Data Structures and Algorithms

Mr. Tahir Iqbal

tahir.iqbal@bahria.edu.pk

Lecture 05: Stack

STACKS

- A stack is a linear list in which insertion and deletion take place at the same end
  - This end is called top
  - The other end is called bottom
- Stacks are known as LIFO (Last In, First Out) lists.
  - The last element inserted will be the first to be retrieved
- E.g. a stack of Plates, books, boxes etc.

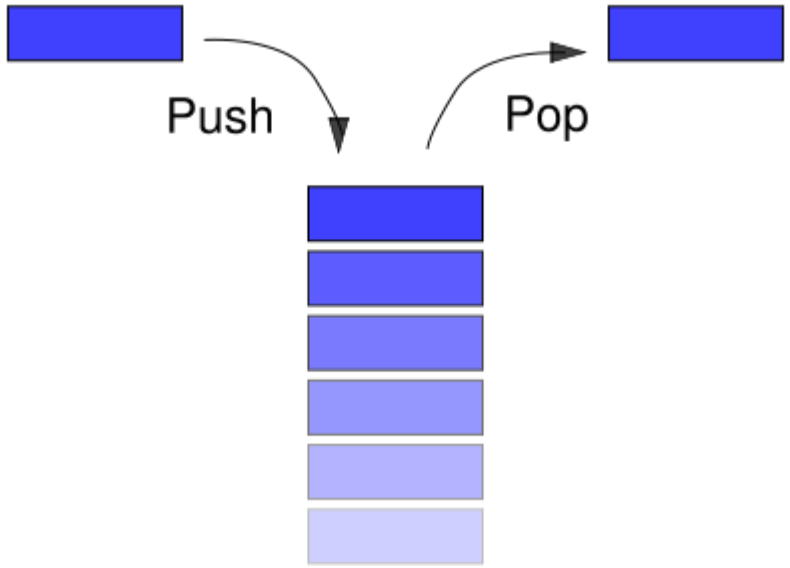
OPERATION ON STACK

- Creating a stack
- Checking stack---- either empty or full
- Insert (PUSH) an element in the stack
- Delete (POP) an element from the stack
- Access the top element
- Display the elements of stack
- Emptying a stack

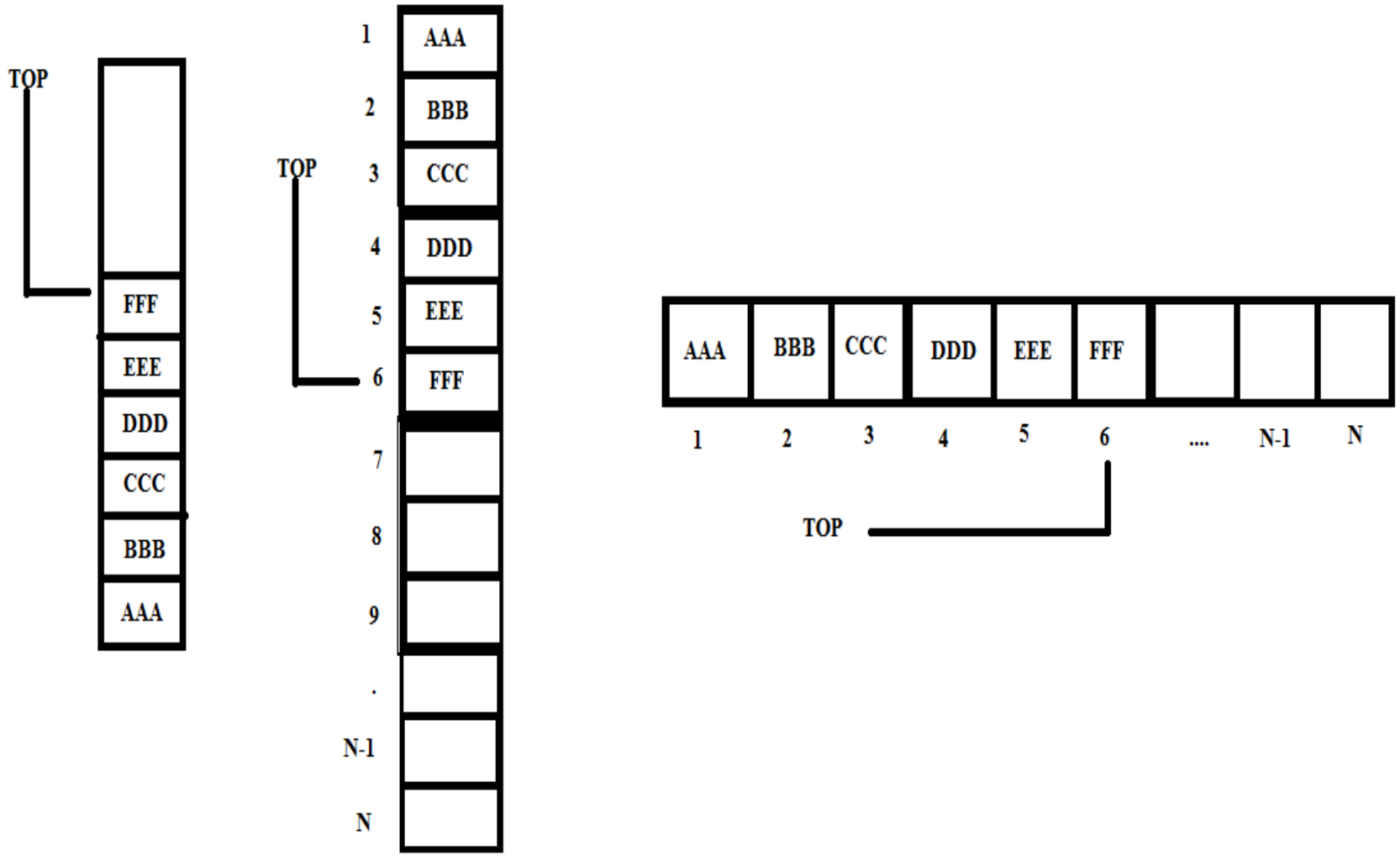
Stack adt

- A linear list in which insertion and deletion take place at the same end (i.e. top)
- Elements:
  - Top element
  - Rest of Stack
- Last-In-First-Out (LIFO)
- Operations:
  - createStack()
  - isFull()
  - isEmpty()
  - Push()
  - Pop()
  - displayTop()
  - displayStack()
  - emptyStack()

Insertion and deletion on stack

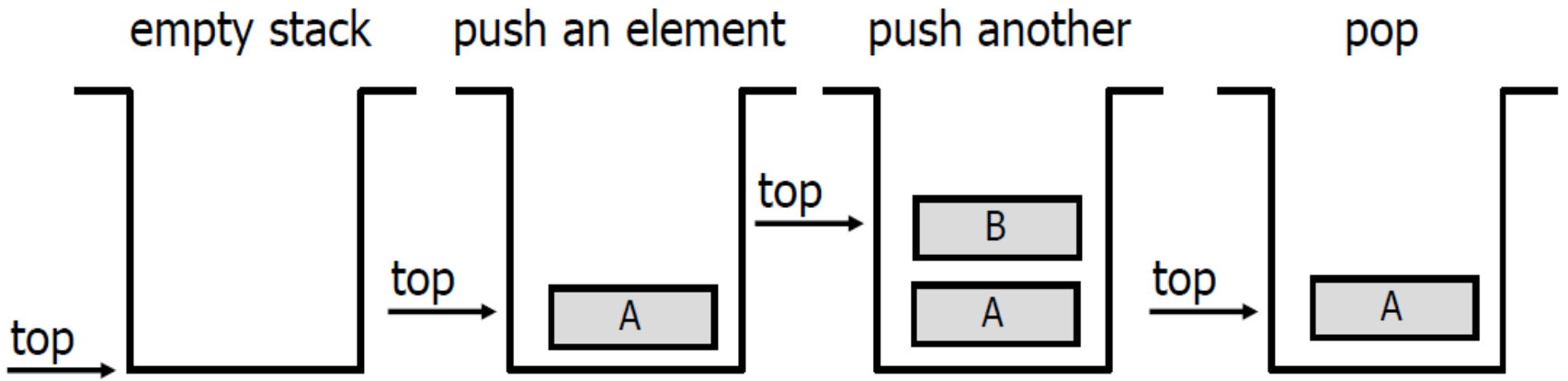


STACK REPRESENTATION



PUSH AND POP

- Primary operations: Push and Pop
- Push
  - Add an element to the top of the stack.
- Pop
  - Remove the element at the top of the stack.



STACK-RELATED TERMS

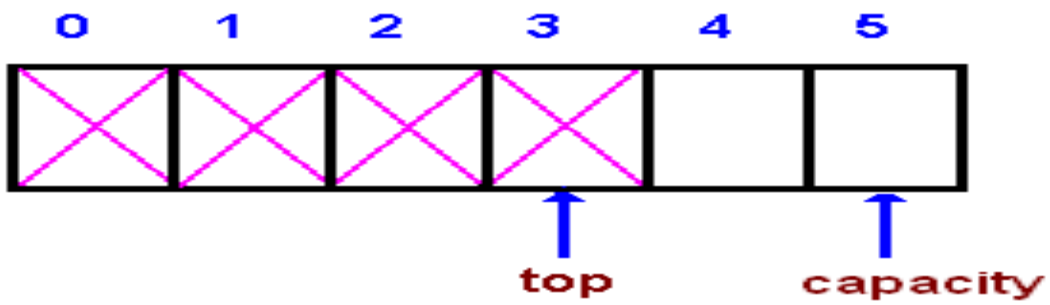
- Top
  - A pointer that points the top element in the stack.
- Stack Underflow
  - When there is no element in the stack, the status of stack is known as stack underflow.
- Stack Overflow
  - When the stack contains equal number of elements as per its capacity and no more elements can be added, the status of stack is known as stack overflow

STACK IMPLEMENTATION

- Implementation can be done in two ways
  - Static implementation
  - Dynamic Implementation
- Static Implementation
  - Stacks have **fixed size**, and are implemented as **arrays**
  - It is also inefficient for utilization of memory
- Dynamic Implementation
  - Stack **grow in size** as needed, and implemented as **linked lists**
  - Dynamic Implementation is done through pointers
  - The memory is efficiently utilize with Dynamic Implementations

Static Implementation

- Linear array STACK (to store elements of stack) .
- A pointer variable TOP (pointing towards the location of top element).
  - TOP =0 or NULL ->indicates that stack is empty
- A variable MAXSTK (giving the capacity of stack)



Static implementation

- PUSH Procedure:
  - PUSH(STACK, TOP, MAXSTK, ITEM)

    (This procedure pushes an ITEM onto a stack)

  1. [Stack already filled?]

     If TOP = MAXSTK, then: Print OVERFLOW, and Return.
  2. Set TOP := TOP + 1. [Increases TOP by 1]
  3. Set STACK[TOP] := ITEM. [Inserts ITEM in new TOP position]
  4. Return

Static implementation

- POP Procedure:
  - POP(STACK, TOP, ITEM)

    (This procedure deletes the top element of stack and assigns it to the variable ITEM)

  1. [Stack has an ITEM to be removed?]

     If TOP = 0, then: Print UNDERFLOW, and Return.
  2. Set ITEM := STACK[TOP]. [Assign TOP element to ITEM.]
  3. Set TOP := TOP - 1. [Decreases TOP by 1]
  4. Return

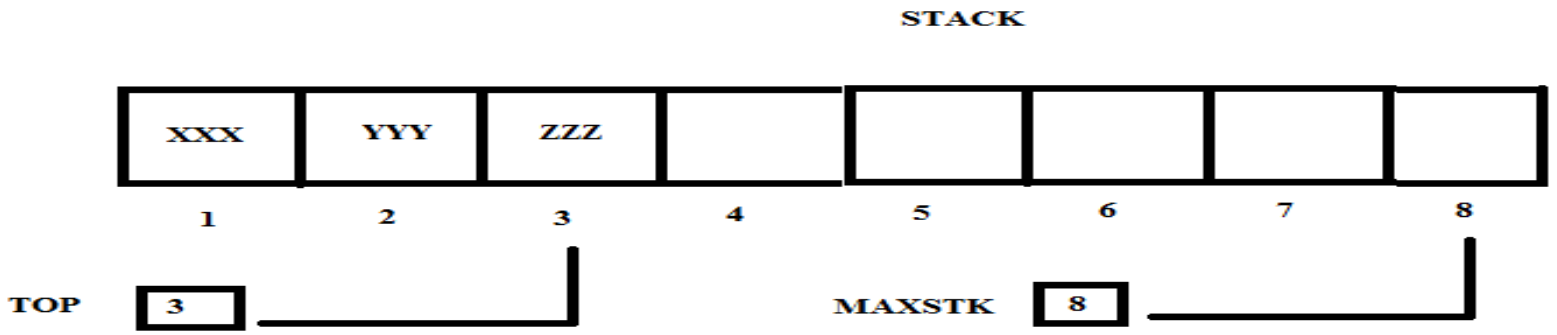
Static implementation

- Usually TOP and MAXSTK are global variables, therefore POP and PUSH can be called using:
  - PUSH(STACK, ITEM)
  - POP(STACK, ITEM)

  - *NOTE: TOP is changed before insertion in PUSH but value of TOP is changed after deletion in POP

EXAMPLE

- Consider Stack given below and simulate following operations



- PUSH(STACK, WWW)
  - Since TOP = 3, control is transferred to Step 2.
  - TOP = 3 + 1 = 4.
  - STACK[TOP]= STACK[4]=WWW.
  - Return.

- POP(STACK, ITEM)
  - Since TOP = 3, control is transferred to Step 2.
  - ITEM = ZZZ.
  - TOP = 3 – 1= 2.
  - Return.

Practice problem-1

- Consider the following stack of characters, where STACK is allocated N=8 memory cells:

  STACK: A,C,D,F,K,_____,_____,_____

  Describe the stack as the following operations take place:

  a) POP(STACK,ITEM)
  b) POP(STACK,ITEM)
  c) PUSH(STACK, L)
  d) PUSH(STACK, P)
  e) POP(STACK, ITEM)
  f) PUSH(STACK, R)
  g) PUSH(STACK, S)
  h) POP(STACK, ITEM)

Practice problem-2

- Suppose STACK is allocated N=6 memory cells and initially STACK is empty, or, in other words, TOP=0. Find the output of the following module:

1. Set AAA := 2 and BBB := 5.
2. Call PUSH(STACK, AAA).
   Call PUSH(STACK, 4).
   Call PUSH(STACK, BBB+2).
   Call PUSH(STACK, 9).
   Call PUSH(STACK, AAA+BBB).
3. Repeat while TOP ≠ 0:
   Call POP(STACK, ITEM).
   Write: ITEM.
   [End of loop.]
4. Return.

Stack Implementation

Stacks can be implemented in several ways using C++:

- Using Arrays
- Using Linked Lists
- Using Struct
- Using Classes

A sample Program using Array

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<iostream>
using namespace std;
#define length 10
int top=-1;
int stack[length];
int push();
int pop();
void listStack();
void main()
{
        int ch;
        do
        {
                cout << endl << "1.push";
                cout << endl << "2.pop";
                cout << endl << "3.exit";
                cout << endl << "enter choice";
                cin >> ch;
                switch(ch)
        {
         case 1:
                cout<<push();
                listStack();
                break;
         case 2:
                cout<<pop();
                listStack();
                break;
         case 3:
                exit(0);
         }
} while(3);
}
```

A sample Program using Array

```
int push()
{
        int data;
        if(top+1==length)
        {
          cout << "stack overflow
            \n Cannot enter new data";
        return -1;
        }
        top++;
        cout << "enter the data ";
        cin >> data;
        stack[top]=data;
        return stack[top];
}
```

```
int pop()
{
          int tempVar;
          if(top==-1)
        //we can also make isEmpty()
          {
             cout << "stack is underflow (Empty)";
             return -1;
          }
          tempVar=stack[top];
          top--;
          return(tempVar);
}
void listStack()
{
           cout << endl << "The stack is" <<
endl ;
           for(int i=top;i>=0;i--)
                cout << stack[i] << endl;
}
```

A sample Program using struct

```
#define max 5
struct stack{
  int top, a[max];
};
int pop(stack *);
int push(stack *);
void display(stack *p3){
 int i;
 if(p3->top==-1) {
   cout<<"stack is empty\n";
   return;}
 for(i=0;i<=p3->top;i++)  {
   cout<<p3->a[i];
 }
}
```

```
void main() {
 stack s;
 int c; s.top=-1;
 do  {
                cout<<"1:push\n2:pop\n3:display\n4:exit\n choice:";
                cin>>c;
                switch(c){
                case 1:
                                push(&s);
                                break;
                case 2:
                                pop(&s);
                                break;
                case 3:
                                display(&s);
                                break;
                case 4:
                                cout<<"pgm ends\n";
                                break;
                default:
                                cout<<"wrong choice\n";
                                break;
                }
                }while(c!=4);
}
```

A sample Program using struct

```
int push(stack *p1)
{
  int x;
  if(p1->top==max-1) //stack is full
   {
            cout<<"stack overflow\n";
            return -1;
  }
   p1->top++; //incr the top
    cout<<"enter a no\n";
    cin>>x;
    p1->a[p1->top]=x; //insert element
   cout<<"succ. pushed\n "<<x;
}
```

```
int pop(stack *p2)
{
  int y;
  if(p2->top==-1) // is Empty
  {
  cout<<"stack underflow\n";
  return -1;
  }
  y=p2->a[p2->top];
  p2->a[p2->top]=0;
  cout<<"succ. popped\n"<<y;
  p2->top--;
}
```

A SIMPLE STACK CLASS

```
class IntStack{
    private:
      int *stackArray;
      int stackSize;
      int top;
   public:
      IntStack(int);
      bool isEmpty();
      bool isFull();
      void push();
      void pop();
      void displayStack();
      void displayTopElement();
};
```

CONSTRUCTOR

```
IntStack::IntStack(int size)
{
   stackArray = new int[size];
   stackSize = size;
   top = -1;
}
```

PUSH()

```
void IntStack::push()
{
   int num;
   if(isFull()==true)
            cout<<"stack
            Overflow"<<endl;
   else
   {
   cout<<"Enter Number=";
   cin>>num;
   top++;
    stackArray[top]=num;
   }
}
```

POP()

```
void IntStack::pop()
{
   if(isEmpty()== true)
            cout<<"Stack
            Underflow"<<endl;
   else
   {
            cout<<"Number
            Deleted From the
            stack=";
   cout<<stackArray[top];
            top--;
   }
}
```

OTHER FUNCTIONS

```
bool IntStack::isEmpty()
{
if(top==-1)
    return true;
}
bool IntStack::isFull()
{
if(top>=stackSize)
    return true;
}
```

```
void IntStack::displayStack()
{
for(int i=top;i>=0;i--)
cout<<stackArray[i]<<endl;
}


void IntStack::displayTopElement()
{
cout<<stackArray[top]<<endl;
}
```

MAIN( )

```
void main ()
{
  IntStack stack(5);
   int choice;
   do
   {
      cout<<"Menu"<<endl;
     cout<<"1-- PUSH"<<endl;
     cout<<"2-- POP"<<endl;
     cout<<"3-- DISPLAY "<<endl;
     cout<<"4– TOP Element"<<endl;
     cout<<"5-- Exit"<<endl;
     cout<<"Enter choice=";
     cin>>choice;
```

```
switch(choice)
{
     case 1:
        stack.push();  break;
     case 2:
        stack.pop();  break;
     case 3:
        stack.displayStack(); break;
     case 4:
        stack.displayTopElement();
        break;
      case 5:
        break;
 }
}while(choice!=5);
  _getch();
}
```

Stack applications

- “Back” button of Web Browser
  - History of visited web pages is pushed onto the stack and popped when “back” button is clicked
- “Undo” functionality of a text editor
- Reversing the order of elements in an array
- Saving local variables when one function calls another, and this one calls another, and so on.

Example of Stack Applications in programming

Converting Decimal to Binary:

Consider the following pseudocode

```
Read (number)
Loop (number > 0)
        digit = number modulo 2
        print (digit)
        number = number / 2
```

The problem with this code is that it will print the binary number backwards. (ex: 19 becomes 11001000 instead of 00010011.) To remedy this problem, instead of printing the digit right away, we can push it onto the stack. Then after the number is done being converted, we pop the digit out of the stack and print it.

Algorithm

1. Create STACK.
2. Read Num.
3. Repeat while NUM > 0
    - 3.1 SET Digit := Num % 2.
    - 3.2 If STACK is full, then:
        - 3.2.1 Print “Stack Overflow”
        - 3.2.2 Return.
    - 3.3 PUSH(STACK, Digit).
    - 3.5 SET Num := Num/2.
4. Repeat while STACK is not empty.
    - 4.1 POP(STACK, Digit).
    - 4.2 Write Digit.
5. Return.

Expression evaluation and syntax parsing

- Many mathematical statements contain nested parenthesis like :-
  - (A+(B*C)) + (C – (D + F))
- We have to ensure that the parenthesis are nested correctly, i.e. :-
  1. There is an equal number of left and right parenthesis
  2. Every right parenthesis is preceded by a left parenthesis
- Expressions such as ((A + B) violate condition 1
- And expressions like) A + B (- C violate condition 2

Expression evaluation and syntax parsing

- To solve this problem, think of each left parenthesis as opening a scope, right parenthesis as closing a scope
- Nesting depth at a particular point in an expression is the number of scopes that have been opened but not yet closed
- Let “parenthesis count” be a variable containing number of left parenthesis minus number of right parenthesis, in scanning the expression from left to right

EXPRESSION EVALUATION AND SYNTAX PARSING

- For an expression to be of a correct form following conditions apply
  - Parenthesis count at the end of an expression must be 0
  - Parenthesis count should always be non-negative while scanning an expression
- Example :
  - Expr: 7 – (A + B) + ((C – D) + F)
  - ParenthesisCount: 0 0 1 1 1 1 0 0 1 2 2 2 2 1 1 1 0
  - Expr: 7 – ((A + B) + ((C – D) + F)
  - ParenthesisCount: 0 0 1 2 2 2 2 1 1 23 3 3 3 2 2 21

Expression evaluation and syntax parsing

- Evaluating the correctness of simple expressions like this one can easily be done with the help of a few variables like “Parenthesis count”
- Things start getting difficult to handle by your program when the requirements get complicated e.g.
- Let us change the problem by introducing three different types of scope de-limiters i.e. (parenthesis), {braces} and [brackets].
- In such a situation we must keep track of not only the number of scope delimiters but also their types
- When a scope ender is encountered while scanning an expression, we must know the scope delimiter type with which the scope was opened
- We can use a stack ADT to solve this problem

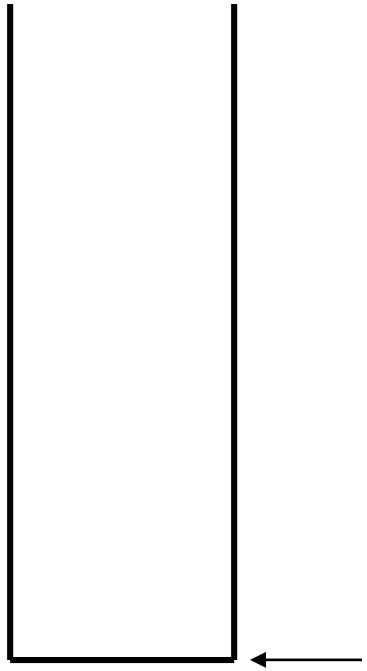
Expression evaluation and syntax parsing

- A stack ADT can be used to keep track of the scope delimiters encountered while scanning the expression
- Whenever a scope “opener” is encountered, it can be “pushed” onto a stack
- Whenever a scope “ender” is encountered, the stack is examined:
  - If the stack is “empty”, there is no matching scope “opener” and the expression is invalid.
  - If the stack is not empty, we pop the stack and check if the “popped” item corresponds to the scope ender
  - If match occurs, we continue scanning the expression
- When end of the expression string is reached, the stack must be empty, otherwise one or more opened scopes have not been closed and the expression is invalid

Expression evaluation and syntax parsing

- Last scope to be opened must be the first one to be closed.
- This scenario is simulated by a stack in which the last element arriving must be the first one to leave
- Each item on the stack represents a scope that has been opened but has yet not been closed
- Pushing an item on to the stack corresponds to opening a scope
- Popping an item from the stack corresponds to closing a scope, leaving one less scope open

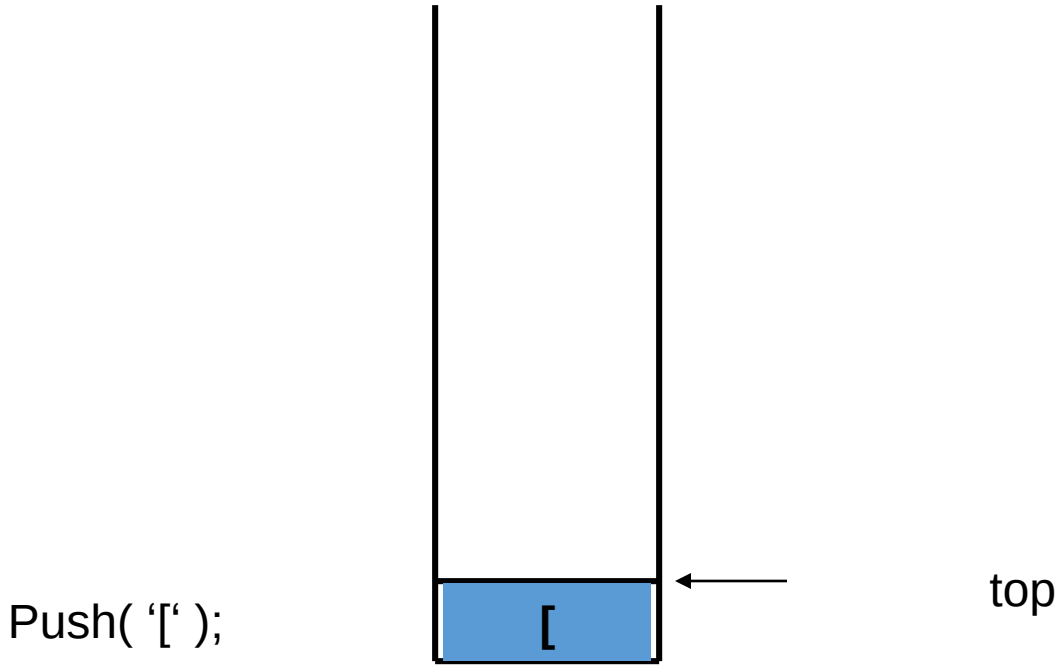
Stack in Action ….



top

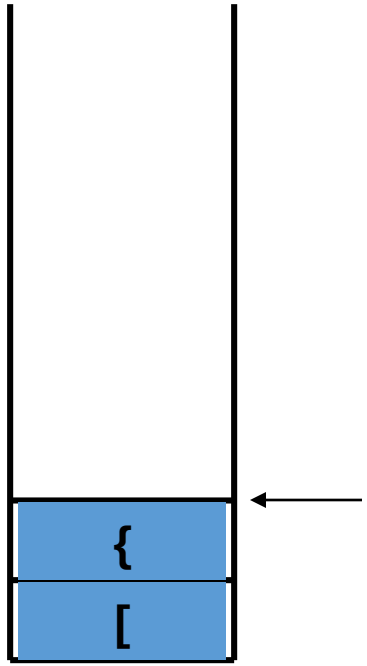
[A + { B - C + (D + E) }]

Stack in Action ….



[A + { B - C + (D + E) }]

Stack in Action ….

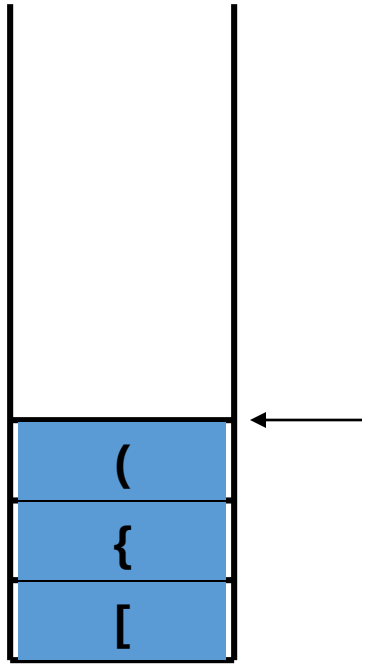


top

Push(‘{‘);

[A + { B - C + (D + E) }]

Stack in Action ….

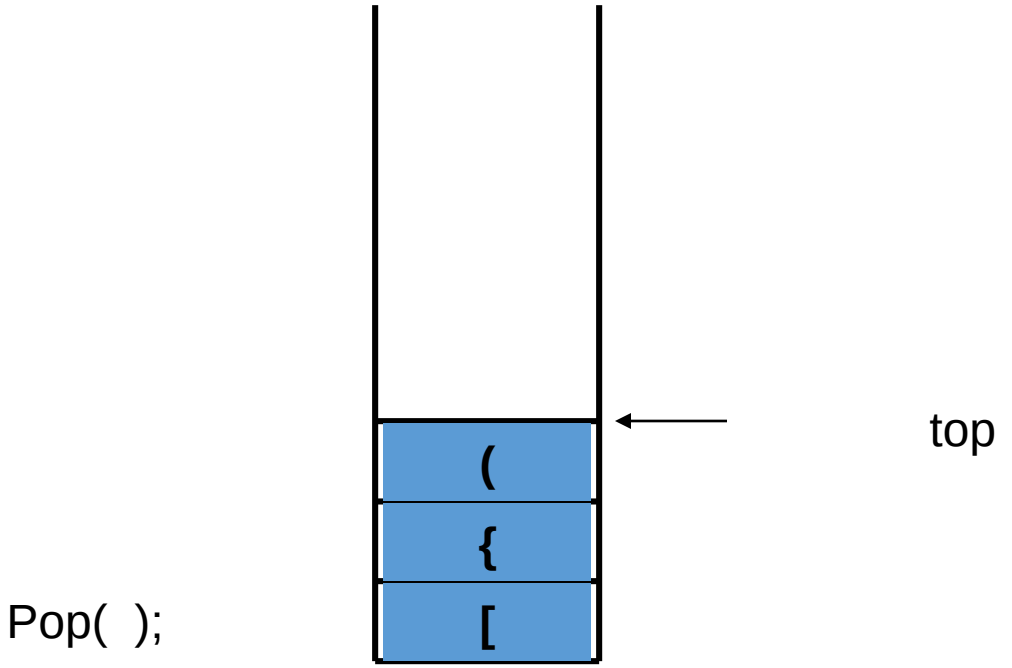


top

Push(‘(‘);

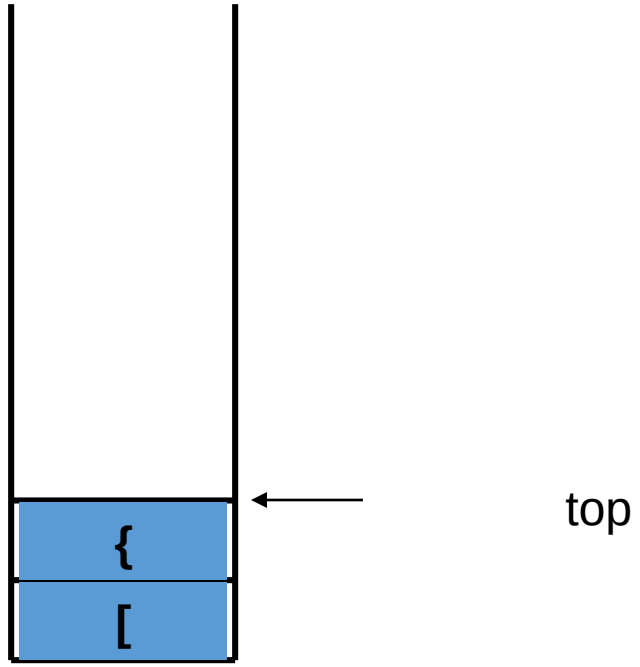
[A + { B - C + (D + E) }]

Stack in Action ….



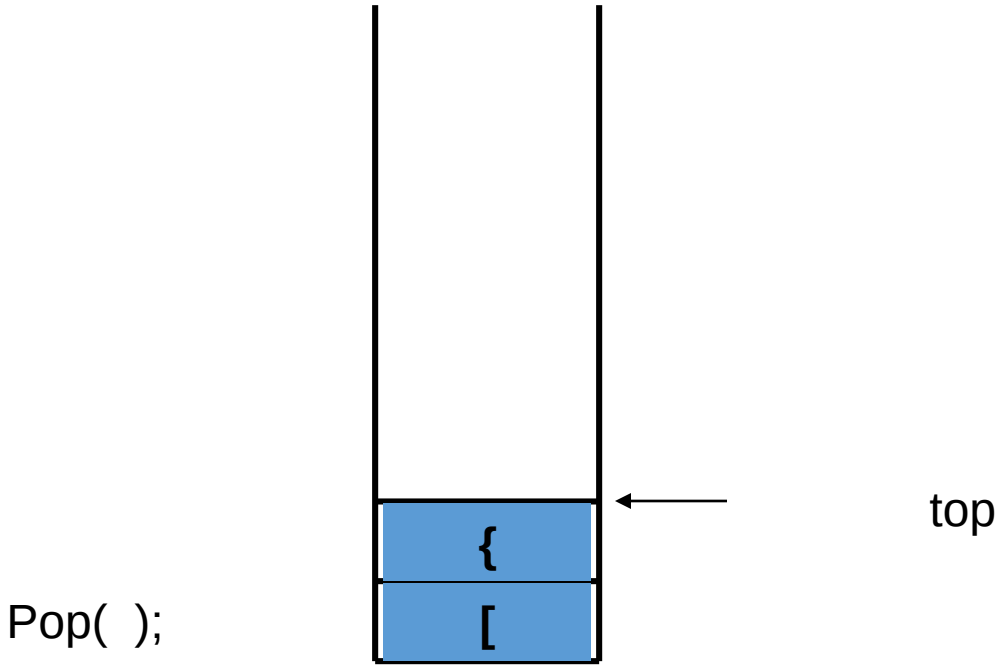
[A + { B - C + (D + E) }]

Stack in Action ….



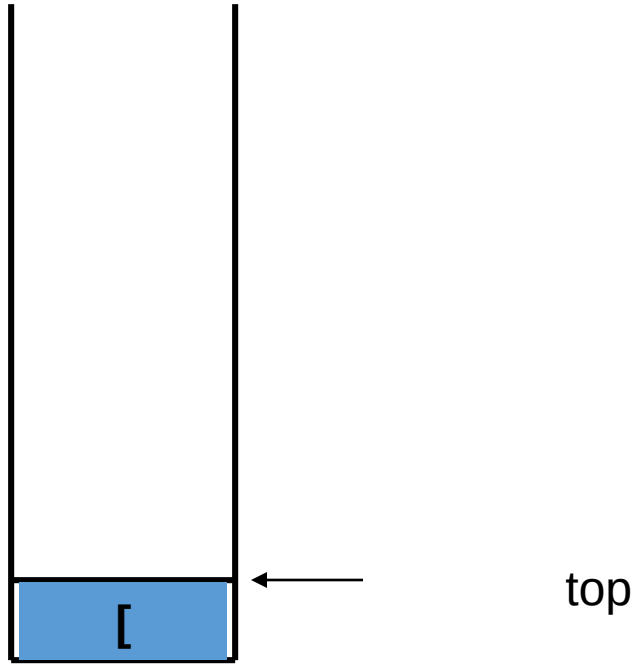
[A + { B - C + (D + E) }]

Stack in Action ….



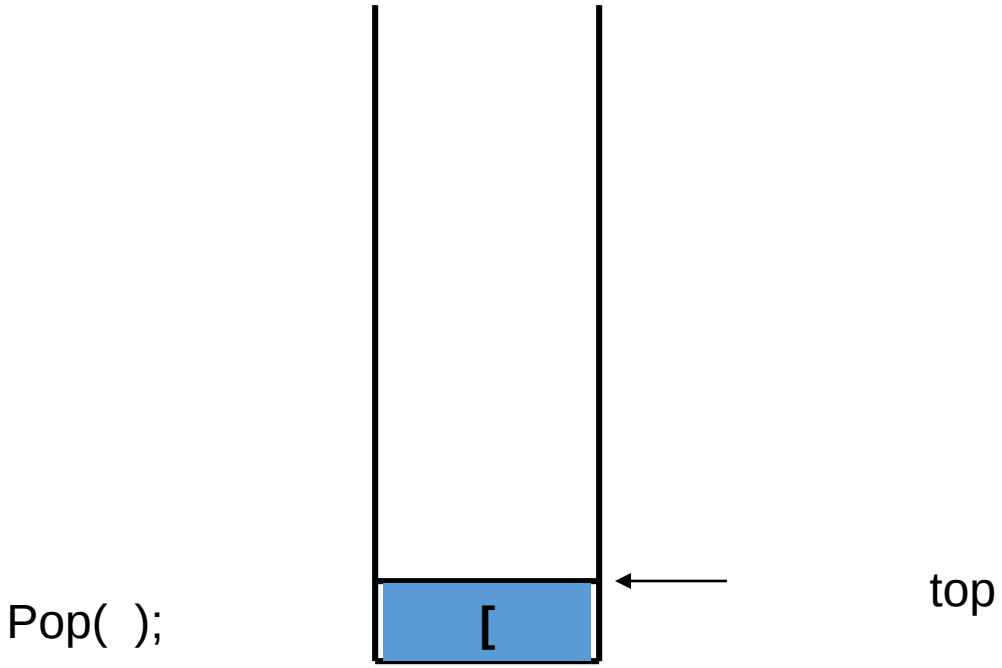
[A + { B - C + (D + E) }]

Stack in Action ….



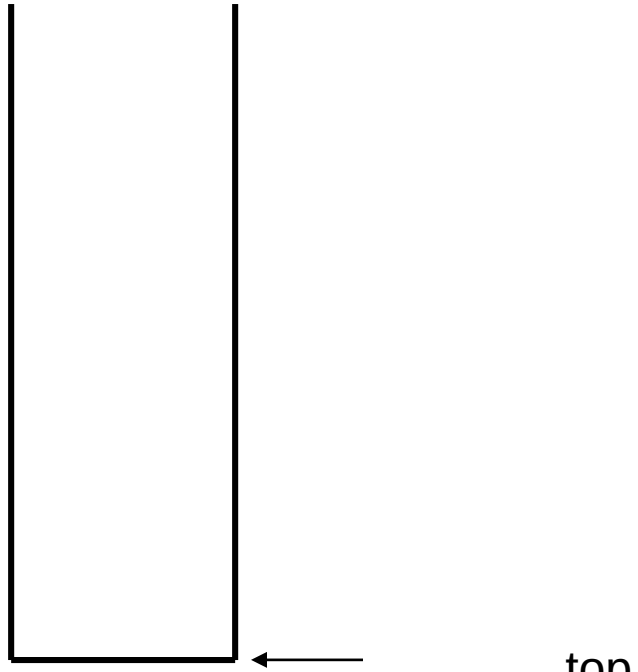
[A + { B - C + (D + E) }]

Stack in Action ….



[A + { B - C + (D + E) }]

Stack in Action ….

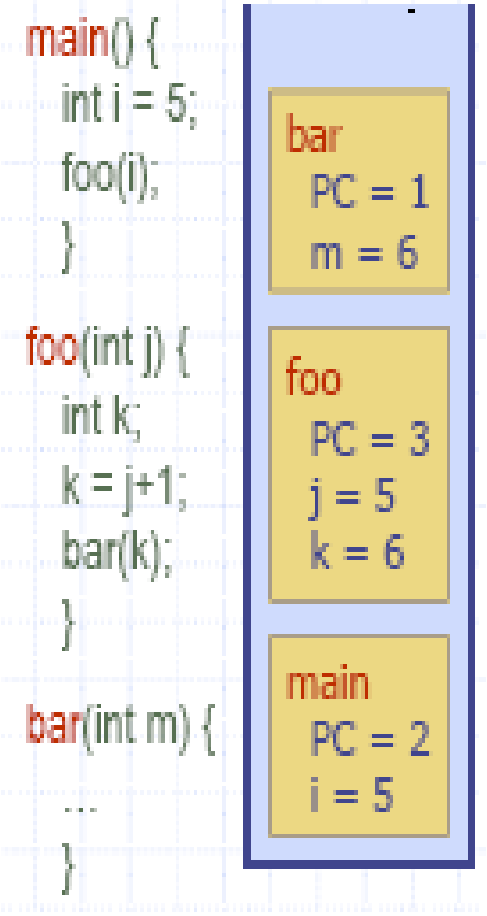


[A + { B - C + (D + E) }]]

Result = A valid expression

Run-time Memory management

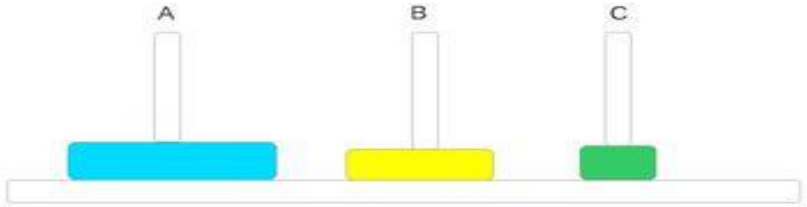
- The C++ run-time system keeps track of the chain of active functions with a stack
- When a function is called, the run-time system pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a function returns, its frame is popped from the stack and control is passed to the method on top of the stack



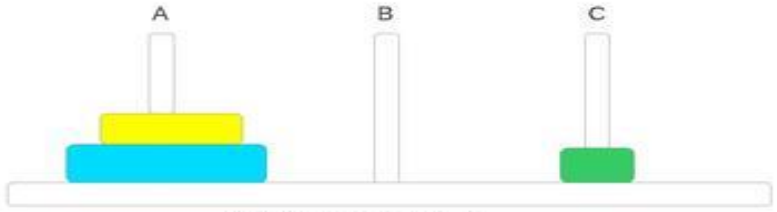
Other applications (Reading Assignment)

- Towers of Hanoi

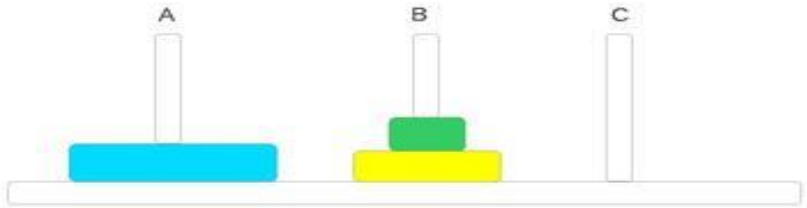
Initial step



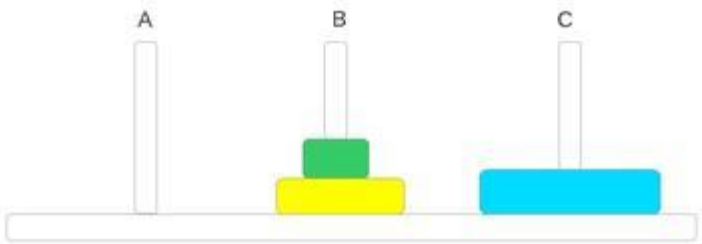
After moving from A to B



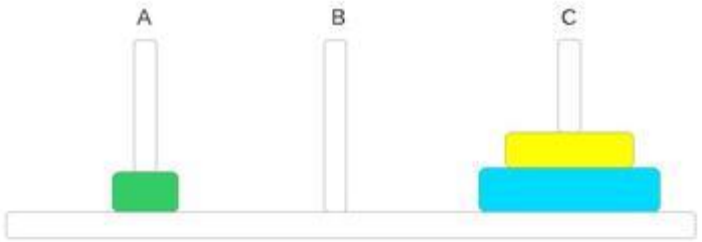
After moving from A to C



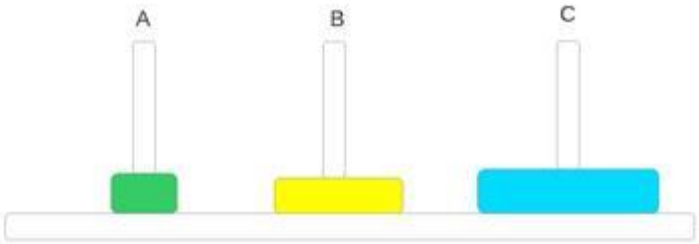
After moving from C to B



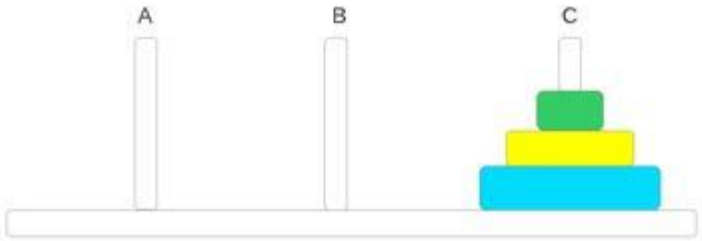
After moving from A to C



After moving from B to C



After moving from B to A



After moving from A to C

Other applications (Reading Assignment)

- **Bactracking**
- **Rearranging railroad cars**
- **Conversion of an Infix expression that is fully parenthesized into a Postfix expression**
- **Quicksort**