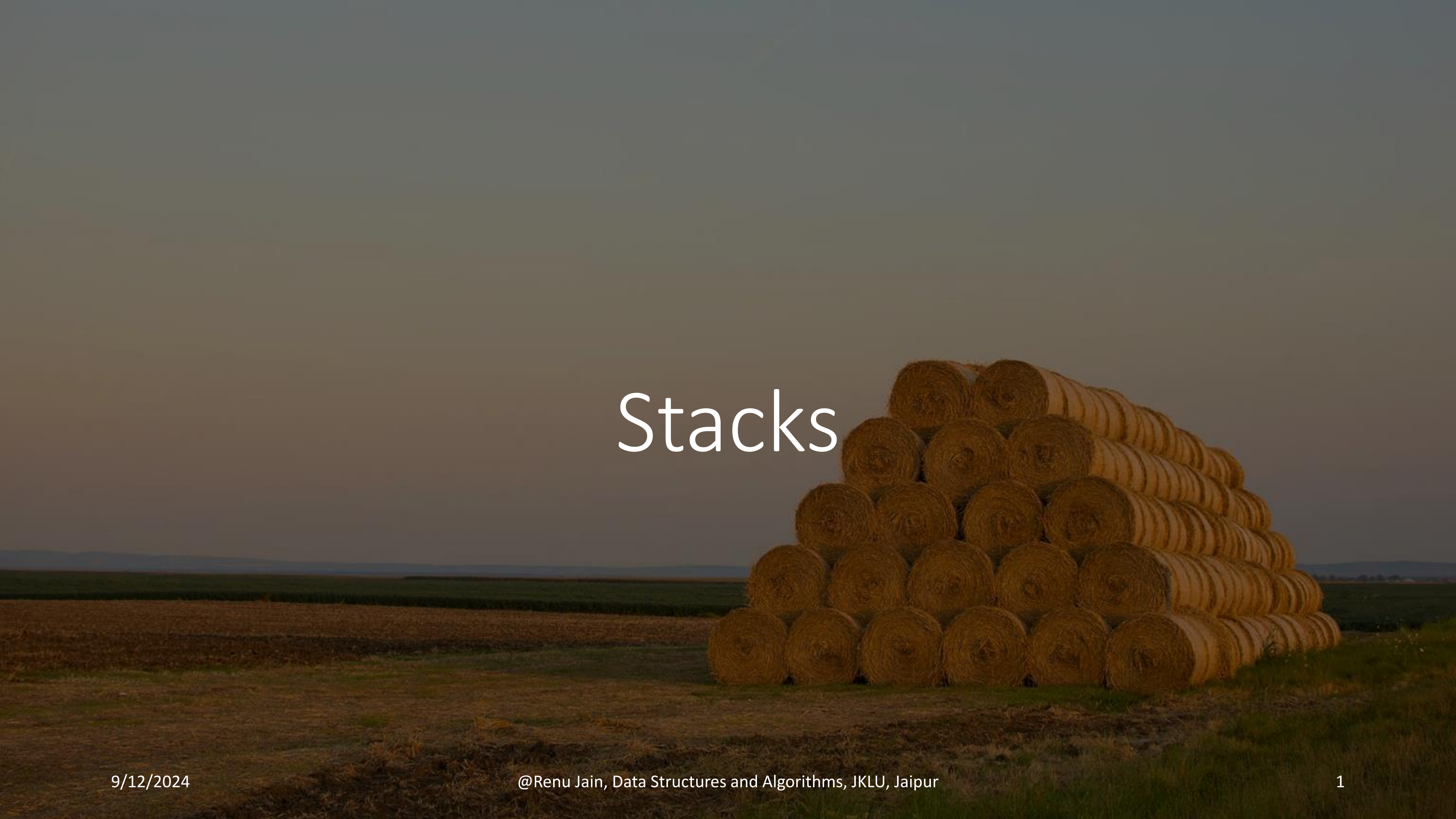

Stacks

Stack

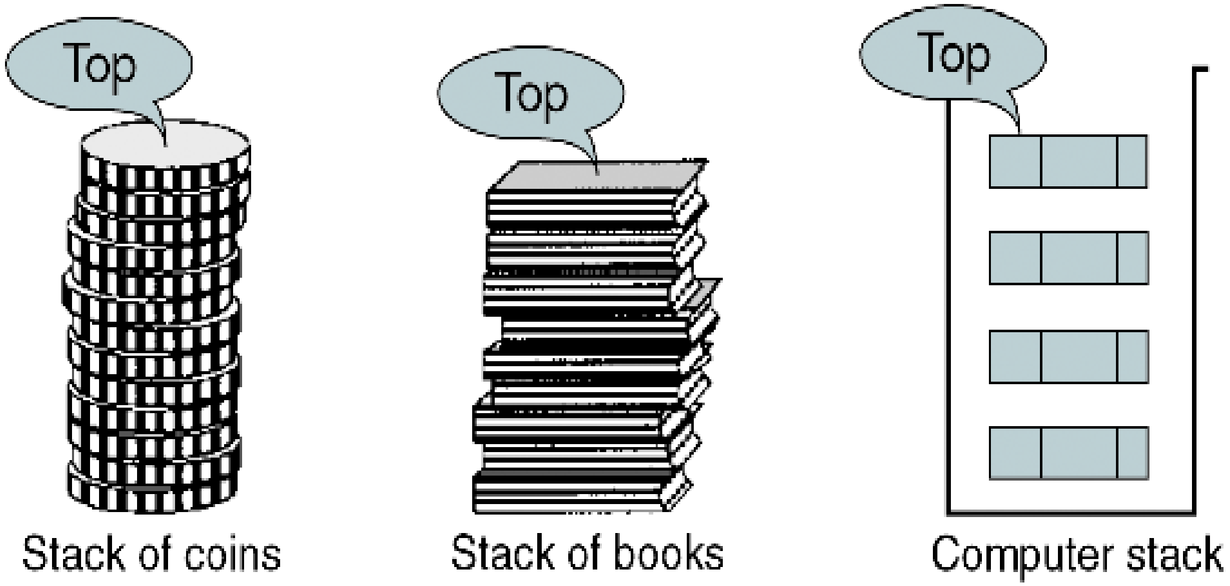

FIGURE 3-1 Stack

What is stack??

- An arrangement of similar type of items where insertions and deletions shall be done in a well-defined manner (Simple definition)
- An ordered collection of similar items in which new items may be inserted and may be deleted ***at one end only***, called the ***top of the stack.***
- **Insert** an item: add the item on top of previous top item
- **Delete** or remove an item: Remove the top most item

Note*: In stack data structure, we can not insert elements at any place, and we can not remove from any place in the collection*

Cont….

- When a new item is put on the top of the stack, stack grows upwards
- Items which are at the top of stack may be removed one by one and stack moves downward
- It is different from arrays because the definition of stack dictates - how to insert or delete items
- We need to define ***stack top*** in stacks
- So, stack is a dynamic, constantly changing object
- Stack is called Last-in First-out (LIFO) data structure

Stack Operations

***Special names are given to operations which can be performed on a stack and these are*:**

- **Push** : When an item is added to a stack, it is pushed onto the stack
- **Pop** : It removes the top element
- **Stacktop** : It just tells what is the topmost element of stack, it does not remove the element

Push Operation

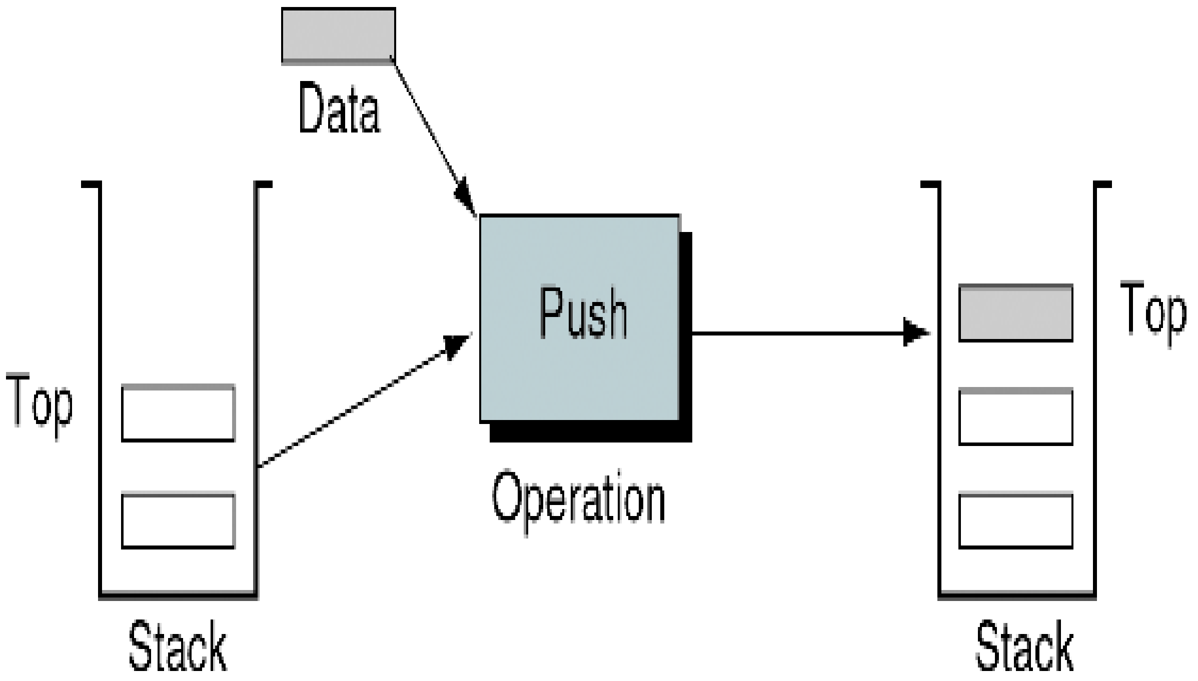

FIGURE 3-2 Push Stack Operation

Pop operation

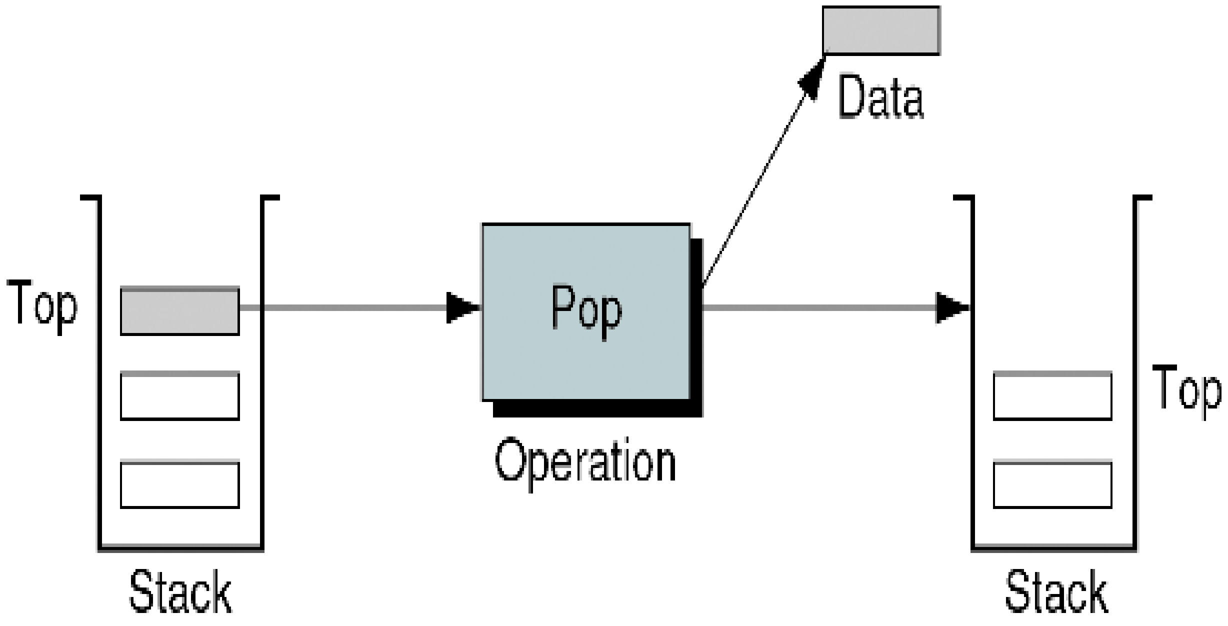

FIGURE 3-3 Pop Stack Operation

Last-in First-out (LIFO)

Stack is called Last-in First-out (LIFO) data structure

Example: stack of books

How to build a stack:

Initially empty space

Add a book A

Add a book B

Add a book C

Delete a book

Add a book D

Delete a book

Implementation of stacks using arrays

What does a stack need:

1. Space
2. Top of the stack

Note: In general stack can grow to any size but in real life, space is always finite

There are two possibilities:

1. *Predefine the size of the stack*
2. *Dynamic allocation of space as required (No need to allocate prior memory)*

Implementation of stacks using arrays

```
struct mystack_def
{ int top;
int items[100];
};

struct mystack_def s1, s2;
```

How to access an element of s1?

Algorithms to perform stack operations

Push ---insert a new element in the existing stack

- Check for overflow

If not

- Increment top and insert new element at top

Pop – delete an element from the stack

- Check for underflow

If not

- Take out the top element and decrement top

What is underflow and Overflow?

Underflow:

- There is no element in the stack
- Trying to pop an element from the stack

Overflow:

There is no space in the stack

Trying to push a new element on the stack

If stack size is an array of size M and initially top = -1

Underflow: top = = -1

Overflow: top = = M-1

Stack functions

- **Empty :** **Check** whether stack is empty (required at the time of pop)
- **Full :** **Check** whether stack is full (required at the time of push)
- **Push :** Needs data that needs to be kept on stack
- **Pop : R**eturns data which is removed from stack (top element)
- **Initialize** : Initialize the stack as empty by top=-1

Steps to write a program using stack data structure

Let us write a simple program where we push an item if it is not 100 and pop an item if input is 100.

Steps:

1. Define stack (create the structure for stack)
   - Identify the **element type** of stack (For exp: stack of numbers, stack of books, stack of characters, etc
   - Decide the maximum **size of stack** if implementing using arrays

2. Write all stack functions according to the structure defined

3. Write main program to incorporate the functionality of program

stack functions

```
struct mystack_def{int top;
                   int items[100];
}; typedef struct mystack_def  mstack;
void initialize( mstack *fs)
{ fs ->top = -1;}
int empty( mstack *fs)
{ if (fs ->top ==-1) return 1; else
return 0;}
int full( mstack *fs)
{ if (fs ->top ==99) return 1; else
return 0;}
```

stack functions…

```
void push( mstack *fs, int i)
{ if!(full(fs))
 { fs ->top = fs ->top +1;
 fs ->item[fs ->top ] = i;
  }
int pop( mstack *fs)
{ if!(empty(fs))
 return(fs ->item[fs ->top-- ]);
  }
```

main program

```
int main()
{mstack s;
int a[10] = {10,20, 100,30,40,5,8,90,100,30};
int i, temp;
Initialize(&s);
for(i= 0; i++;i<10)
{ if (a[i] !=100)
 push(&s,a[i]);
 else { if (a[i] ==100) temp = pop(&s);
printf(“the popped element is : %d\n”,temp);}}
while(!(empty(&s)) printf(“the remaining  elements on stack : %d\n”,pop(&s));
}
```

Applications of stack

Given a mathematical expression, find whether parenthesis are nested correctly or not (only one type of parenthesis)

We need to ensure that:

- There are an equal number of right and left parentheses
- Every right parenthesis is preceded by a left parenthesis
- Exp: (A+B+C)), (A+B)), ((A+B), (a+b)) –(c-d)

To solve this :

- Think of each left parenthesis as opening scope and each right parenthesis as closing a scope.
- The nesting depth at a particular point is the number of scopes that have been opened but not yet closed at that point

Solution without using stacks

- We can define parenthesis count at a particular point in an expression as the number of left parenthesis minus the number of right parenthesis encountered till that point
- Then two conditions must hold good:

1. The parenthesis count at the end of the expression is 0
2. The parenthesis count at each point in the expression is nonnegative

Slightly modified parenthesis problem

Suppose, expression has three types of scope delimiters – parentheses (), brackets [] and braces {}

Now, we need to keep track of not only how many scopes have been opened but also their types

At every point for every type of scope we need to follow above

Solution…

The algorithm to solve above problem will be very simple and more logical if we make use of stack data structure

Algorithm to check parentheses using stacks

```
    valid = true
    S = empty stack
    while(we have read the complete
expression)
  { read the next character(symb)
 if(symb== '(' || symb == '[' ||symb ==
'{')
    push(s, symb)
if(valid)
 printf ("string is a valid string\n");
```

Cont..

if(symb== ‘)‘ || symb == ‘]‘ || symb = ‘}‘)

{ if (empty(s)) valid = false

else

{ch = pop(s)

if(ch ***not matching with*** symb) valid = false

}}}

if(!empty(s)) valid = false

if(valid) printf (“string is a valid string\n”

Assignments on stack

1. Write a program to check whether a given mathematical expression has proper placing of parentheses without using stacks
2. Write a program to check whether a given mathematical expression comprising of three types of scope delimiters ((),[],{})has proper placing of parentheses without using stacks
3. Write a program to check whether a given mathematical expression has proper placing of parentheses using stacks
4. Write a program to check whether a given mathematical expression comprising of three types of scope delimiters ((),[],{})has proper placing of parentheses using stacks

More applications of stack

- Reversing a string
- Polish notations used for mathematical expression
- Implementation of recursion
- Implementation of function calling

Function calling

```
void three()
{
printf("Three started\n");
printf("Three ended\n");
}
void two()
{
printf("Two started\n");
three();
printf("Two ended\n");
}
void one()
{
printf("One started\n");
two();
printf("One ended\n");
}
void main()
{
clrscr();
printf("Main started\n");
one();
printf("Main ended\n");
getch();
}
```

The order of starting and ending

Output

Main started
One started
Two started
Three started
Three ended
Two ended
One ended
Main ended