
    }

    new->value = element;
    new->next = *top;
    *top = new;
}
```

**LINKED LIST**

# Popping an element from stack

```
int pop (stack *s)
{
    if (s->top == -1)
    {
        printf ("\n Stack
underflow");
        exit(-1);
    }
    else
    {
        return (s->st[s->top--]);
    }
}
```

**ARRAY**

```
int pop (stack **top)
{
    int t;
    stack *p;
    if (*top == NULL)
    {
        printf ("\n Stack is empty");
        exit(-1);
    }
    else
    {
        t = (*top)->value;
        p = *top;
        *top = (*top)->next;
        free (p);
        return t;
    }
}
```

**LINKED LIST**

# Checking for stack empty

```
int isempty (stack *s)
{
    if (s->top == -1)
        return 1;
    else
        return (0);
}
```

ARRAY

```
int isempty (stack *top)
{
    if (top == NULL)
        return (1);
    else
        return (0);
}
```

LINKED LIST

# Checking for Stack Full

```
int isempty (stack *s)
{
    if (s->top == -1)
        return 1;
    else
        return (0);
}
```

**ARRAY**

```
int isempty (stack *top)
{
    if (top == NULL)
        return (1);
    else
        return (0);
}
```

**LINKED LIST**

# Example: A Stack using an Array

```
#include <stdio.h>
#define MAXSIZE 100

struct lifo
{
    int st[MAXSIZE];
    int top;
};

typedef struct lifo stack;

main() {
    stack A, B;
    create(&A);
    create(&B);
    push(&A, 10);
    push(&A, 20);
    push(&A, 30);
    push(&B, 100);
    push(&B, 5);

    printf ("%d %d", pop(&A), pop(&B));

    push (&A, pop(&B));

    if (isempty(&B))
        printf ("\n B is empty");
    return;
}
```

# Example: A Stack using Linked List

```
#include <stdio.h>
struct lifo
{
    int value;
    struct lifo *next;
};
typedef struct lifo stack;

main() {
    stack *A, *B;
    create(&A);
    create(&B);
    push(&A, 10);
    push(&A, 20);
    push(&A, 30);
    push(&B, 100);
    push(&B, 5);

    printf ("%d %d", pop(&A), pop(&B));

    push (&A, pop(&B));

    if (isempty(B))
        printf ("\n B is empty");
    return;
}
```

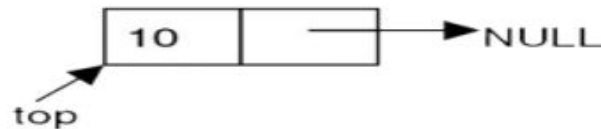
# Stack using SLL

## Understanding PUSH

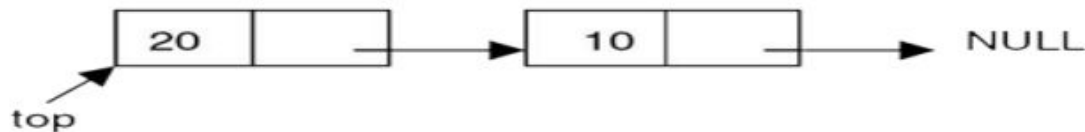
top → NULL

*head to SLL is top to STACK*

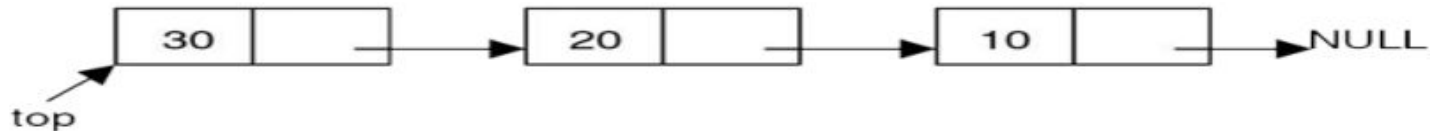
Initially



After first iteration



After second iteration



After third iteration



After last iteration

*insertBegin to SLL  
is push to STACK*



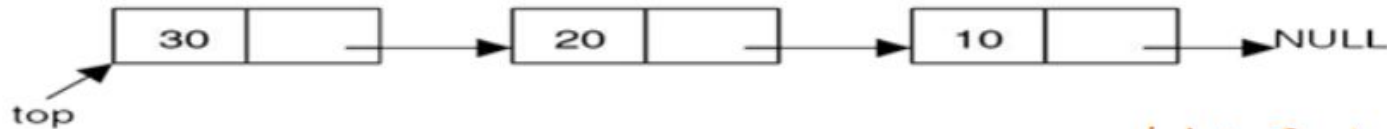
# Stack using SLL

## Understanding POP

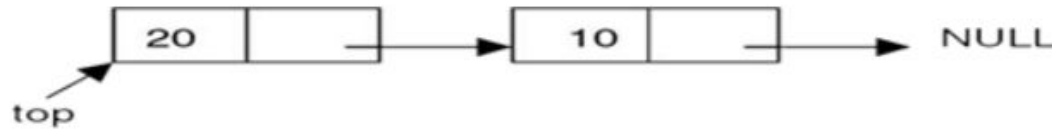
Initially



After first iteration



After second iteration



*deleteBegin to SLL is  
pop to STACK*

After third iteration



After last iteration

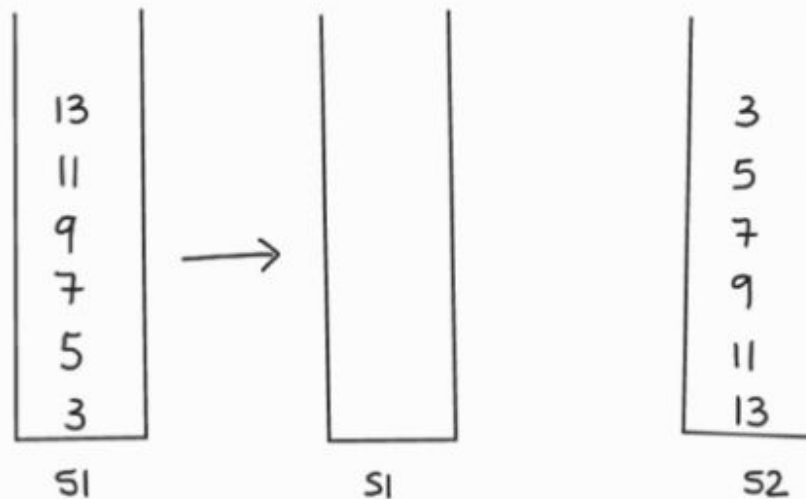
top → NULL

*isEmpty to SLL is  
isEmpty to STACK*

1. Imagine we have two empty stacks of integers, s1 and s2. Draw a picture of each stack after the following operations:

- i.
- ```
pushStack(s1, 3);
pushStack(s1, 5);
pushStack(s1, 7);
pushStack(s1, 9);
pushStack(s1, 11);
pushStack(s1, 13);
while (femptyStack(s1))
{
popStack(s1, x);
pushStack(s2,x);
} //while
```

Stack - Question 1.i

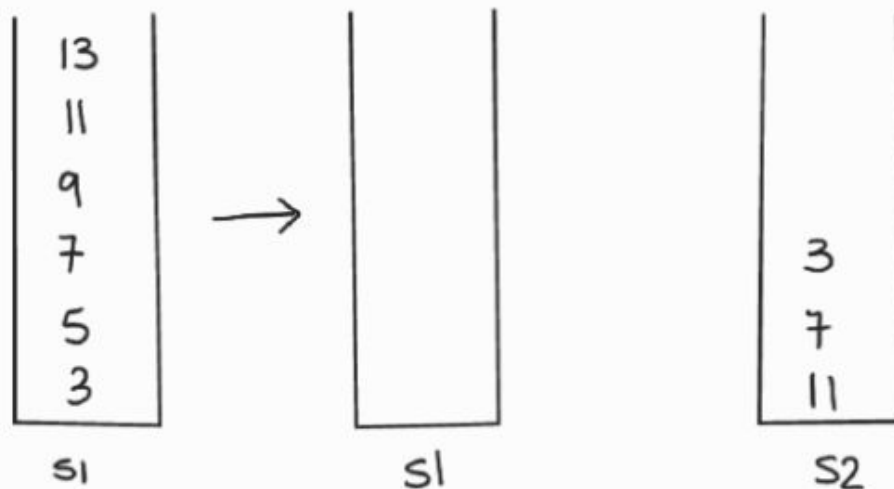


Correct

ii.

```
pushStack(s1, 3);
pushStack(s1, 5);
pushStack(s1, 7);
pushStack(s1, 9);
pushStack(s1, 11);
pushStack(s1, 13);
while (femptyStack(s1))
{
    popStack(s1, x);
    popStack(s1, x);
    pushStack(s2,x);
} //while
```

Stack - Question 1.ii



Correct

Imagine we have two empty stacks of integers,  $s_1$  and  $s_2$ . Draw a picture of each stack after the following operations:

```
pushStack (s1, 3);
pushStack (s1, 5);
pushStack (s1, 7);
pushStack (s1, 9);
while (!emptyStack (s1)) {
  popStack (s1, x);
  pushStack (s2, x);
} //while
pushStack (s1, 11);
pushStack (s1, 13);
while (!emptyStack (s2)) {
  popStack (s2, x);
  pushStack (s1, x);
} //while
```

**Solution:**

---

|    |    |
|----|----|
| 9  |    |
| 7  |    |
| 5  |    |
| 3  |    |
| 13 |    |
| 11 |    |
| S1 | S2 |

---

# Applications of Stacks

- Direct applications:
  - Page-visited history in a Web browser
  - Undo sequence in a text editor
  - Chain of method calls in the Java Virtual Machine
  - Used to solve problem like Infix to Prefix Conversion
- Indirect applications:
  - Auxiliary data structure for algorithms
  - Component of other data structures

# STACK VERSUS LINKED LIST

| STACK                                                                                                              | LINKED LIST                                                                               |
|--------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------|
| An abstract data type that serves as a collection of elements with two principal operations which are push and pop | A linear collection of data elements whose order is not given by their location in memory |
| Push, pop and peek are the main operations performed on a stack                                                    | Insert, delete and traversing are the main operations performed on a linked list          |
| Can read the topmost element                                                                                       | It is required to traverse through each element to access a specific element              |
| Works according to the FIFO mechanism                                                                              | Elements connect to each other by references                                              |
| Simpler than linked list                                                                                           | More complex than stack                                                                   |



# Application of Stacks

- Prefix and Postfix expressions can be evaluated faster than an infix expression. This is because we don't need to process any brackets or follow operator precedence rule.
- In postfix and prefix expressions which ever operator comes before will be evaluated first, irrespective of its priority. Also, there are no brackets in these expressions.

# Infix and Postfix Notations

- Infix: operators placed between operands:

$$A+B*C$$

- Postfix: operands appear before their operators:-

$$ABC*+$$

- There are no precedence rules to learn in postfix notation, and parentheses are never needed



# Infix to Postfix

| Infix               | Postfix         |
|---------------------|-----------------|
| $A + B$             | $A B +$         |
| $A + B * C$         | $A B C * +$     |
| $(A + B) * C$       | $A B + C *$     |
| $A + B * C + D$     | $A B C * + D +$ |
| $(A + B) * (C + D)$ | $A B + C D + *$ |
| $A * B + C * D$     | $A B * C D * +$ |

$A + B * C \quad \square \quad (A + (B * C)) \quad \square \quad (A + (B C *)) \quad \square \quad A B C * +$

$A + B * C + D \quad \square \quad ((A + (B * C)) + D) \quad \square \quad ((A + (B C *)) + D) \quad \square \quad ((A B C * +) + D) \quad \square \quad A B C * + D +$

# Evaluation of Postfix Expression

Postfix expression: 6 5 2 3 + 8 \* + 3 + \*

| Symbol | Operand 1 | Operand 2 | Value | Stack      | Remarks                                                                             |
|--------|-----------|-----------|-------|------------|-------------------------------------------------------------------------------------|
| 6      |           |           |       | 6          |                                                                                     |
| 5      |           |           |       | 6, 5       |                                                                                     |
| 2      |           |           |       | 6, 5, 2    |                                                                                     |
| 3      |           |           |       | 6, 5, 2, 3 | The first four symbols are placed on the stack.                                     |
| +      | 2         | 3         | 5     | 6, 5, 5    | Next a '+' is read, so 3 and 2 are popped from the stack and their sum 5, is pushed |
| 8      | 2         | 3         | 5     | 6, 5, 5, 8 | Next 8 is pushed                                                                    |
| *      | 5         | 8         | 40    | 6, 5, 40   | Now a '*' is seen, so 8 and 5 are popped as $8 * 5 = 40$ is pushed                  |
| +      | 5         | 40        | 45    | 6, 45      | Next, a '+' is seen, so 40 and 5 are popped and $40 + 5 = 45$ is pushed             |
| 3      | 5         | 40        | 45    | 6, 45, 3   | Now, 3 is pushed                                                                    |
| +      | 45        | 3         | 48    | 6, 48      | Next, '+' pops 3 and 45 and pushes $45 + 3 = 48$ is pushed                          |
| *      | 6         | 48        | 288   | 288        | Finally, a '*' is seen and 48 and 6 are popped, the result $6 * 48 = 288$ is pushed |

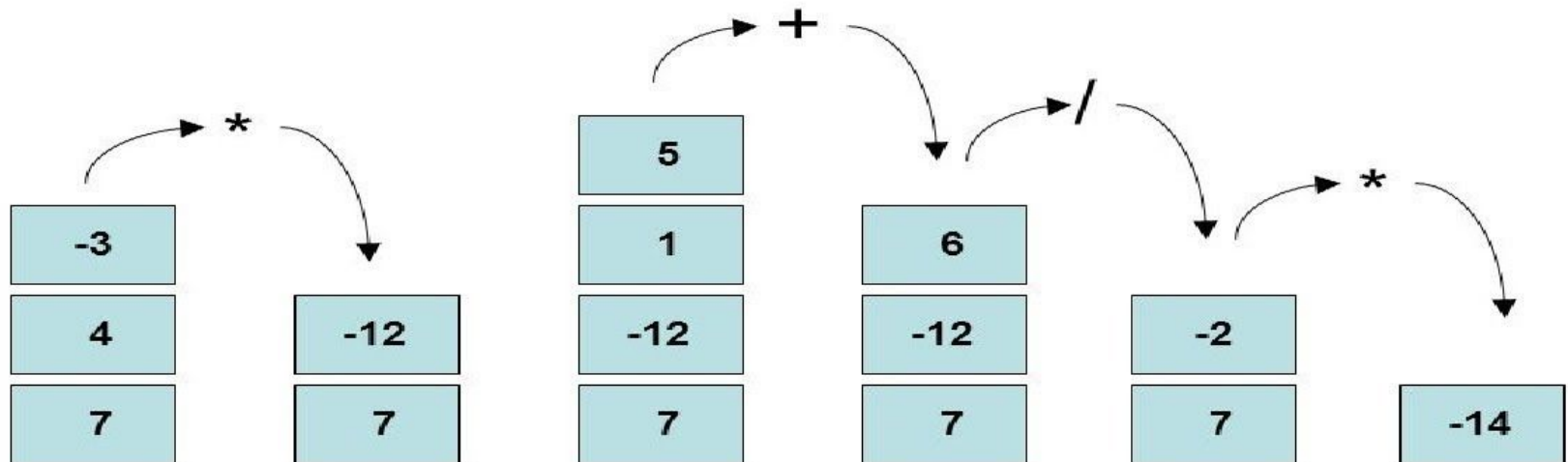
In **post-fix notation**, the operands come before their operators.

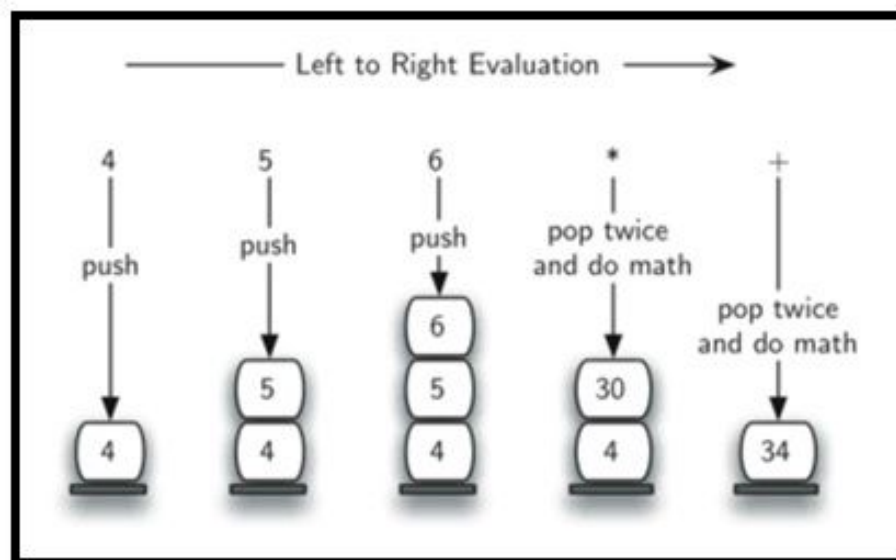
For the evaluation of post-fix notation, we use the **stack** data structure. The following are the **rules** of evaluating post-fix notation using stack:

- Start scanning from left to right.
- If the current value is an operand, push it onto the stack.
- If the current is an operator, pop two elements from the stack, apply the at-hand operator on those two popped operands, and push the result onto the stack.
- At the end, pop an element from the stack, and that is the answer.

# Evaluating Postfix Expressions

- Expression = 7 4 -3 \* 1 5 + / \*





| Step | Input Symbol | Operation                     | Stack | Calculation |
|------|--------------|-------------------------------|-------|-------------|
| 1.   | 4            | Push                          | 4     |             |
| 2.   | 5            | Push                          | 4,5   |             |
| 3.   | 6            | Push                          | 4,5,6 |             |
| 4.   | *            | Pop(2 elements)<br>& Evaluate | 4     | $5*6=30$    |
| 5.   |              | Push result(30)               | 4,30  |             |
| 6.   | +            | Pop(2 elements)<br>& Evaluate | Empty | $4+30=34$   |
| 7.   |              | Push result(34)               | 34    |             |
| 8.   |              | No-more elements(pop)         | Empty | 34(Result)  |

## Evaluation of Postfix Expression (using Stack)

2 10 + 9 6 - /

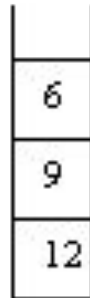
push 2  
push 10



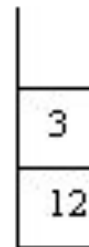
pop 10  
pop 2  
push  $2 + 10 = 12$



push 9  
push 6



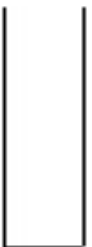
pop 6  
pop 9  
push  $9 - 6 = 3$



pop 3  
pop 12  
push  $12 / 3 = 4$



pop answer: 4



**Code:**

<https://www.geeksforgeeks.org/evaluation-of-postfix-expression/>




# Evaluation of Prefix Expression (using Stack)

Example:  $-*+4325$

| Symbol | opnd1 | opnd2 | value | opndstack  |
|--------|-------|-------|-------|------------|
| 5      |       |       |       | 5          |
| 2      |       |       |       | 5, 2       |
| 3      |       |       |       | 5, 2, 3    |
| 4      |       |       |       | 5, 2, 3, 4 |
| +      | 4     | 3     | 7     | 5, 2       |
|        |       |       |       | 5, 2, 7    |
| *      | 7     | 2     | 14    | 5          |
|        |       |       |       | 5, 14      |
| -      | 14    | 5     | 9     |            |
|        |       |       |       | 9          |

result



## Algorithm:

EVALUATE\_PREFIX(String)

Step 1: Put a pointer P at the end of the end

Step 2: If character at P is an operand push it to Stack

Step 3: If the character at P is an operator pop two elements from the Stack. Operate on these elements according to the operator, and push the result back to the Stack

Step 4: Decrement P by 1 and go to Step 2 as long as there are characters left to be scanned in the expression.

Step 5: The Result is stored at the top of the Stack, return it

Step 6: End



Expression: +9\*26

| Character | Stack              | Explanation                                                                    |
|-----------|--------------------|--------------------------------------------------------------------------------|
| Scanned   | (Front to<br>Back) |                                                                                |
| -----     |                    |                                                                                |
| 6         | 6                  | 6 is an operand,<br>push to Stack                                              |
| 2         | 6 2                | 2 is an operand,<br>push to Stack                                              |
| *         | 12 (6*2)           | * is an operator,<br>pop 6 and 2, multiply<br>them and push result<br>to Stack |
| 9         | 12 9               | 9 is an operand, push<br>to Stack                                              |
| +         | 21 (12+9)          | + is an operator, pop<br>12 and 9 add them and<br>push result to Stack         |

Result: 21

## Examples:

Input :  $-+8/632$

Output : 8

Input :  $-+7*45+20$

Output : 25

# Evaluation of Prefix Expression (using Stack)

In **prefix notation**, the operators come before their operands.

Code (C++):

<https://www.geeksforgeeks.org/evaluation-prefix-expressions/>

Code in C:

<https://docs.google.com/document/d/11eq6pX267wqccTfWQDuptwkksjBzAvUkJax-DJBXuwc/edit?usp=sharing>

# Stack Application

1. Code to **reverse a stack using recursion:**

<https://www.techcrashcourse.com/2016/06/c-program-to-reverse-stack-using-recursion.html>

2. Code to **reverse a string using stack:**

<https://www.geeksforgeeks.org/stack-set-3-reverse-string-using-stack/>