CS 2124: DATA STRUCTURES
Spring 2024

Third Lecture (Part – 2)

Topics: Creating Libraries, Stacks

Topics

- Library (Review)
- Stacks
  - Stack Operations
  - Stack vs Array
  - Example/ Implementation
  - Stack implementation using linked-list
  - Exploiting stack buffer overflows
- Infix, Prefix and Postfix Expressions
  - Infix-to-Postfix Conversion
  - Infix-to-Postfix Conversion (using stacks)
  - Postfix Evaluation

Library (Review)

- Advantages of Using C library functions
  - The functions are optimized for performance
  - It saves considerable development time
  - The functions are portable
  - Flexibility in modification
  - Abstraction (reduce complexity and improve security)

Lets see how this plays a role in real world implementation >>>
Next Slide

Design Principles

National Centers of Academic Excellence in Cybersecurity

NCAE-C 2022

Designation Requirements and Application Process

For

CAE Cyber Operations (CAE-CO)

- The **National Security Agency (NSA)** has designated 11 design principles as part of the **Cyber Operations Centers for Academic Excellence (CAE-CO)**
- Their purpose:
  - Help programmers develop more secure code
  - When followed, code should:
    - Contain fewer errors (often a source of security vulnerabilities)
    - Be designed with security built in (Rather then designing additional modules for security)
- These principles are based on the previous work by **Saltzer and Schroeder**

Source: Link
PDF of NCAE-C 2022: Link

The Eleven Design Principles

- General/Fundamental Design Principles
  1. Simplicity (related to Economy of Mechanism but not exactly the same)
  2. Open Design
  3. Design for Iteration (not specifically identified by Saltzer and Schroeder)
  4. Least Astonishment (related to Psychological Acceptability)
- Security Design Principles
  5. Minimize Secrets (not specifically identified by Saltzer and Schroeder)
  6. Complete Mediation
  7. Fail-safe Defaults
  8. Least Privilege
  9. Economy of Mechanism
  10. Minimize Common Mechanism (related to Least Common Mechanism)
  11. Isolation, Separation and Encapsulation

Methods for Reducing Complexity

- Goes along with the **Simplicity** Design Principle

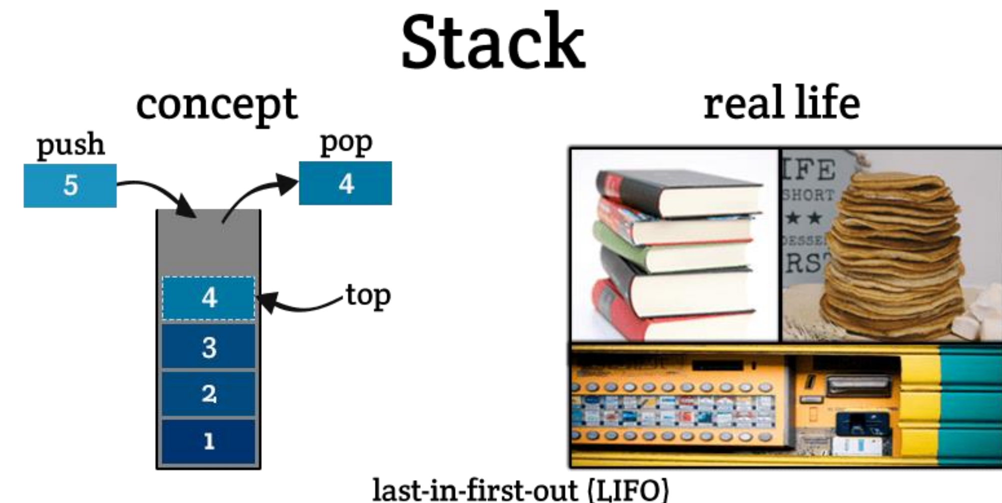
- They are:

1. Abstraction
   - Hide background details or any unnecessary implementation about the data so that users only see the required information.
2. Modularity
   - Making building blocks, and even building blocks made up of smaller building blocks
3. Layering
   - Layer distinction is often expressed as "import" dependencies between software modules
4. Hierarchy
   - Organizational structure in which items are ranked in a specific manner, usually according to levels of importance

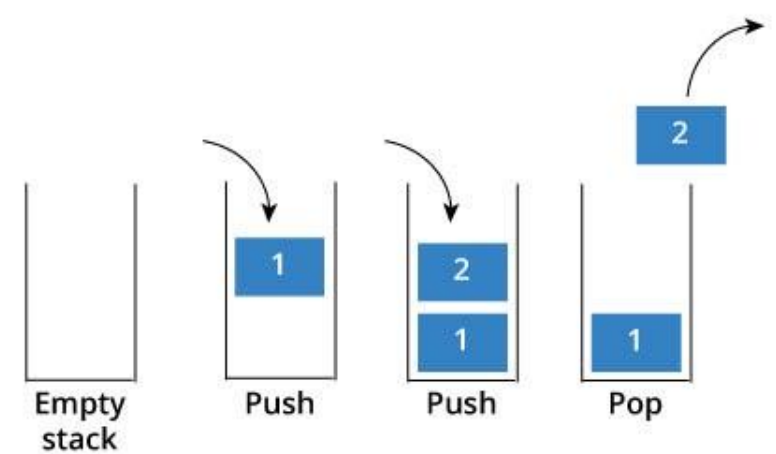
Stack

- What is a stack?
  - A data structure (i.e. stores data). New items are inserted on the “top” and deleted or removed from the top. Top is the most recent item inserted
  - A stack is a Last In First Out (LIFO) or First In Last Out (FILO) data structure. This means that the last item to get stored on the stack (often called Push operation) is the first one to get out of it (often called as Pop operation).

- Where are stacks used?
  - Call stacks in program execution (structures programming)
  - Undo buttons
  - Back button in a browser/phone
  - Implement certain math notations (infix, prefix, postfix)
  - Expression Evaluation, Expression Conversion
  - Memory Management



Stack Operations



1. **push:** Adds an element to the top of the stack.
2. **pop:** Removes the topmost element from the stack.
3. **isEmpty:** Checks whether the stack is empty.
4. **isFull:** Checks whether the stack is full.
5. **top:** Displays the topmost element of the stack.
6. **Peek**: View the top element on the stack

- Important thing to remember when working with 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.

Stack Operations

<u>PUSH:</u>

1. begin
2. if stack is full
3. return
4. endif
5. else
6. increment top
7. stack[top] assign value
8. end else
9. end procedure

<u>POP:</u>

1. begin
2. if stack is empty
3. return
4. endif
5. else
6. store value of stack[top]
7. update top
8. return value
9. end else
10. end procedure

<u>TOP:</u>

1. begin
2. return stack[top]
3. end procedure

<u>ISFULL</u>

1. begin
2. if top equals to MAXSIZE
3. return true
4. else
5. return false
6. endif
7. end procedure

<u>ISEMPTY</u>

1. begin
2. if top < 1
3. return true
4. else
5. return false
6. end procedure

<u>PEEK (retrieve do not pop)</u>

1. begin procedure peek
2. return stack[top]
3. end procedure

There are **two ways to implement** a stack:

1. Using array (Easy to implement, memory efficient because of pointers, need to define array size)
2. Using linked list (Stack size flexibility, extra memory require, random accessing is not possible)

Stack pop vs top vs peek operations

- The only difference between peek and pop is that the peek method just returns the topmost element; however, in the case of a pop method, the topmost element is returned and also that element is deleted from the stack.

- top() - only returns the element but doesn't remove it.
- pop() - only removes the element but doesn't return anything.

Stack vs Array

Basis of Comparison	Stacks	Array
Definition	Stack is a linear data structure represented by a sequential collection of elements in a fixed order	An array is a collection of related data values called elements each identified by an indexed array
Principle	Stacks are based on the LIFO principle	In the array the elements belong to indexes
Operations	Insertion and deletion in stacks take place only from one end of the list called the top.	Insertion and deletion in the array can be done at any index in the array.
Storage	The stack has a dynamic size.	The array has a fixed size.
Data Types	The stack can contain elements of different data types.	The array contains elements of the same data type.
Search	We can do only a linear search	We can do both linear and Binary search
Data Access	Random access to elements is not allowed in stacks	Random access to elements is allowed in arrays
Implementation	We can implement a stack using the array	We cannot implement an array using stack
Methods	There are limited number of operations can be performed on a stack: push, pop, peek, etc.	It is rich in methods or operations that can be perform on it like sorting, traversing, reverse, push, pop, etc.

Example (Stacks for checking parenthesis)

- Example: [({a+b})]

- Operations
  1. Push [
  2. Push (
  3. Push {
  4. } Pop
  5.) Pop
  6.] Pop

Parenthesis are balanced

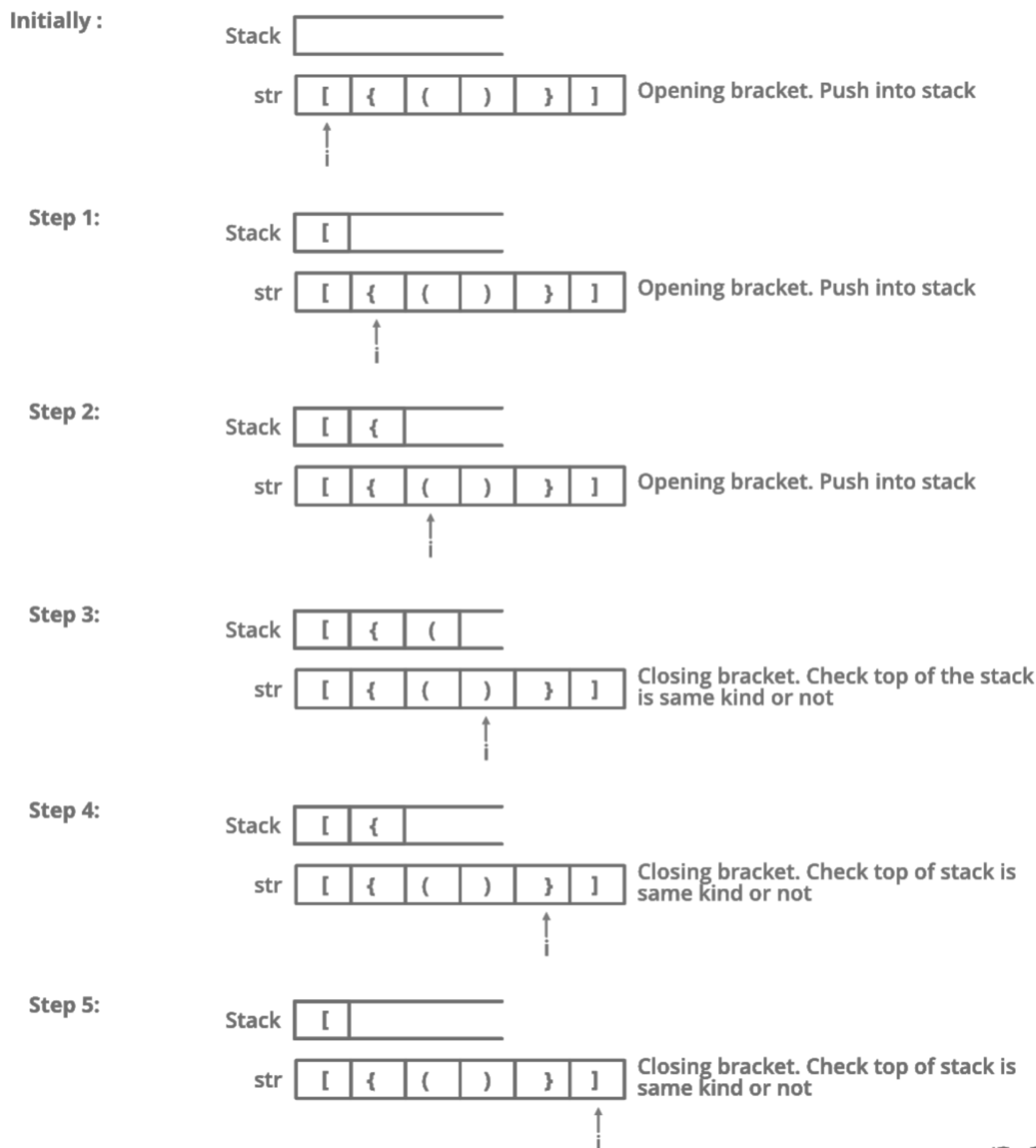
Example

- Equation: [({A * B} + (C / D)]

- Operations
  1. Push [
  2. Push (
  3. Push {
  4. } Pop
  5. Push (
  6.) Pop
  7. ?

- Missing ‘)’
- Sequence is also important

Stack	[	([	{ ([	([	(([	([	
Steps	1	2	3	4	5	6	7



1. Declare a character stack (say **temp**).
2. Now traverse the string exp.
   I. If the current character is a starting bracket (**'(' or '{' or '['**) then push it to stack.
   II. If the current character is a closing bracket (**')' or '}' or ']'**) then pop from stack and if the popped character is the matching starting bracket then fine.
   III. Else brackets are **Not Balanced**.
3. After complete traversal, if there is some starting bracket left in stack then **Not balanced**, else **Balanced**.

Example (Try to solve it your self)

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

- Steps:

Stack										
Steps	1	2	3	4	5	6	7	8	9	10

Example (Stack implementation 1/2)

```c
#include <stdio.h>
int MAXSIZE = 8;
int stack[8];
int top = -1;

/* Check if the stack is empty */
int isempty(){
    if(top == -1)
        return 1;
    else
        return 0;
}

/* Check if the stack is full */
int isfull(){
    if(top == MAXSIZE)
        return 1;
    else
        return 0;
}
```

```c

/* Function to return the topmost element in the stack */
int peek(){
    return stack[top];
}
```

```c

/* Function to delete from the stack */
int pop(){
    int data;
    if(!isempty()) {
        data = stack[top];
        top = top - 1;
        return data;
    } else {
        printf("Could not retrieve data, Stack is empty.\n");
    }
}
```

Continue >>>

(Stack implementation 2/2)

```
38
39  /* Function to insert into the stack */
40  int push(int data){
41      if(!isfull()) {
42          top = top + 1;
43          stack[top] = data;
44      } else {
45          printf("Could not insert data, Stack is full.\n");
46      }
47  }
```

```
49  /* Main function */
50  int main(){
51      push(3);
52      push(2);
53      push(0);
54      push(2);
55      printf("Element at top of the stack: %d\n" ,peek());
56      printf("Elements: \n");
57
58      // print stack data
59      while(!isempty()) {
60          int data = pop();
61          printf("%d\n",data);
62      }
63      printf("Stack full: %s\n" , isfull()?"true":"false");
64      printf("Stack empty: %s\n" , isempty()?"true":"false");
65      return 0;
66  }
```

Advantages of array implementation:

- Easy to implement.
- Memory is saved as pointers are not involved.

Disadvantages of array implementation:

- It is not dynamic i.e., it doesn’t grow and shrink depending on needs at runtime.
- The total size of the stack must be defined beforehand.

Example (Stack implementation using linked-list)

```
1.  struct StackNode
2.  {
3.          int data;
4.          struct StackNode* next;
5.  };
6.  struct StackNode* newNode(int data)
7.  {
8.          struct StackNode* stackNode = (struct StackNode*)
9.          malloc(sizeof(struct StackNode));
10.         stackNode->data = data;
11.         stackNode->next = NULL;
12.         return stackNode;
13. }
```

```
1.  int main()
2.  {
3.          struct StackNode* root = NULL;
4.          push(&root, 10);
5.          push(&root, 20);
6.          push(&root, 30);
7.          printf("%d popped from stack\n", pop(&root));
8.          printf("Top element is %d\n", peek(root));
9.          return 0;
10. }
```

Advantages of Linked List implementation:

- The linked list implementation of a stack can grow and shrink according to the needs at runtime.
- It is used in many virtual machines

Disadvantages of Linked List implementation:

- Requires extra memory due to the involvement of pointers.
- Random accessing is not possible in stack.

Example (Stack implementation using linked-list)

```c
#include <limits.h>
#include <stdio.h>
#include <stdlib.h>
// A structure to represent a stack
struct StackNode {
    int data;
    struct StackNode* next;
};
struct StackNode* newNode(int data)
{
    struct StackNode* stackNode =
    (struct StackNode*)
    malloc(sizeof(struct StackNode));
    stackNode->data = data;
    stackNode->next = NULL;
    return stackNode;
}
int isEmpty(struct StackNode* root)
{
    return !root;
}
```

```c
void push(struct StackNode** root, int data)
{
    struct StackNode* stackNode = newNode(data);
    stackNode->next = *root;
    *root = stackNode;
    printf("%d pushed to stack\n", data);
}
int pop(struct StackNode** root)
{
    if (isEmpty(*root))
        return INT_MIN;
    struct StackNode* temp = *root;
    *root = (*root)->next;
    int popped = temp->data;
    free(temp);
    return popped;
}
int peek(struct StackNode* root)
{
    if (isEmpty(root))
        return INT_MIN;
    return root->data;
}
int main()
{
    struct StackNode* root = NULL;
    push(&root, 10);
    push(&root, 20);
    push(&root, 30);
    printf("%d popped from stack\n", pop(&root));
    printf("Top element is %d\n", peek(root));
    return 0;
}
```

Example (Stack implementation using linked-list)

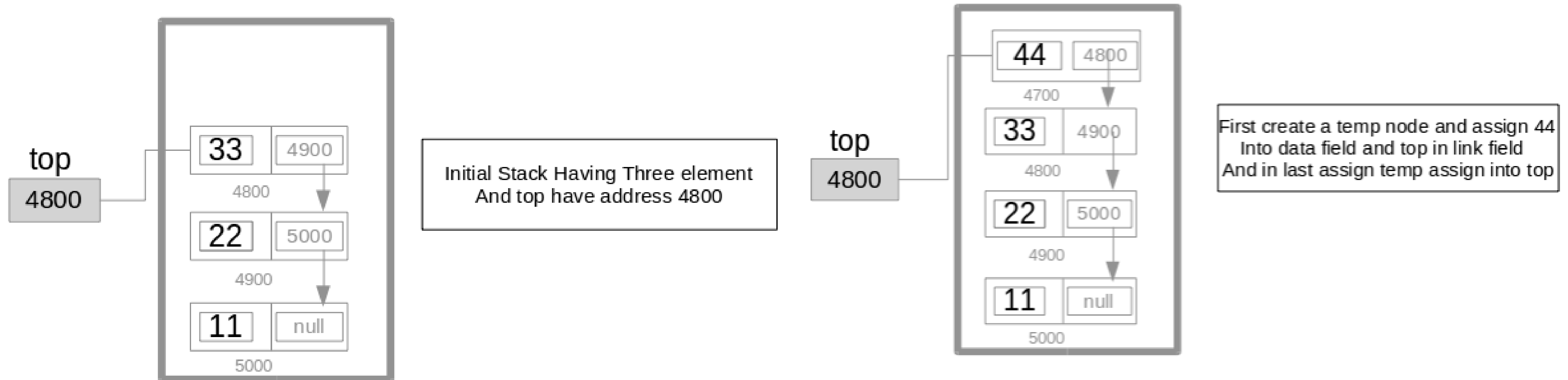


Image Source: [Link]()

Stacks

- A Stack is a linear abstract data structure.
- The stack data structure can be of two types: static and dynamic stack.
- A static stack is implemented using an array in C, whereas a dynamic stack is implemented using a linked list in C.
- Push and Pop are the two primary operations of a stack, others being Peek, isEmpty and isFull.
- Stack can be used in expression conversion, like, infix to postfix, infix to prefix, postfix to infix, and prefix to infix.
- The time complexity of all the operations of a Stack is O(1).

Infix, Prefix and Postfix Expressions

- When you write an arithmetic expression such as B * C, the form of the expression provides you with information so that you can interpret it correctly.
- In this case we know that the variable B is being multiplied by the variable C since the multiplication operator * appears between them in the expression.
- This type of notation is referred to as **infix** since the operator is in between the two operands that it is working on.

- Consider another infix example, A + B * C. The operators + and * still appear between the operands, but there is a problem.
- Which operands do they work on?
- Does the + work on A and B or does the * take B and C?

1. Apply + First
2. Apply * First
3. Order of operators do not matter

Infix, Prefix and Postfix Expressions

- When you write an arithmetic expression such as B * C, the form of the expression provides you with information so that you can interpret it correctly.
- In this case we know that the variable B is being multiplied by the variable C since the multiplication operator * appears between them in the expression.
- This type of notation is referred to as **infix** since the operator is in between the two operands that it is working on.

- Consider another infix example, A + B * C. The operators + and * still appear between the operands, but there is a problem. Which operands do they work on? Does the + work on A and B or does the * take B and C?

*Let's interpret the troublesome expression A + B * C using operator precedence. B and C are multiplied first, and A is then added to that result. (A + B) * C would force the addition of A and B to be done first before the multiplication.*

Infix, Prefix and Postfix Expressions

- **Fully parenthesized expression (Computer need to know the order of operations)**
- The expression A + B * C + D can be rewritten as ((A + (B * C)) + D) to show that the multiplication happens first, followed by the leftmost addition. A + B + C + D can be written as (((A + B) + C) + D) since the addition operations associate from left to right.
- Consider the **infix** expression A + B.
  - **Prefix** expression notation requires that all operators precede the two operands that they work on.
  - **Postfix,** on the other hand, requires that its operators come after the corresponding operands.

Infix Expression	Prefix Expression	Postfix Expression
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +

We will focus on Postfix and infix

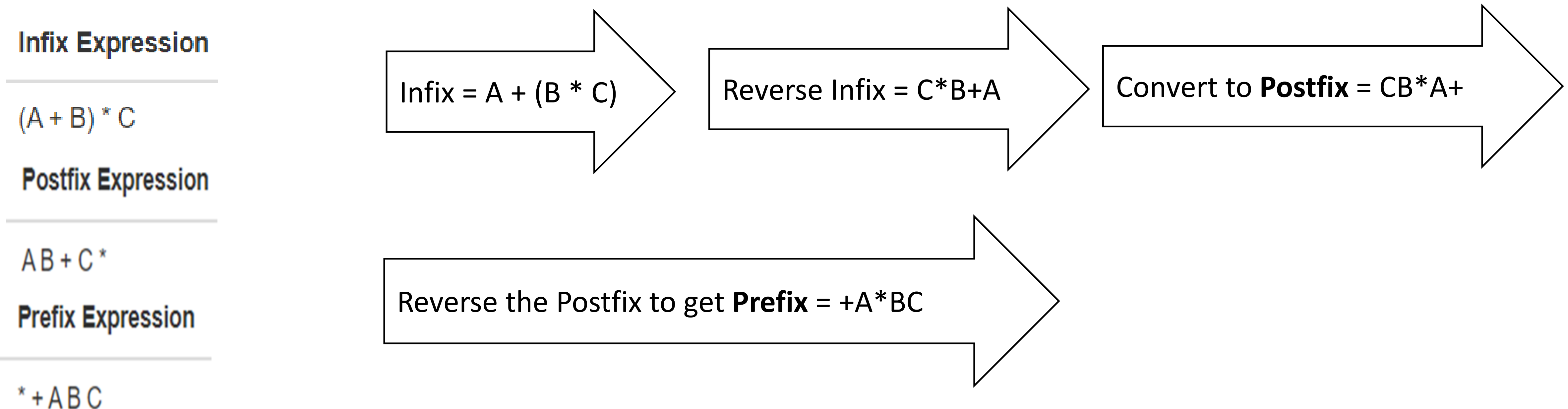
Infix, Prefix and Postfix Expressions

- **Infix** expression A + B * C
- A + B * C would be written as + A * B C in **prefix**.
  - The multiplication operator comes immediately before the operands B and C, denoting that * has precedence over +. The addition operator then appears before the A and the result of the multiplication.
- In **postfix,** the expression would be A B C * +.
  - Again, the order of operations is preserved since the * appears immediately after the B and the C, denoting that * has precedence, with + coming after.

Infix Expression	Prefix Expression	Postfix Expression
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +

Infix, Prefix and Postfix Expressions

- Now consider the infix expression **(A + B) * C.**
  - In this case, **infix** requires the parentheses to force the performance of the addition before the multiplication.
  - However, when A + B was written in **prefix**, the addition operator was simply moved before the operands, + A B. The result of this operation becomes the first operand for the multiplication. The multiplication operator is moved in front of the entire expression, giving us * + A B C.
  - Likewise, in **postfix** A B + forces the addition to happen first. The multiplication can be done to that result and the remaining operand C. The proper postfix expression is then A B + C *.



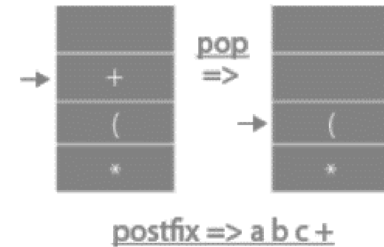
Infix-to-Postfix Conversion (Stacks)

- A*(b+c+d)

- Step 1:
  - Postfix = a
  - Stack = | * |
- Step 2:
  - Postfix = a
  - Stack = | (| * |
- Step 3:
  - Postfix = a b
  - Stack = | (| * |
- Step 4:
  - Postfix = a b
  - Stack = | + | (| * |
- Step 5:
  - Postfix = a b c
  - Stack = | + | (| * |
- Step 6:
  - Postfix = a b c +
  - Stack = | | (| * |

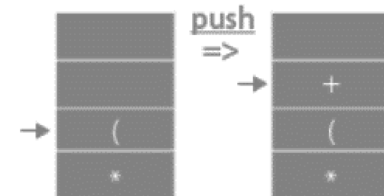
=> '+' is an operator, so push it into stack after removing higher or equal priority operators from top of stack.

=> Current top of stack is '+', it has equal precedence with with input symbol '+'. So, pop it and append to postfix.



- Step 7:
  - Postfix = a b c + d
  - Stack = | + | (| * |

=> Now the top of stack is '(', so push the input operator.

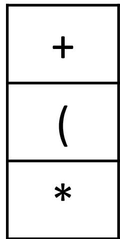


Source: Link

Infix-to-Postfix Conversion (Stacks)

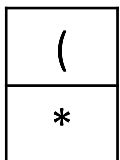
- A*(b+c+d)

- Step 7:
  - Postfix = a b c + d
  - Stack =



')' is an right paranthesis, flow step n of algorithm and pop all content of stack until respective '(' is popped and append to the output.

- Step 8:
  - Postfix = a b c + d +
  - Stack =



- Step 9:
  - Postfix = a b c + d + *
  - Stack =

General Infix-to-Postfix Conversion

- Consider the expression A + B * C.
- A B C * + is the **postfix** equivalent.
  - The operands A, B, and C stay in their relative positions.
  - It is only the operators that change position.
- In the infix expression the first operator that appears from left to right is +.
- However, in the postfix expression, + is at the end since the next operator, *, has precedence over addition.
- The order of the operators in the original expression is reversed in the resulting postfix expression.

Input String	Output Stack	Operator Stack
A+ B*C	A	
A+ B*C	A	+
A+ B*C	A	+
A+ B*C	A B	+
A+ B*C	A B	+*
A+ B*C	A BC	+*
A+ B*C	A BC*+	

General Infix-to-Postfix Conversion

- Conversion algorithm working on the expression **A * B + C * D.**
  - Note that the first * operator is removed upon seeing the + operator.
  - Also, + stays on the stack when the second * occurs, since multiplication has precedence over addition.
  - At the end of the infix expression the stack is popped twice, removing both operators and placing + as the last operator in the postfix expression.

Input String	Output Stack	Operator Stack
A * B + C * D	A	
A * B + C * D	A	
A * B + C * D	A	*
A * B + C * D	A	*
A * B + C * D	A B	*
A * B + C * D	A B	*
A * B + C * D	A B *	+
A * B + C * D	A B *	+
A * B + C * D	A B * C	+
A * B + C * D	A B * C	+
A * B + C * D	A B * C	+*
A * B + C * D	A B * C	+*
A * B + C * D	A B * C D	+*
A * B + C * D	A B * C D*+	

General Infix-to-Postfix Conversion (Steps)

1. Read from left-to-right
   i. Append operands to postfix string
2. If we find an operator:
   i. If precedence function returns true, append previously stacked operators to postfix string.
   ii. Once the higher precedence are appended, push our new operator on stack. (Discard a ‘)’ operator instead).
3. Repeat until we reach the end of the input.
4. Pop the remaining symbols in the operator stack and join them to the postfix string.

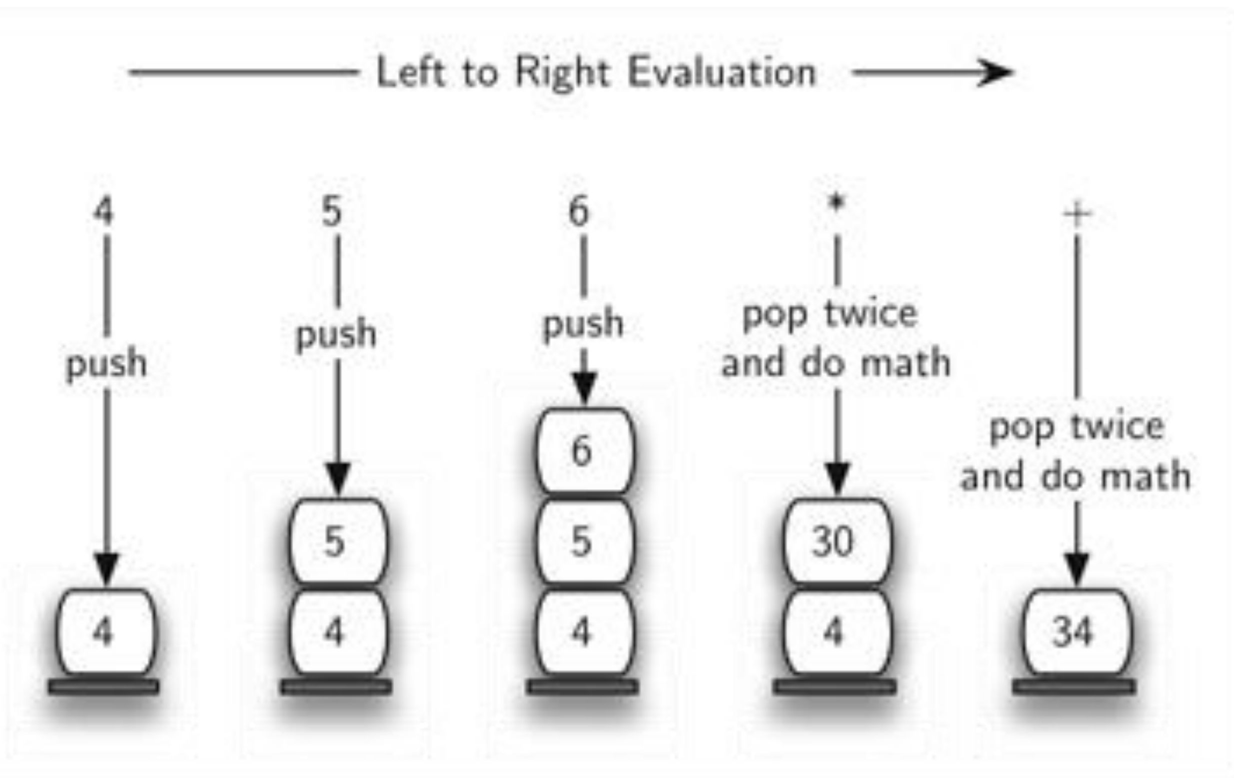
Infix-to-Postfix Conversion (using stacks)

```c
1  #include<stdio.h>
2  #include<ctype.h>
3  char stack[10];
4  int top = -1;
5  void push(char x)
6  {
7      stack[++top] = x;
8  }
9  char pop()
10 {
11     if(top == -1)
12         return -1;
13     else
14         return stack[top--];
15 }
16 int priority(char x)
17 {
18     if(x == '(')
19         return 0;
20     if(x == '+' || x == '-')
21         return 1;
22     if(x == '*' || x == '/')
23         return 2;
24     return 0;
25 }
```

```c
26 int main()
27 {
28     char exp[100];
29     char *e, x;
30     printf("Enter the expression : ");
31     scanf("%s",exp);
32     printf("\n");
33     e = exp;
34         while(*e != '\0')
35     {           if(isalnum(*e))
36             printf("%c ",*e);
37         else if(*e == '(')
38             push(*e);
39         else if(*e == ')')
40         {               while((x = pop()) != '(')
41                 printf("%c ", x);
42         }           else
43         {    while(priority(stack[top]) >= priority(*e))
44                 printf("%c ",pop());
45             push(*e);
46         }           e++;
47     }
48         while(top != -1)
49     {       printf("%c ",pop());
50     }return 0;
51 }
```

Postfix Evaluation

- Infix expression: 4+5*6
- Postfix expression: 4 5 6 * +



1. 4 is pushed in stack
2. + is kept as an operation
3. 5 is pushed in the stack
4. * is kept as operation
5. 6 is pushed in stack
6. * is the first operation that need to be executed so
   - 5*6 = 30
7. + is the next operation
   - 30+4=34

Postfix Evaluation

- Postfix expression: 7 8 + 3 2 + /.
- What will be the answer if you solve the equation?