Applications of stack

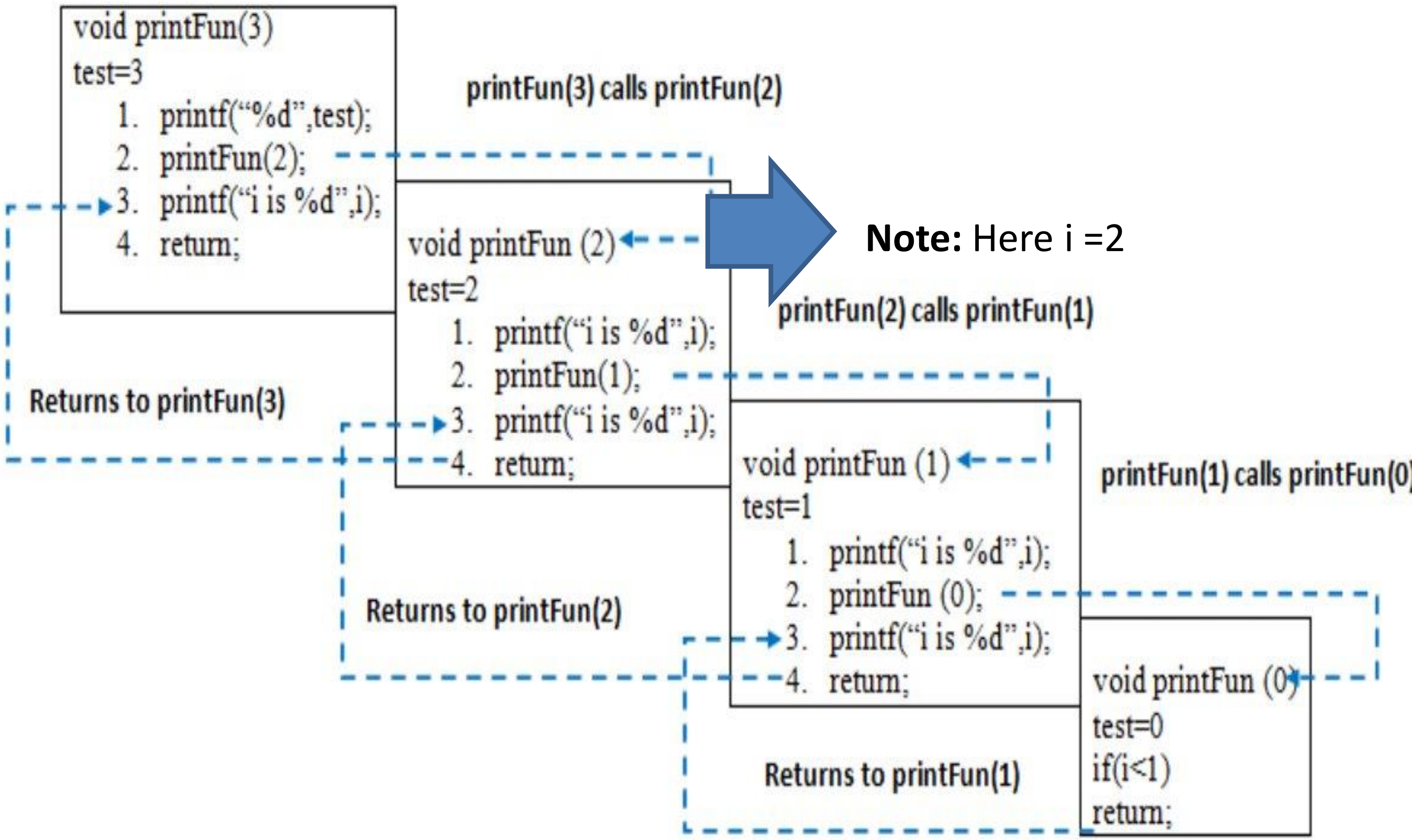
Recursion Example

When **printFun(3)** is called from main(), memory is allocated to **printFun(3)** and a local variable test is initialized to 3 and statement 1 to 4 are pushed on the stack as shown in below diagram. It first prints '3'. In statement 2, **printFun(2)** is called and memory is allocated to **printFun(2)** and a local variable test is initialized to 2 and statement 1 to 4 are pushed in the stack.

Similarly, **printFun(2)** calls **printFun(1)** and **printFun(1)** calls **printFun(0)**. **printFun(0)** goes to if statement and it return to **printFun(1)**. Remaining statements of **printFun(1)** are executed and it returns to **printFun(2)** and so on. In the output, value from 3 to 1 are printed and then 1 to 3 are printed. The memory stack has been shown in next slide.

Output: 3 2 1 1 2 3

Memory stack



Note: Here i =2

Output: 3 2 1 1 2 3

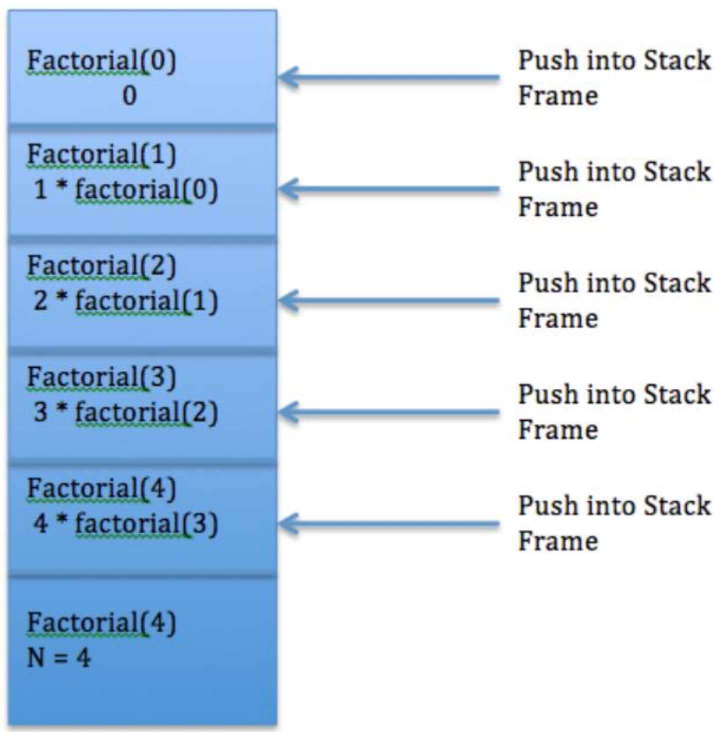
Example

Finding Factorial of a Number

```
long int count = 1;
long int factorial(long int n){
  count++;//to count how many times the method getting called
  if (n == 0) {
   return 1;
  }
  return n * factorial(n - 1);
}

int main(int argc, const char * argv[])
{
   int n = 13;
   printf("Factorial of %d = %lu , number of time called %lu\n",n,factorial(n),count);
   return 0;
}
```

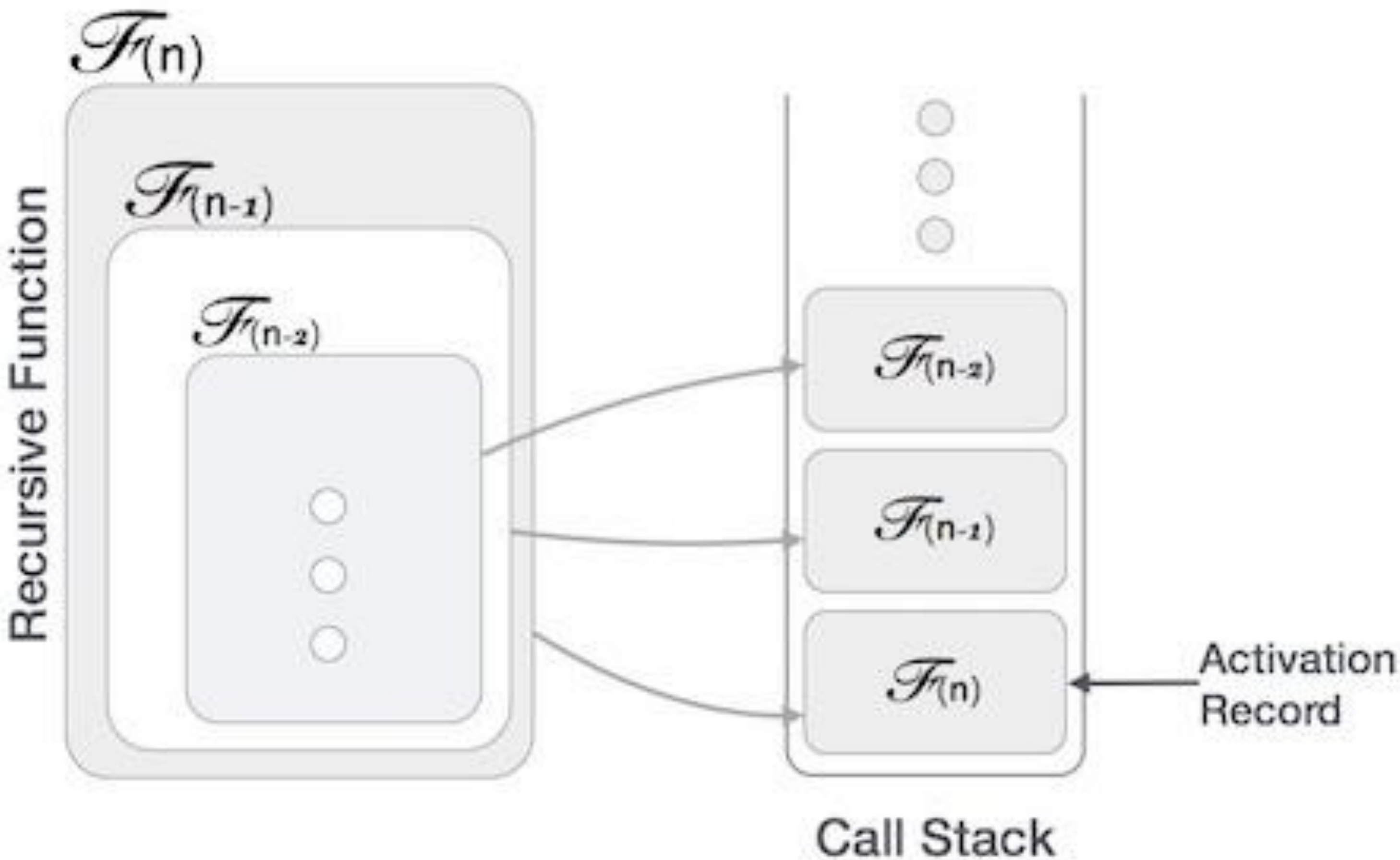
When factorial(n) will be called every time , a stack frame will be created and the call will be pushed to stack, the entire call stack looks like below. The below image shows stack operations for a given recursive call.



Analysis

- If I give input as 10, the recursive method will be called 12 times,
- If I give input as 14, the recursive method will be called 16 times,
- If I generalise the number time of call then it will be – n + 2 times.

In recursion, a function **α** either calls itself directly or calls a function **β** that in turn calls the original function **α**. The function **α** is called recursive function.



Operators Precedence & Associativity

The precedence of operators determines which operator is executed first if there is more than one operator in an expression.

Operators Precedence & Associativity table

	Operator	Description	Associativity
	( ) [ ] . -> ++ --	Parentheses: grouping or function call Brackets (array subscript) Dot operator (Member selection via object name) Arrow operator(Member selection via pointer) Postfix increment/decrement	Left to Right
U	+ - ++ -- ! ~ * & (datatype) sizeof	Unary plus Unary minus Prefix increment/decrement Logical NOT One's complement Indirection Address (of operand) Type cast Determine size in bytes on this implementation	Right to Left
A	* / %	Multiplication Division Modulus	Left to Right
	+ -	Addition Subtraction	Left to Right
S	<< >>	Left shift Right shift	Left to Right
C	< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left to Right
	== !=	Equal to Not equal to	Left to Right
B	&	Bitwise AND	Left to Right
	^	Bitwise XOR	Left to Right
	|	Bitwise OR	Left to Right
L	&&	Logical AND	Left to Right
	||	Logical OR	Left to Right
	? :	Conditional operator	Right to Left
A	= *= /= %= += -= &= ^= |= <<= >>=	Assignment operators	Right to Left
C	,	Comma operator	Left to Right

Precedence level decreasing

Example1

- int x = 5 - 17* 6;

The expression above is equivalent to:

- int x = 5 – (17* 6);

Example2

- 1 == 2 != 3
- Here, operators == and != have the same precedence. And, their associativity is from left to right. Hence, 1 == 2 is executed first.
- The expression above is equivalent to:

  (1 == 2) != 3

Example3

How to evaluate ++*p and *++p?

Apply right to left associativity

So it is (++(*p)) increment the value pointed by p.

*++p taken as *(++p) means increment the value of p and take the value pointed by p.

Note: here * and ++ are of same priority and having right to left associativity.

How to **convert infix to postfix using stack in C language program**?

- Infix to Postfix conversion is one of the most important applications of stack.
- One of the applications of Stack is in the conversion of arithmetic expressions in high-level programming languages into machine readable form.
- As our computer system can only understand and work on a binary language, it assumes that an arithmetic operation can take place in two operands only e.g., **A+B, C*D,D/A** etc. But in our usual form an arithmetic expression may consist of more than one operator and two operands e.g. **(A+B)*C(D/(J+D))**.

Infix and Postfix expressions

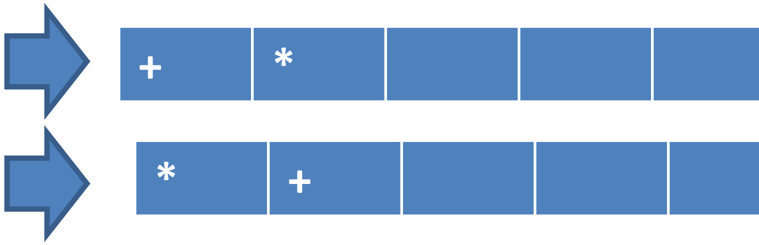
- These complex arithmetic operations can be converted into polish notation using stacks which then can be executed in two operands and an operator form.
- Infix Expression
- It follows the scheme of **<operand><operator><operand>** i.e. an <operator> is preceded and succeeded by an <operand>. Such an expression is termed infix expression. E.g., **A+B**
- Postfix Expression
- It follows the scheme of **<operand><operand><operator>** i.e. an <operator> is succeeded by both the <operand>. E.g., **AB+**

Infix and postfix examples

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

Motivation

- 1. a+b*c->a+(bc*) -> abc*+
- 2. a*b+c ->(ab*)+c-> ab*c+
- 3. a*b+(c-d)-> (ab*)+ (cd-)-> ab*cd-+



(This cannot ha

Print a
Push *
Print b
Push +, not possible
Pop*
Print*
Push+
Printc
Pop+

Algorithm to convert Infix To Postfix

Let, X is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Y.

1. Push “(“onto Stack, and add “)” to the end of X.
2. Scan X from left to right and repeat Step 3 to 6 for each element of X until the Stack is empty.
3. If an operand is encountered, add it to Y.
4. If a left parenthesis is encountered, push it onto Stack.
5. If an operator is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) which has the same precedence as or higher precedence than operator.
    2. Add operator to Stack.
       [End of If]
6. If a right parenthesis is encountered ,then:
    1. Repeatedly pop from Stack and add to Y each operator (on the top of Stack) until a left parenthesis is encountered.
    2. Remove the left Parenthesis.
       [End of If]
       [End of If]
7. END.

Simple way

- We use a stack
- When an operand is read, output it
- When an operator is read
  - Pop until the top of the stack has an element of lower precedence
  - Then push it
- When) is found, pop until we find the matching (
- (has the lowest precedence when in the stack
- but has the highest precedence when in the input
- When we reach the end of input, pop until the stack is empty

Let’s take an example to better understand the algorithm

Infix Expression: **A+ (B*C-(D/E^F)*G)*H**, where **^** is an exponential operator.

Symbol	Scanned	STACK	Postfix Expression	Description
1.		(		Start
2.	A	(	A	
3.	+	(+	A	
4.	(	(+(	A	
5.	B	(+(	AB	
6.	*	(+(*	AB	
7.	C	(+(*	ABC	
8.	-	(+(-	ABC*	‘*’ is at higher precedence than ‘-‘
9.	(	(+(-(	ABC*	
10.	D	(+(-(	ABC*D	
11.	/	(+(-(/	ABC*D	
12.	E	(+(-(/	ABC*DE	
13.	^	(+(-(/^	ABC*DE	
14.	F	(+(-(/^	ABC*DEF	
15.	)	(+(-	ABC*DEF^/	Pop from top on Stack, that’s why ‘^’ Come first
16.	*	(+(-*	ABC*DEF^/	
17.	G	(+(-*	ABC*DEF^/G	
18.	)	(+	ABC*DEF^/G*-	Pop from top on Stack, that’s why ‘^’ Come first
19.	*	(+*	ABC*DEF^/G*-	
20.	H	(+*	ABC*DEF^/G*-H	
21.	)	Empty	ABC*DEF^/G*-H*+	END

Here the stack is called operator stack

Résultant Postfix Expression: ABC*DEF^/G*-H*+

Advantage of Postfix Expression over Infix Expression

- An infix expression is difficult for the machine to know and keep track of precedence of operators. On the other hand, a postfix expression itself determines the precedence of operators (as the placement of operators in a postfix expression depends upon its precedence).Therefore, for the machine it is easier to carry out a postfix expression than an infix expression.

Evaluation of Postfix Expressions Using Stack

- As discussed in Infix To Postfix Conversion Using Stack, the compiler finds it convenient to evaluate an expression in its postfix form.
- The virtues of postfix form include elimination of parentheses which signify priority of evaluation and the elimination of the need to observe rules of hierarchy, precedence and associativity during evaluation of the expression.

- As **Postfix expression** is without parenthesis and can be evaluated as two operands and an operator at a time, this becomes easier for the compiler and the computer to handle.

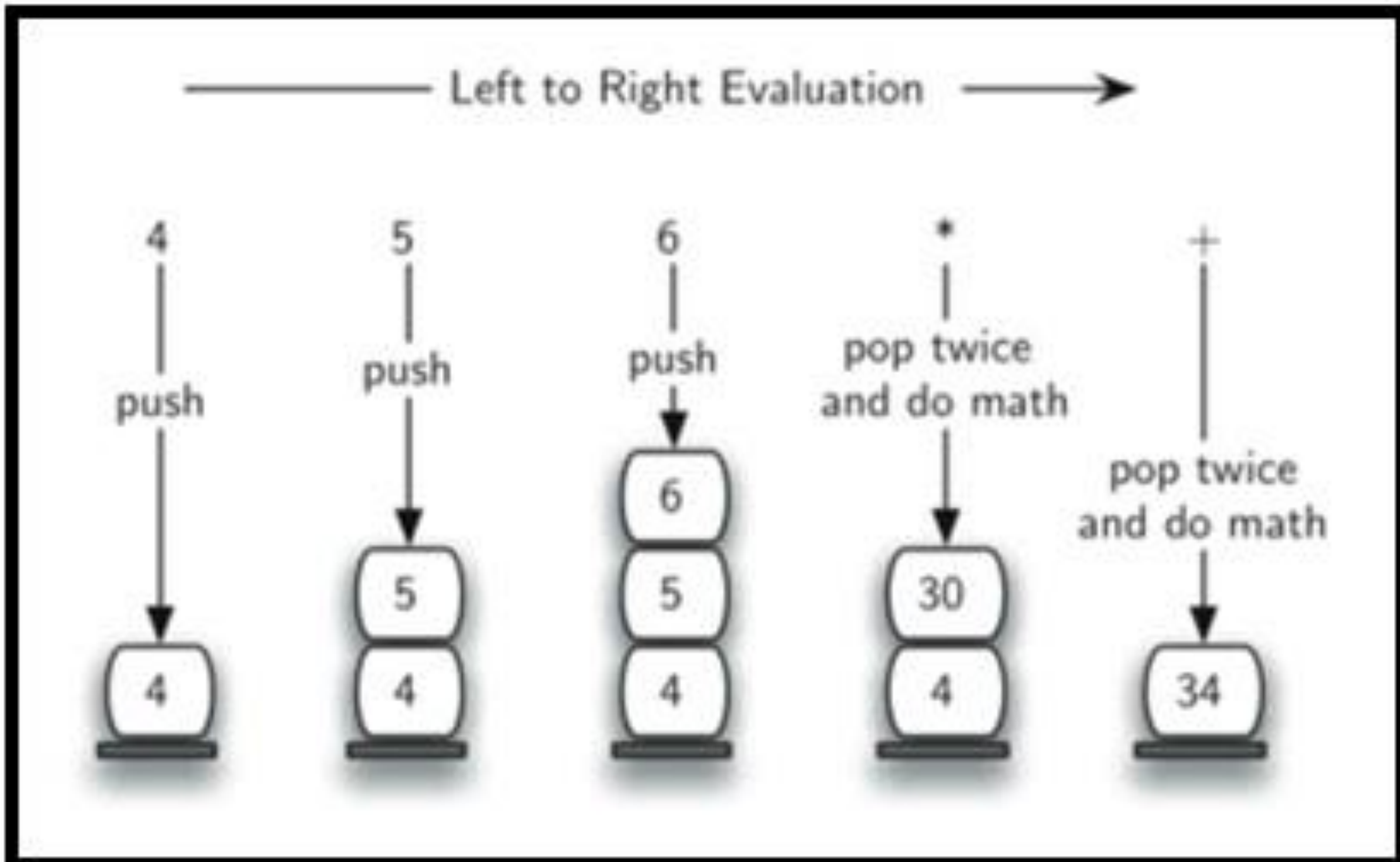
Contd…

- **Evaluation rule of a Postfix Expression states:**
- While reading the expression from left to right, push the element in the stack if it is an operand.
- Pop the two operands from the stack, if the element is an operator and then evaluate it.
- Push back the result of the evaluation. Repeat it till the end of the expression.

Postfix evaluation

- Algorithm
- **1)** Add) to postfix expression.
  2) Read postfix expression Left to Right
  until) encountered
  3) If operand is encountered, push it onto Stack
  [End If]
  4) If operator is encountered, Pop two elements
  i) A -> Top element
  ii) B-> Next to Top element
  iii) Evaluate B operator A
  push B operator A onto Stack
  5) Set result = pop
  6) END

Expression: 456*+



Expression: 456*+

Here the stack is called operand stack

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) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)

Examples

- 53+62/*35*+ Output: The result is: 39
- 10 2 8 * + 3 - Output: 23
- 2 3 4 + * 6 – Output: 8

Postfix Expression	Infix Equivalent	Result
4 5 7 2 + − ×	4 × (5 − (7 + 2))	−16
3 4 + 2 × 7 /	((3 + 4) × 2)/7	2
5 7 + 6 2 − ×	(5 + 7) × (6 − 2)	48
4 2 3 5 1 − + × + ×	? × (4 + (2 × (3 + (5 − 1))))	not enough operands
4 2 + 3 5 1 − × +	(4 + 2) + (3 × (5 − 1))	18
5 3 7 9 + +	(3 + (7 + 9)) ... 5???	too many operands

Check for balanced parentheses in an expression

Given an expression string exp, write a program to examine whether the pairs and the orders of “{“, “}”, “(“, “)”, “[“, “]” are correct in exp.

Example:

Input: *exp = “[()]{}{[()()]()}”*

Output: *Balanced*

Input: *exp = “[(])”*

Output: *Not Balanced*

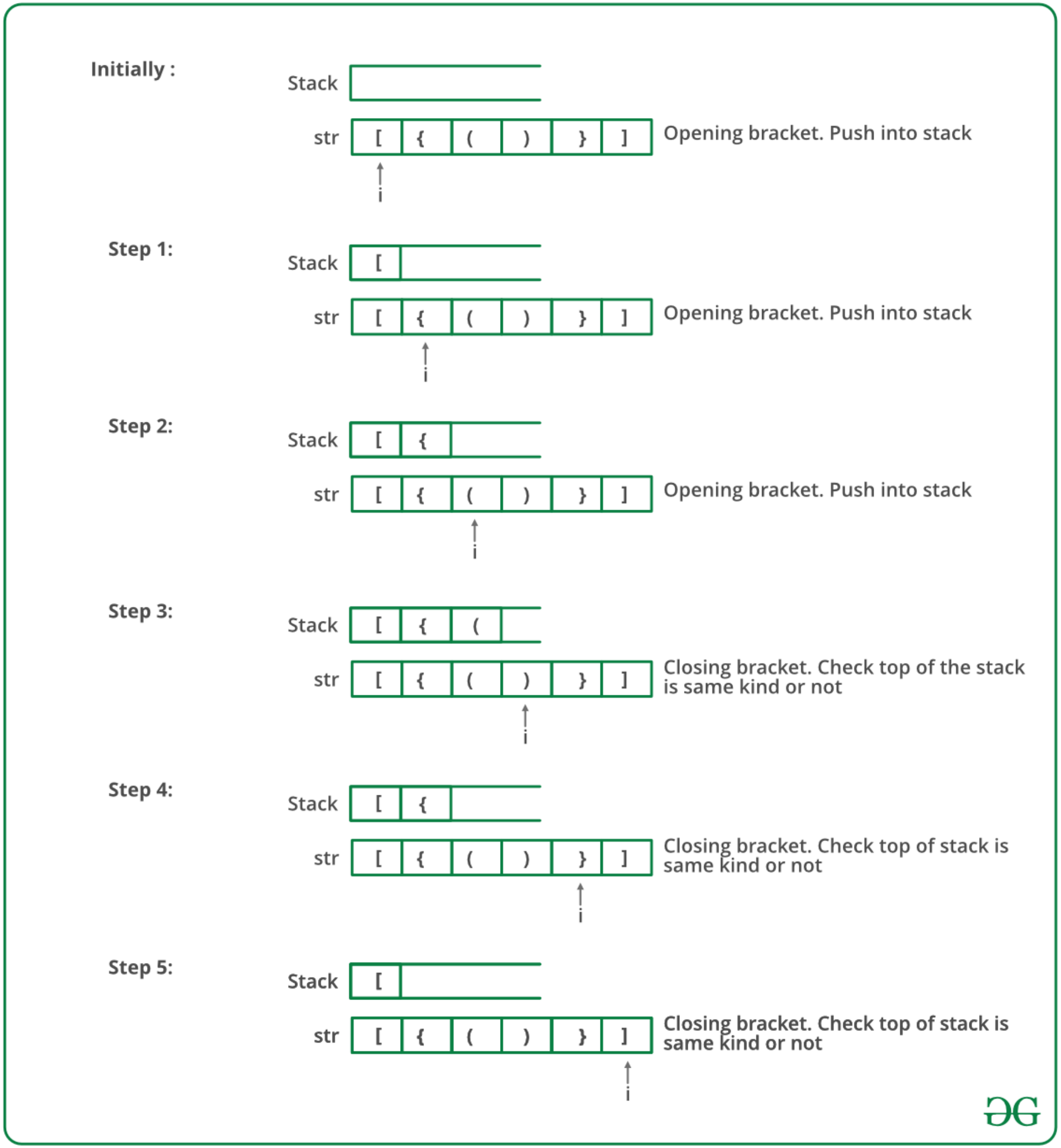
Checking for balanced parentheses is one of the most important task of a compiler.

```
int main( ){

    for ( int i=0; i < 10; i++)
    {
      //some code
    }
}
}   Compiler generates error
```

Algorithm:

- Declare a character stack S.
- Now traverse the expression string exp.
    1. If the current character is a starting bracket (**'(' or '{' or '['**) then push it to stack.
    2. 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 else parenthesis are not balanced.
- After complete traversal, if there is some starting bracket left in stack then "not balanced"



Initially :
Stack
str
[{ ( ) }]
Opening bracket. Push into stack
i
Step 1:
Stack
[
str
[{ ( ) }]
Opening bracket. Push into stack
i
Step 2:
Stack
[{
str
[{ ( ) }]
Opening bracket. Push into stack
i
Step 3:
Stack
[{ (
str
[{ ( ) }]
Closing bracket. Check top of the stack is same kind or not
i
Step 4:
Stack
[{
str
[{ ( ) }]
Closing bracket. Check top of stack is same kind or not
i
Step 5:
Stack
[
str
[{ ( ) }]
Closing bracket. Check top of stack is same kind or not
i

References

- http://knowledge-cess.com/recursion-vs-iteration-an-analysis-fibonacci-and-factorial/
- Tutorials point
- https://www.journaldev.com/36220/queue-in-c
- https://www.studytonight.com/
- https://www.geeksforgeeks.org/