Data Structure & Algorithms

Problem Solving with Stack

Infix, Prefix and Postfix Notations

Infix, Postfix and Prefix Notations

Postponement: Evaluating arithmetic expressions.

Prefix: + a b

Infix: a + b (what we use in grammar school)

Postfix: a b +

In high level languages, infix notation cannot be used to evaluate expressions. We must analyze the expression to determine the order in which we evaluate it. A common technique is to convert a infix notation into postfix notation, then evaluating it.

Infix, Postfix and Prefix Notations

- The usual way of expressing the sum of two numbers A and B is :

  A+B

- The operator '+' is placed between the two operands A and B
- This is called the "***Infix Notation***"
- Consider a bit more complex example:

  (13 – 5) / (3 + 1)

- When the parentheses are removed the situation becomes ambiguous

  13 – 5 / 3 + 1

  is it (13 – 5) / (3 + 1)

  or 13 – (5 / 3) + 1

- To cater for such ambiguity, you must have operator precedence rules to follow (as in C)

Infix, Postfix and Prefix Notations

- In the absence of parentheses

  13 – 5 / 3 + 1

- Will be evaluated as 13 – (5 / 3) + 1
- Operator precedence is **by-passed** with the help of parentheses as in (13 – 5) / (3 + 1)
- The infix notation is therefore cumbersome due to
  - Operator Precedence rules and
  - Evaluation of Parentheses

Postfix Notation

- It is a notation for writing arithmetic expressions in which operands appear before the operator
- E.g. A + B is written as A B + in postfix notation
- There are no precedence rules to be learnt in it.
- Parentheses are never needed
- Due to its simplicity, some calculators use postfix notation
- This is also called the "Reverse Polish Notation or RPN"

Postfix Notation – Some examples

Infix Expressions	Corresponding Postfix
5 + 3 + 4 + 1	5 3 + 4 + 1 +
(5 + 3) * 10	5 3 + 10 *
(5 + 3) * (10 – 4)	5 3 + 10 4 - *
5 * 3 / (7 – 8)	5 3 * 7 8 - /
(b * b – 4 * a * c) / (2 * a)	b b * 4 a * c * - 2 a * /

Conversion from Infix to Postfix Notation

- We have to accommodate the presence of operator precedence rules and Parentheses while converting from infix to postfix
- What are possible items in an input Infix expression?
- **Data objects** required for the conversion?
  - An infix expression string read one item at a time
  - An operator / parentheses stack
  - A Postfix expression string to store the resultant

Conversion from Infix to Postfix

- The Algorithm
  - Read an item from input infix expression
  - If item is an operand append it to postfix string
  - If item is "(" push it on the stack
  - If the item is an operator
    - If the operator has higher precedence than the one already on top of the stack then push it onto the operator stack
    - If the operator has lower or equal precedence than the one already on top of the stack then
      - pop the operator on top of the operator stack and append it to postfix string, and
      - push lower precedence operator onto the stack
  - If item is ")" pop all operators from top of the stack one-by-one, until a "(" is encountered on stack and removed
  - If end of infix string pop the stack one-by-one and append to postfix string

Infix to Postfix Example#1

Infix to Postfix Example#1

- Dry Run of Infix to Postfix Conversion using Stack in C

Infix expression => a * (b + c + d)

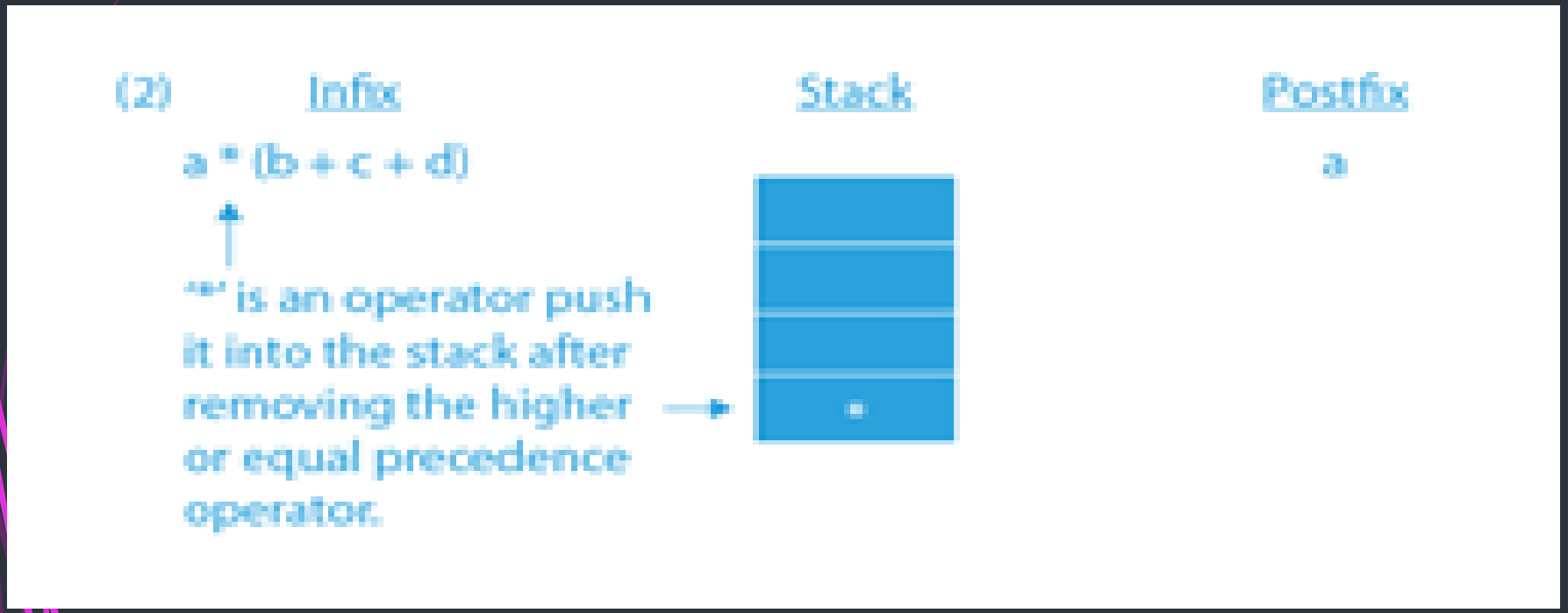
(1)

Infix	Stack	Postfix
a * (b + c + d)		a

'a' is an operand append it to the post fix

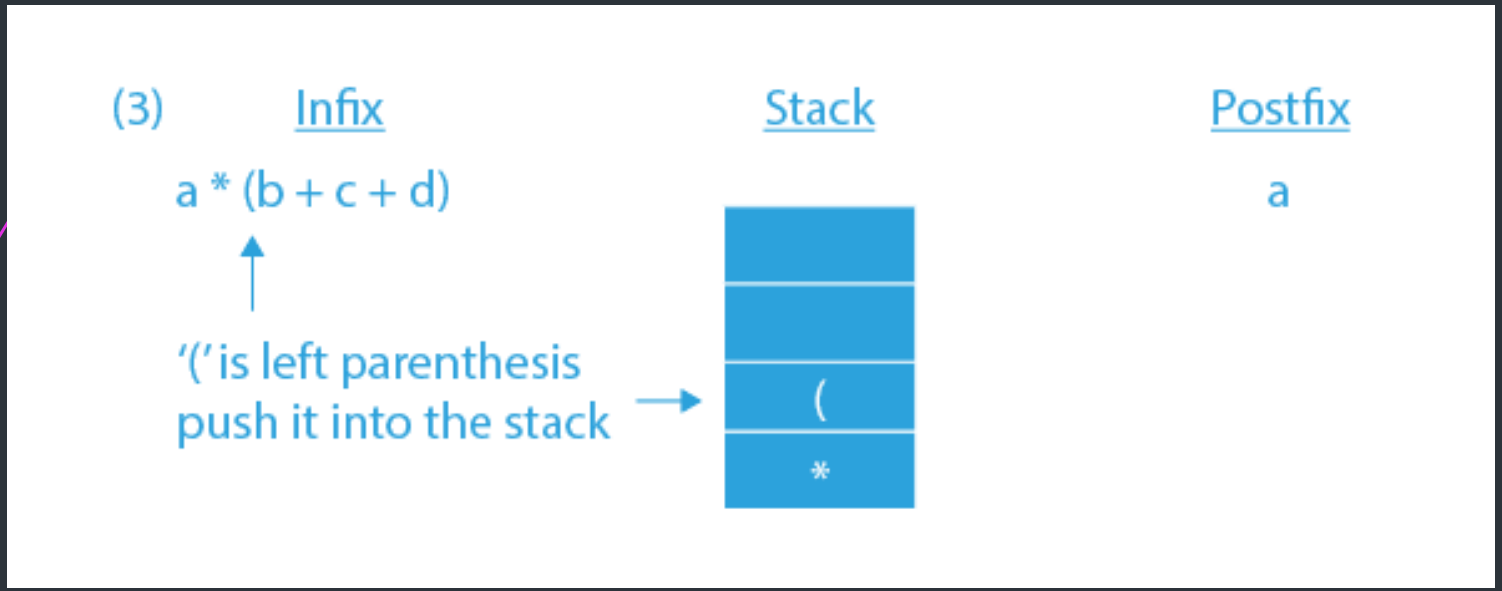
Infix to Postfix Example#1

- Dry Run of Infix to Postfix Conversion using Stack in C



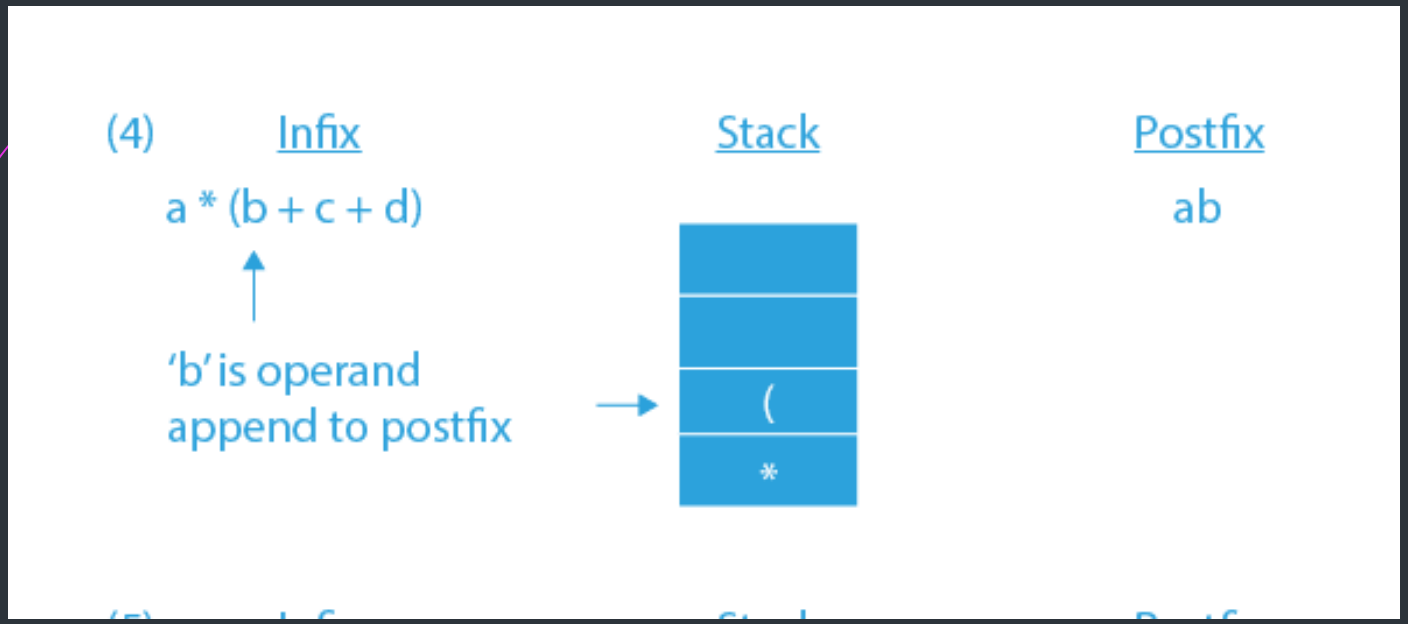
Infix to Postfix Example#1

- Dry Run of Infix to Postfix Conversion using Stack in C



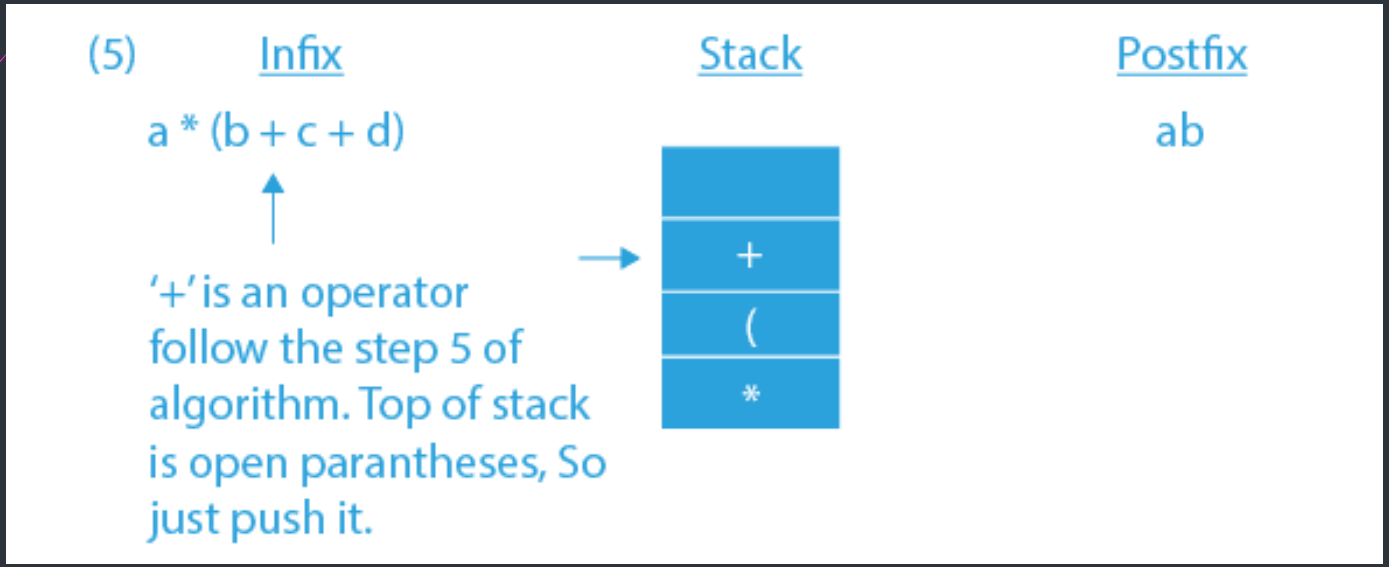
Infix to Postfix Example#1

- Dry Run of Infix to Postfix Conversion using Stack in C



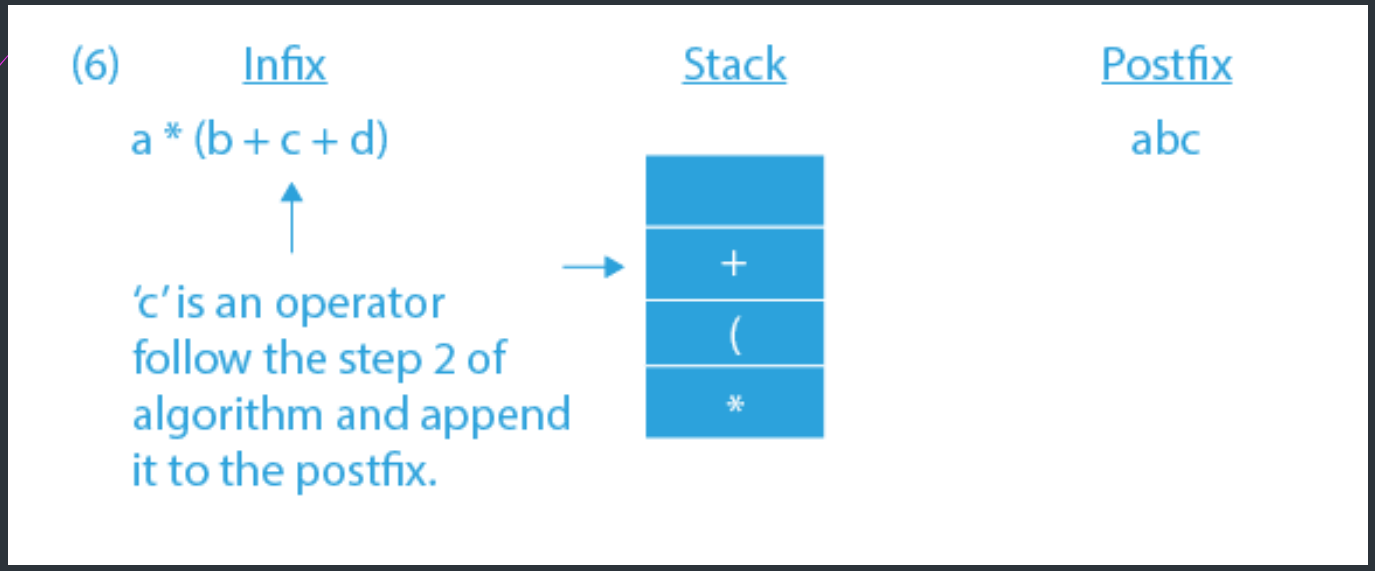
Infix to Postfix Example#1

- Dry Run of Infix to Postfix Conversion using Stack in C



Infix to Postfix Example#1

- Dry Run of Infix to Postfix Conversion using Stack in C



Infix to Postfix Example#1

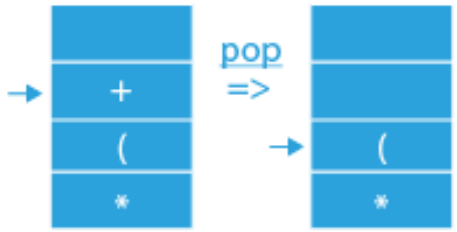
- Dry Run of Infix to Postfix Conversion using Stack in C

(7) Infix

a * (b + c + d)

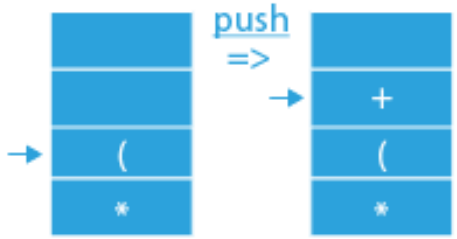
=> '+' is an operator, so push it into stack after removing higher or equal priority operators from top of stack.

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



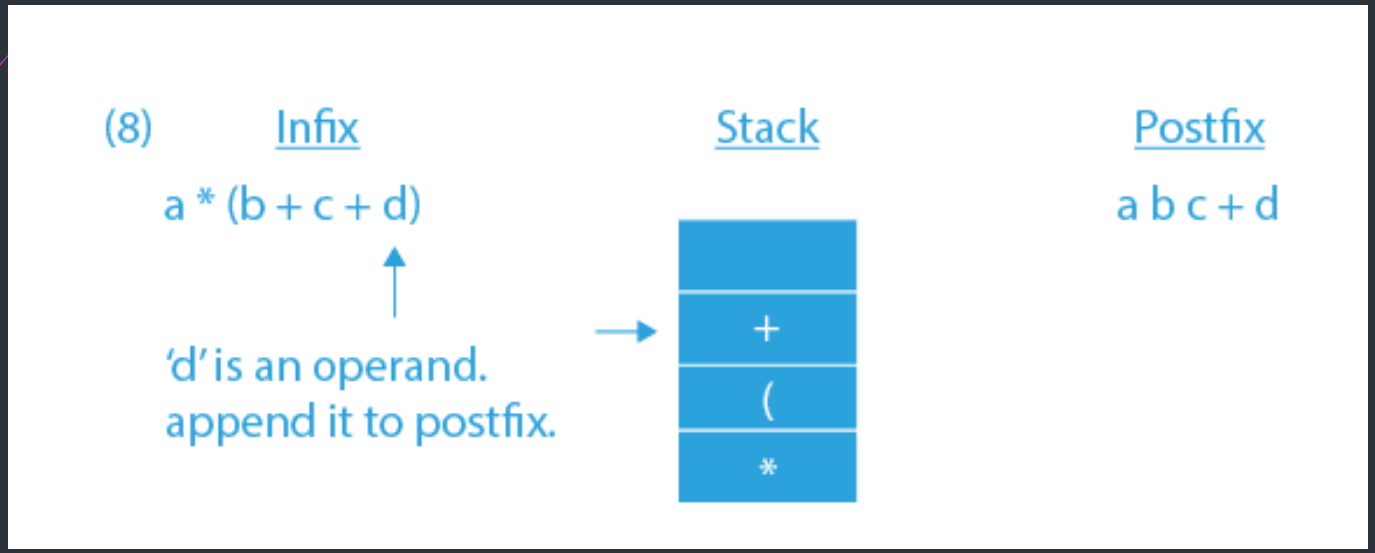
postfix => a b c +

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



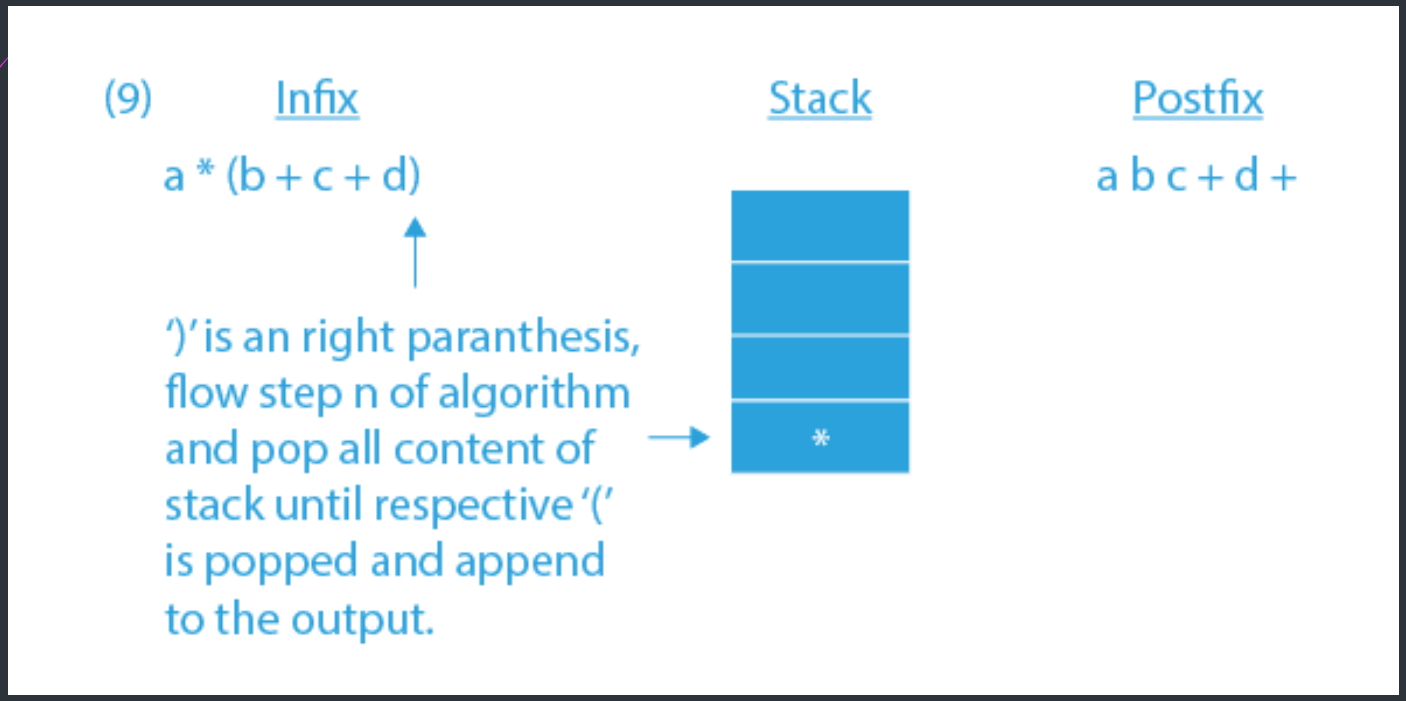
Infix to Postfix Example#1

- Dry Run of Infix to Postfix Conversion using Stack in C



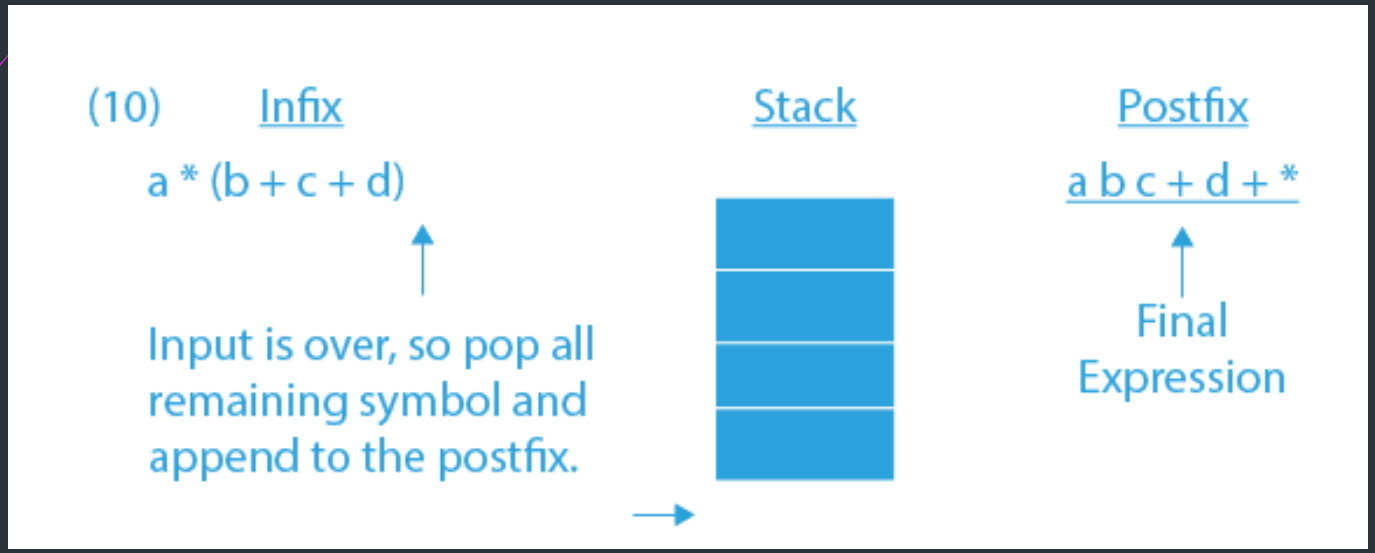
Infix to Postfix Example#1

- Dry Run of Infix to Postfix Conversion using Stack in C



Infix to Postfix Example#1

- Dry Run of Infix to Postfix Conversion using Stack in C



Infix to Postfix Example #2

A * B – (C + D) + E

	Infix	Stack(bot->top)	Postfix
a)	A * B – (C – D) + E	empty	empty
b)	* B – (C + D) + E	empty	A
c)	B – (C + D) + E	*	A
d)	– (C + D) + E	*	A B
e)	– (C + D) + E	empty	A B *
f)	(C + D) + E	–	A B *
g)	C + D) + E	– (	A B *
h)	+ D) + E	– (	A B * C
i)	D) + E	– (+	A B * C
j)	) + E	– (+	A B * C D
k)	+ E	–	A B * C D +
l)	+ E	empty	A B * C D + –
m)	E	+	A B * C D + –
n)		+	A B * C D + – E
o)		empty	A B * C D + – E +

Postfix Notation – Some examples

Infix Expressions	Corresponding Postfix
5 + 3 + 4 + 1	5 3 + 4 + 1 +
(5 + 3) * 10	5 3 + 10 *
(5 + 3) * (10 – 4)	5 3 + 10 4 - *
5 * 3 / (7 – 8)	5 3 * 7 8 - /
(b * b – 4 * a * c) / (2 * a)	b b * 4 a * c * - 2 a * /

Try it yourself

- Show a trace of algorithm that converts the infix expression
  - (X + Y) * (P – Q / L)
  - L – M / (N * O ^ P)

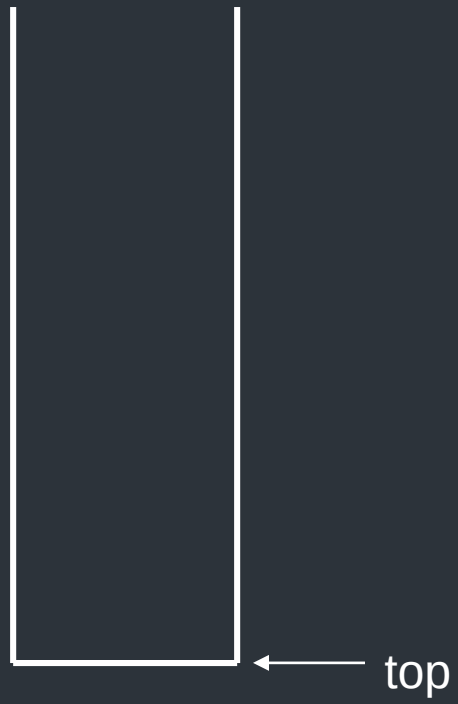
Evaluation of Postfix Expression

- After an infix expression is converted to postfix, its evaluation is a simple affair
- Stack comes in handy, AGAIN
- The Algorithm
  - Read the postfix expression one item at-a-time
  - If item is an operand push it on to the stack
  - If item is an operator:
    - pop the top two operands from stack such that **first popped operand is B** and **second popped operand is A**
    - and apply the operator as **A (Operator) B**
  - Push the result back on top of the stack, which will become an operand for next operation
  - Final result will be the only item left on top of the stack

Stack in Action ….

Postfix Expression

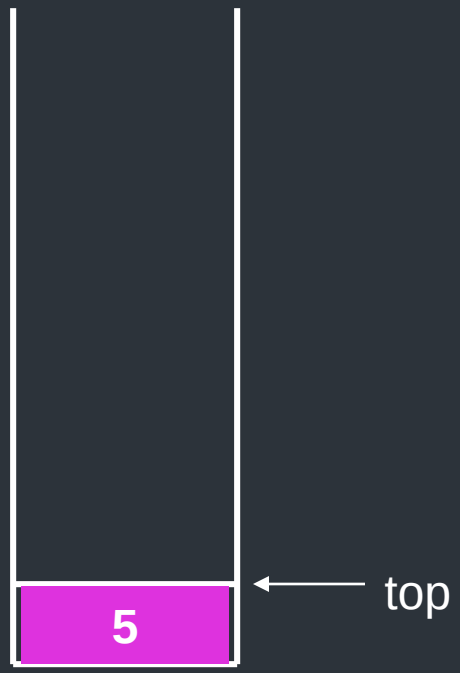
5 7 + 6 2 - *



Stack in Action ….

Postfix Expression

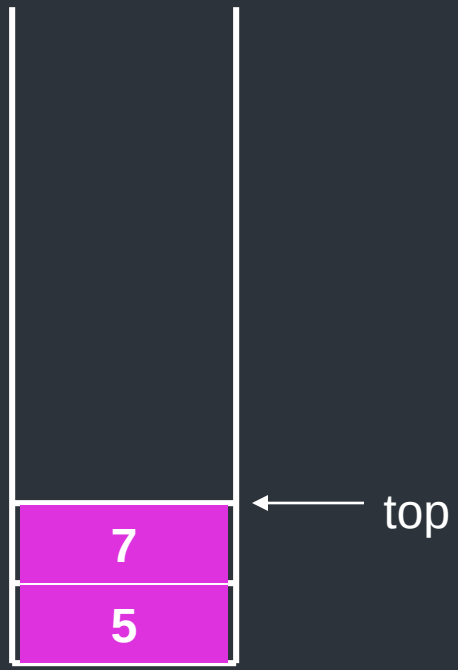
5 7 + 6 2 - *



Stack in Action ….

Postfix Expression

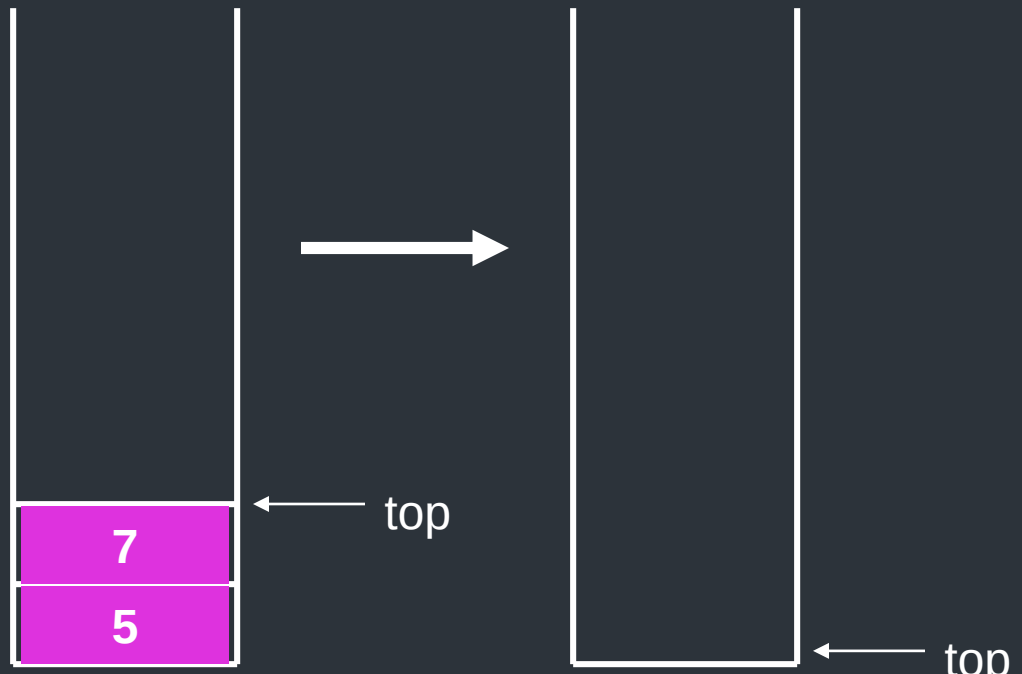
5 7 + 6 2 - *



Stack in Action ….

Postfix Expression

5 7 + 6 2 - *

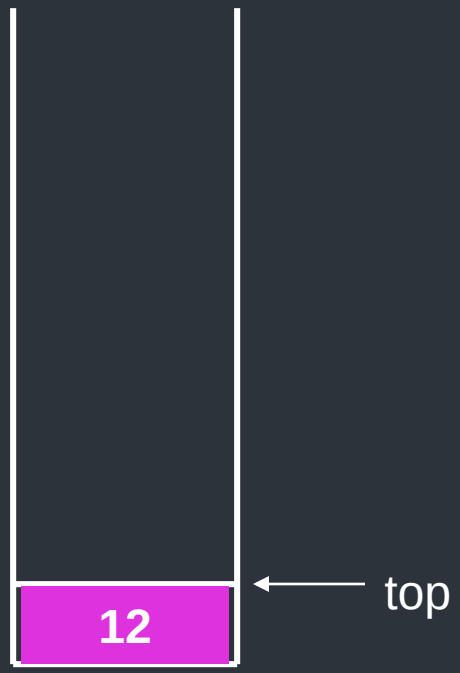


Result = Pop(5) “+” Pop(7) Push (Result)

Stack in Action ….

Postfix Expression

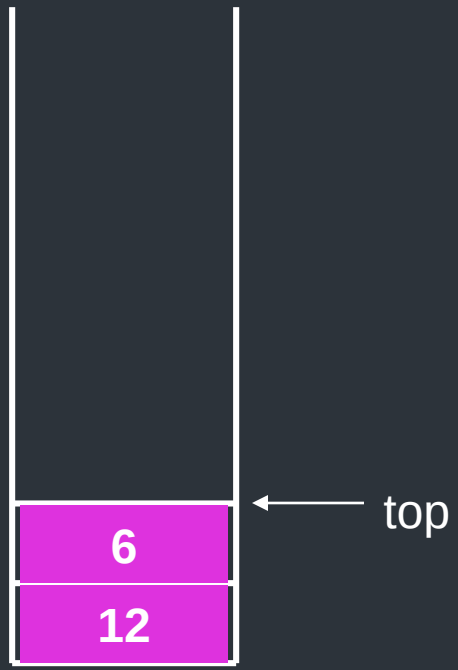
5 7 + 6 2 - *



Stack in Action ….

Postfix Expression

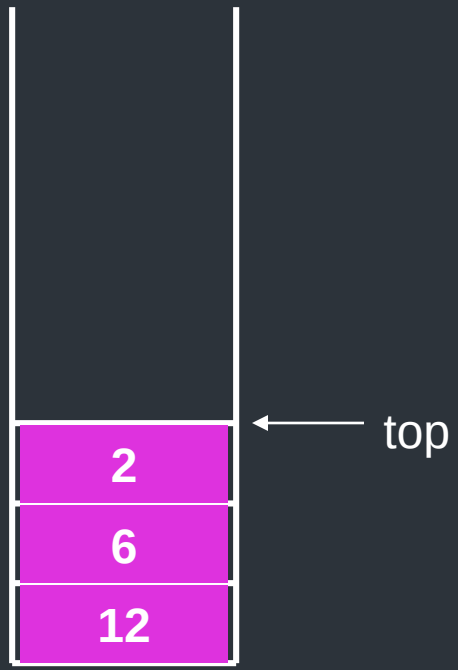
5 7 + 6 2 - *



Stack in Action ….

Postfix Expression

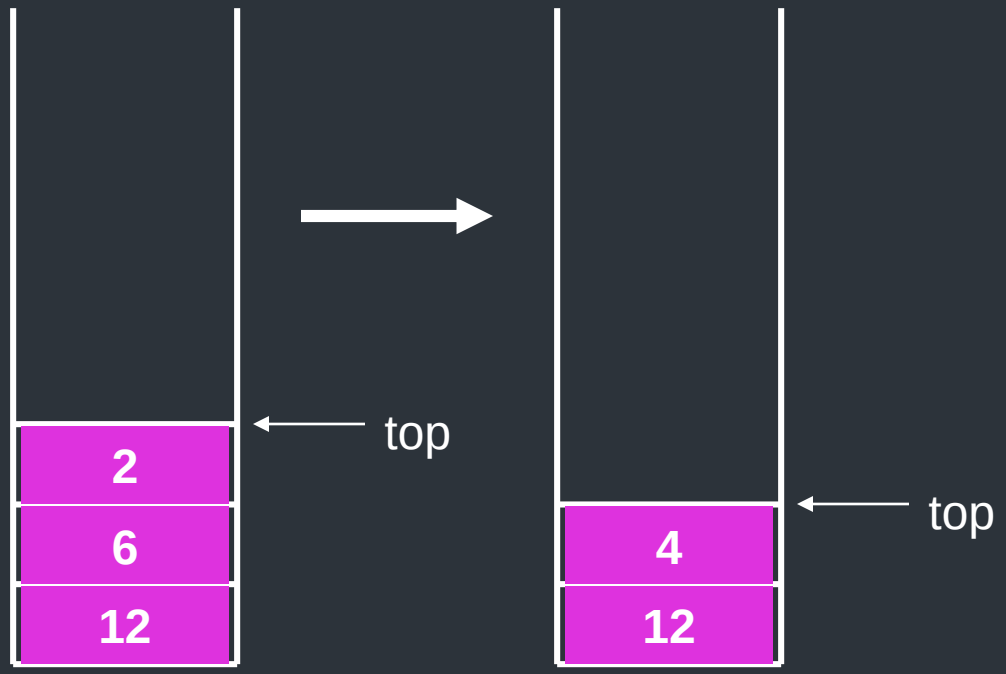
5 7 + 6 2 - *



Stack in Action ….

Postfix Expression

5 7 + 6 2 - *

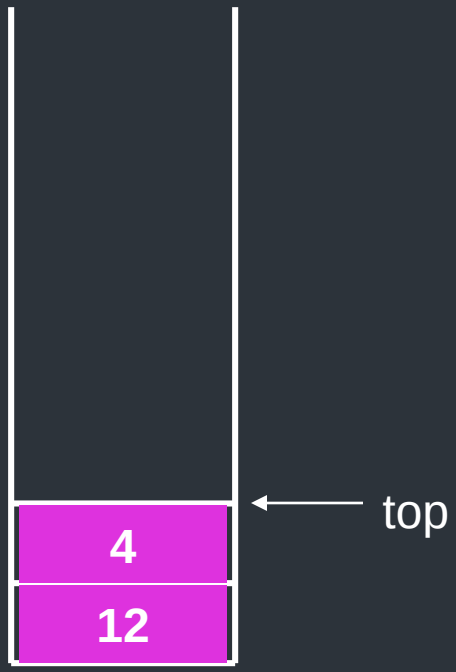


Result = Pop(6) “-” Pop(2) Push (Result)

Stack in Action ….

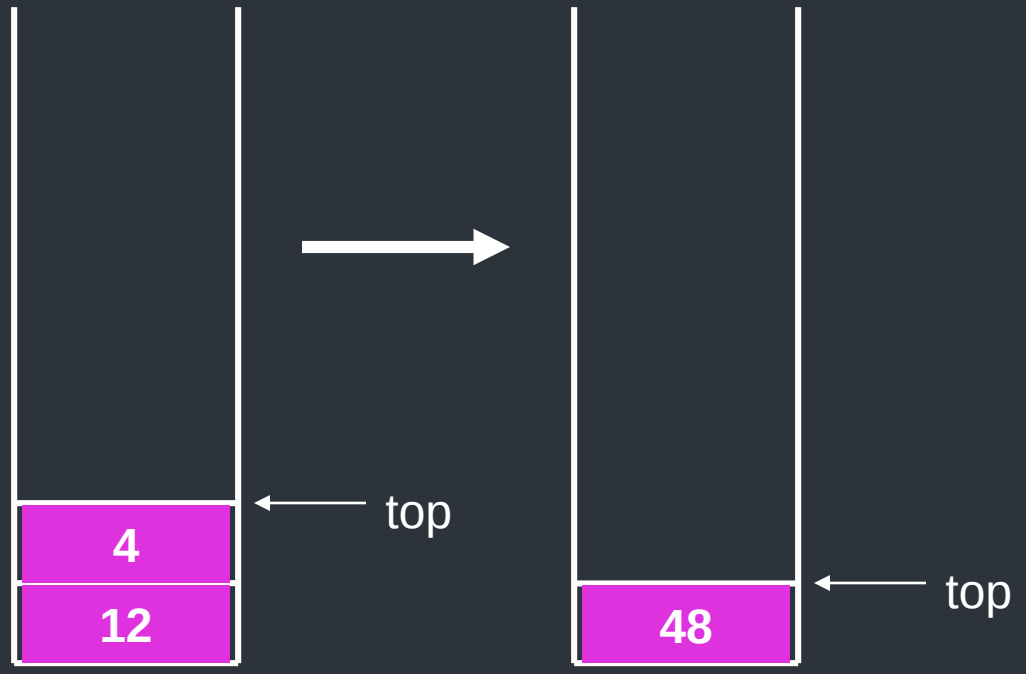
Postfix Expression

5 7 + 6 2 - *

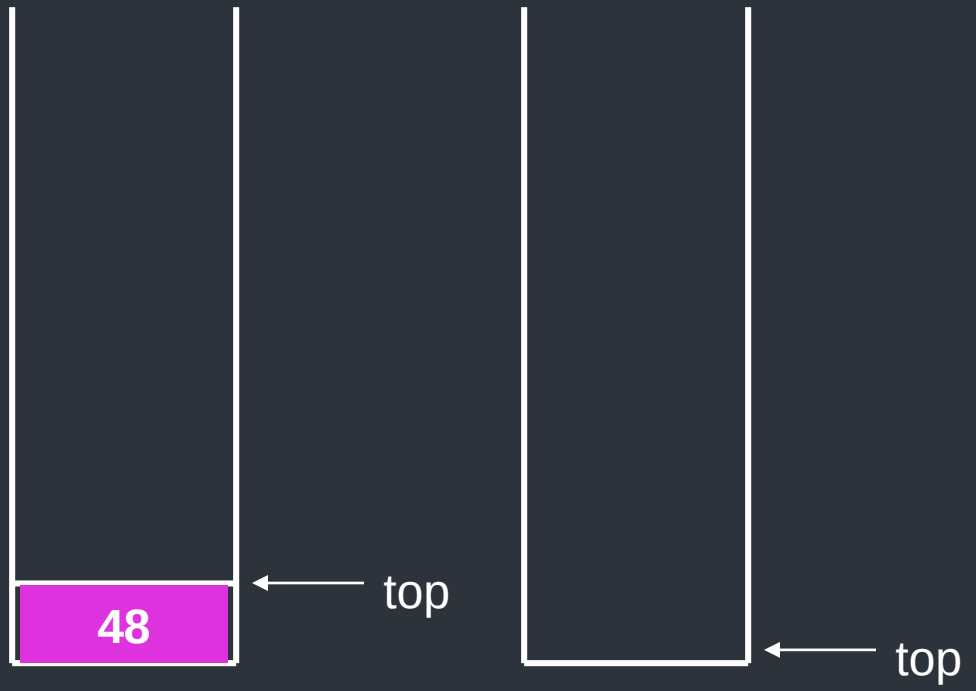


Postfix Expression

5 7 + 6 2 - *



Result = Pop(12) “ * ” Pop(4) Push (Result)



Postfix Expression

5 7 + 6 2 - *

Result = Pop(48)

Result = 48

Postfix Expression Evaluation

- Pseudocode:

```
for ( each character ch in the string )
    { if (ch is an operand )
          Push value that operand ch represents onto stack
    else                    // ch is an operator named op
    {                       // evaluate and push the result
        Pop the stack
                    operand2 = top of stack
         Pop the stack
                    operand1 = top of stack
        result = operand1 op operand2
        Push Result onto stack
    }  // end if
} // end for
```

Example

- Evaluate 2 3 4 5 + 2 * 1 + + 3 2 * + *

Evaluation of Postfix Expression

- Evaluation of infix Expression is difficult because :
  - Rules governing the precedence of operators are to be catered for
  - Many possibilities for incoming characters
  - To cater for parentheses
  - To cater for error conditions / checks
- Evaluation of postfix expression is very simple to implement because operators appear in precisely the order in which they are to be executed

Motivation for the conversion

- Motivation for this conversion is the need to have the operators in the precise order for execution
- While using paper and pencil to do the conversion we can "foresee" the expression string and the depth of all the scopes (if the expressions are not very long and complicated)
- When a program is required to evaluate an expression, it must be accurate
- At any time during scanning of an expression we cannot be sure that we have reached the inner most scope
- Encountering an operator or parentheses may require frequent "backtracking"
- Rather than backtracking, we use the stack to **"remember"** the operators encountered previously

Problem Solving with Stacks

- 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

Problem Solving (Cont....)

- 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

Problem Solving (Cont....)

- 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

Problem Solving (Cont….)

- 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

Problem Solving with Stack

- 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

Why the need for a Stack

- 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

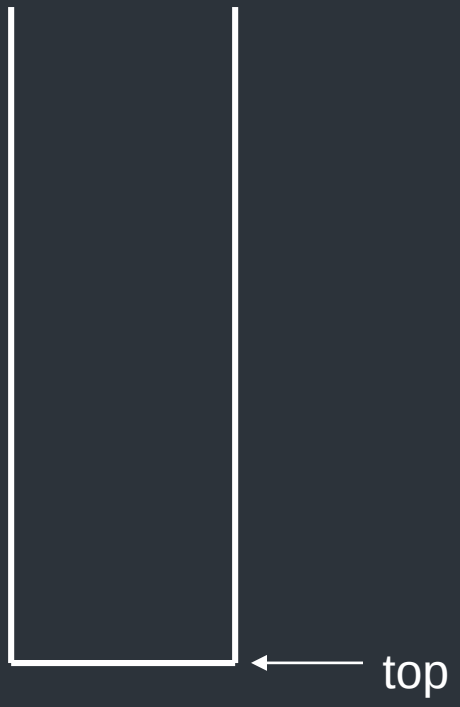
Delimiter Checking

- The common application of Stack is delimiter checking, i.e., parsing that involves analyzing a source program syntactically.
- It is also called parenthesis checking.
- When the compiler translates a source program written in some programming language such as C, C++ to a machine language, it parses the program into multiple individual parts such as variable names, keywords, etc.
- By scanning from left to right. The main problem encountered while translating is the unmatched delimiters.
- We make use of different types of delimiters include the parenthesis checking (,), curly braces {,} and square brackets [,], and common delimiters /* and */.
- Every opening delimiter must match a closing delimiter, i.e., every opening parenthesis should be followed by a matching closing parenthesis.
- Also, the delimiter can be nested. The opening delimiter that occurs later in the source program should be closed before those occurring earlier.

Delimiter Checking

Valid Delimiter	Invalid Delimiter
While (i > 0)	While (i >
/* Data Structure */	/* Data Structure
{ (a + b) - c }	{ (a + b) - c

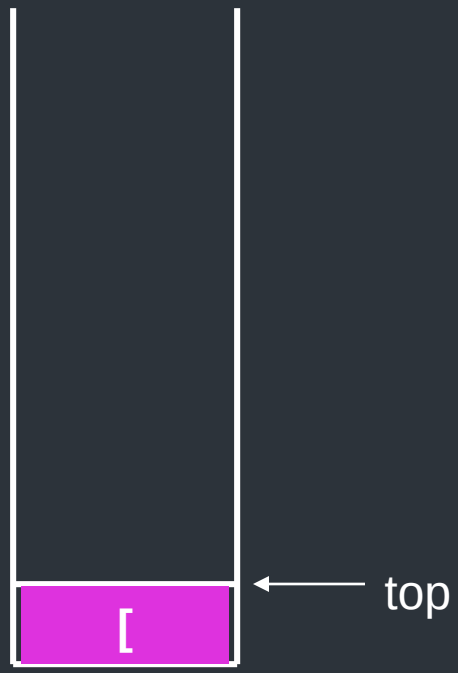
Stack in Action ….



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

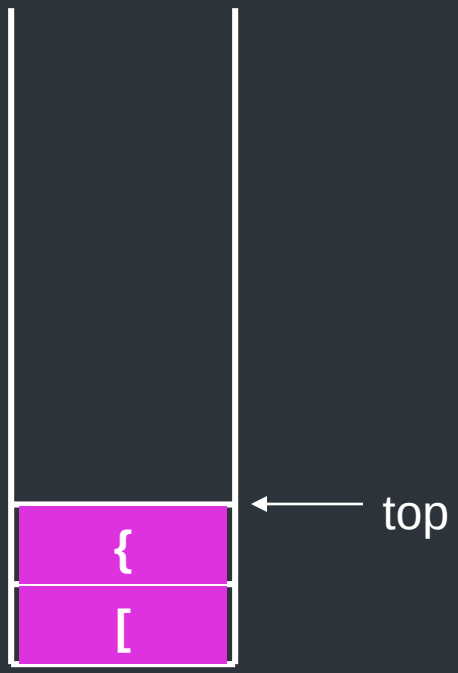
Stack in Action ….

Push('[');



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

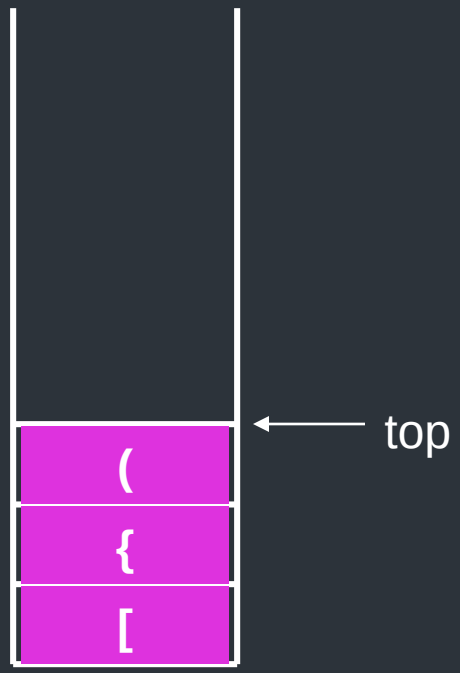
Stack in Action ….



Push(‘{‘);

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

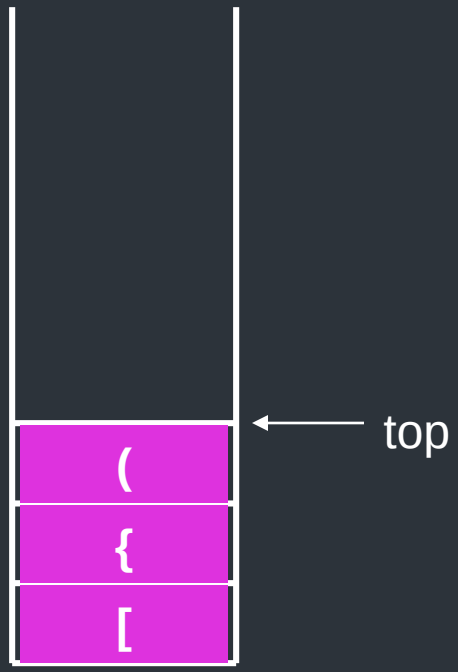
Stack in Action ….



Push(‘(‘);

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

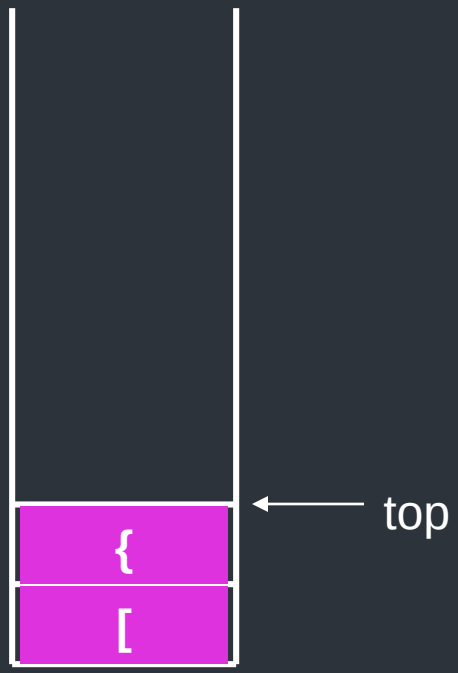
Stack in Action ….



Pop( );

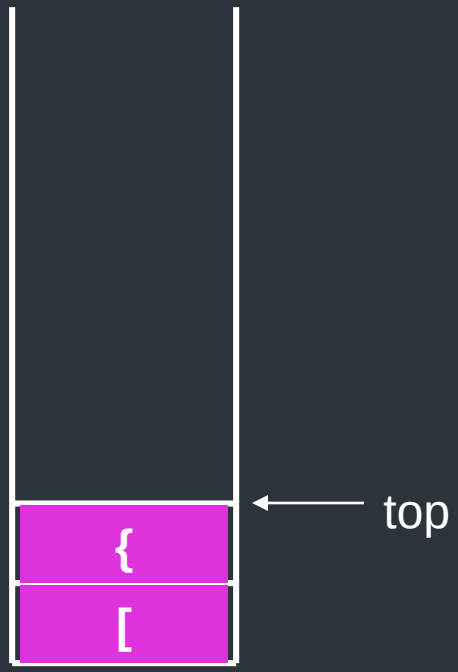
[A + { B - C + (D + E) }]

Stack in Action ….



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

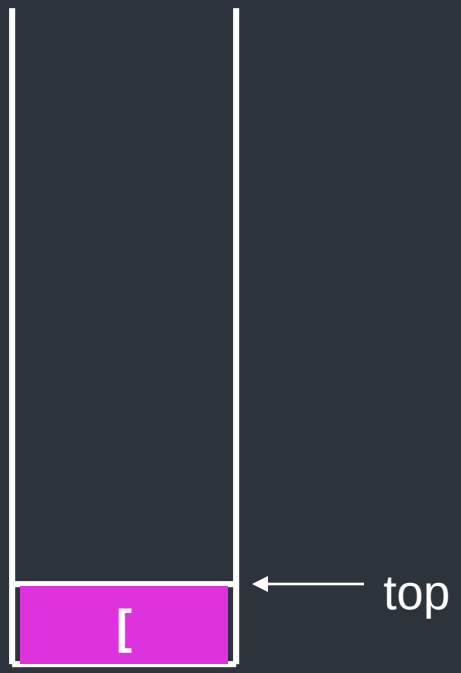
Stack in Action ….



Pop( );

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

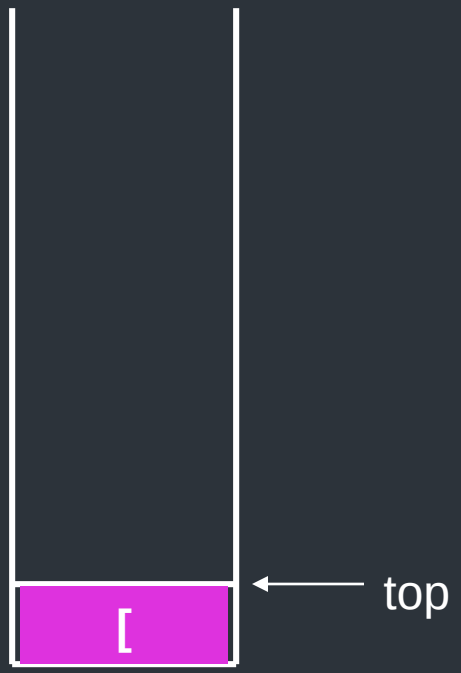
Stack in Action ….



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

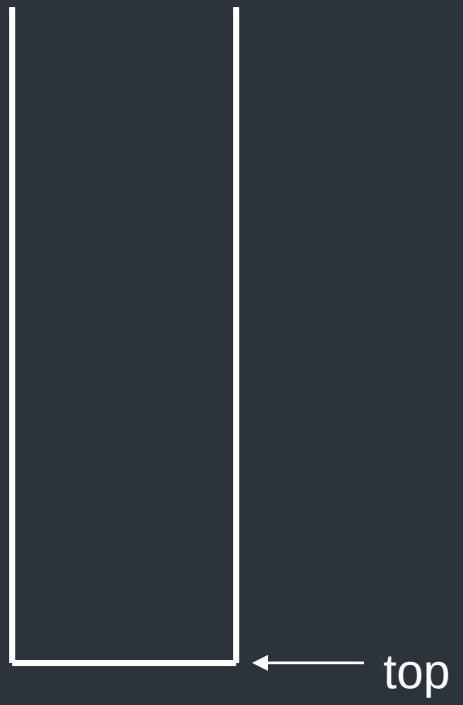
Stack in Action ….

Pop( );



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

Stack in Action ….



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

Result = A valid expression

Example: To explain this concept, let's consider the following expression.

[{a -b) * (c -d)}/f]

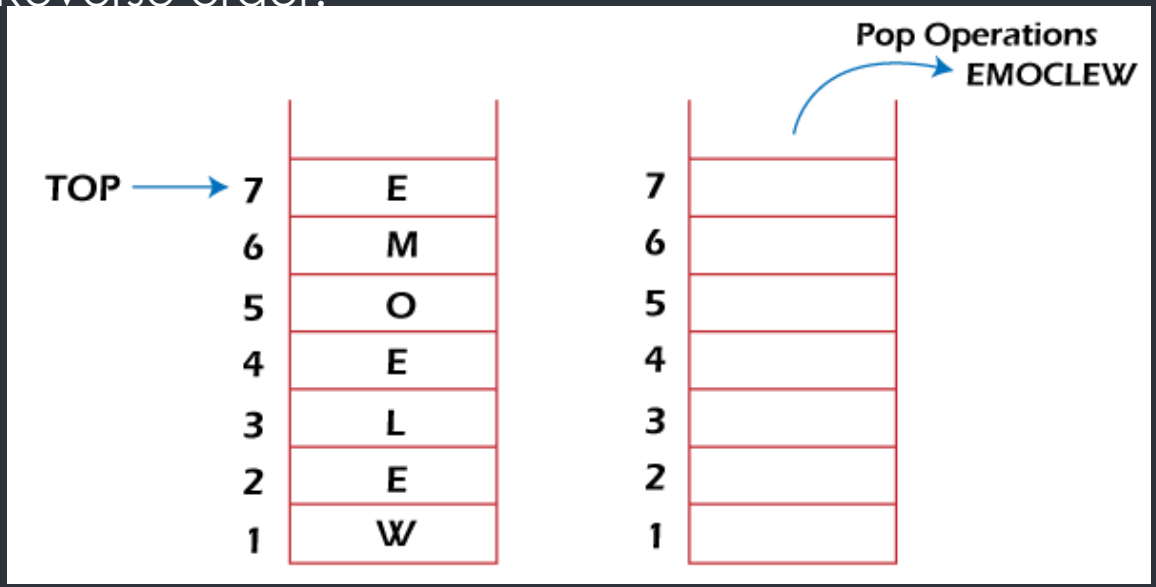
Input left	Characters Read	Stack Contents
{{(a-b) * (c-d)}/f]	[	[
{(a-b) * (c-d)}/f]	{	[{
(a-b) * (c-d)}/f]	(	[{ (
a-b) * (c-d)}/f]	a	[{ (
-b) * (c-d)}/f]	-	[{ (
b) * (c-d)}/f]	b	[{ (
) * (c-d)}/f]	)	[{
* (c-d)}/f]	*	[{
(c-d)}/f]	(	[{ (
c-d)}/f]	g	[{ (
-d)}/f]	-	[{ (
d)}/f]	d	[{ (
)}/f]	)	[{
}/f]	}	[
/f]	/	[
f]	f	[
]		

Reverse a Data:

- To reverse a given set of data, we need to reorder the data so that the first and last elements are exchanged, the second and second last element are exchanged, and so on for all other elements.
- **Example:** Suppose we have a string Welcome, then on reversing it would be Emoclew.
- **There are different reversing applications:**
  - Reversing a string
  - Converting Decimal to Binary

Reverse a String

- A Stack can be used to reverse the characters of a string.
- This can be achieved by simply pushing one by one each character onto the Stack, which later can be popped from the Stack one by one.
- Because of the **last in first out** property of the Stack, the first character of the Stack is on the bottom of the Stack and the last character of the String is on the Top of the Stack and after performing the pop operation in the Stack, the Stack returns the String in Reverse order.



Stack Applications

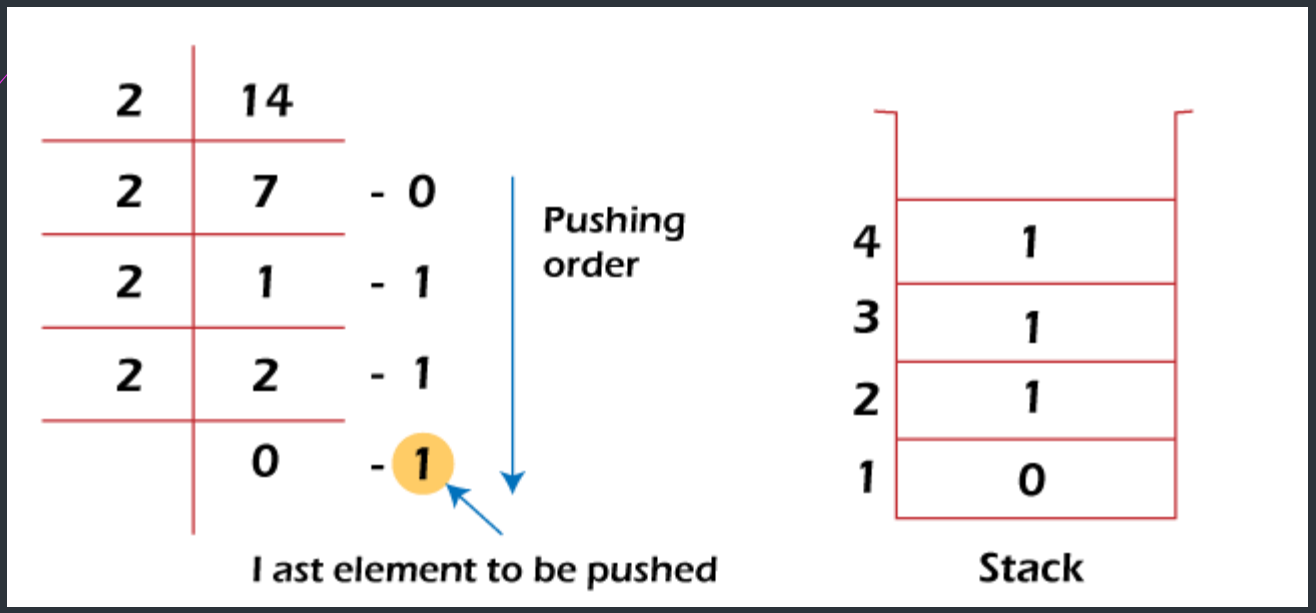
Converting Decimal to Binary: Consider the following pseudocode

```
Read (number)
Loop (number > 0)
        digit = number modulo 2
        print (digit)
        number = number / 2
// from Data Structures by Gilbert and Forouzan
```

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.

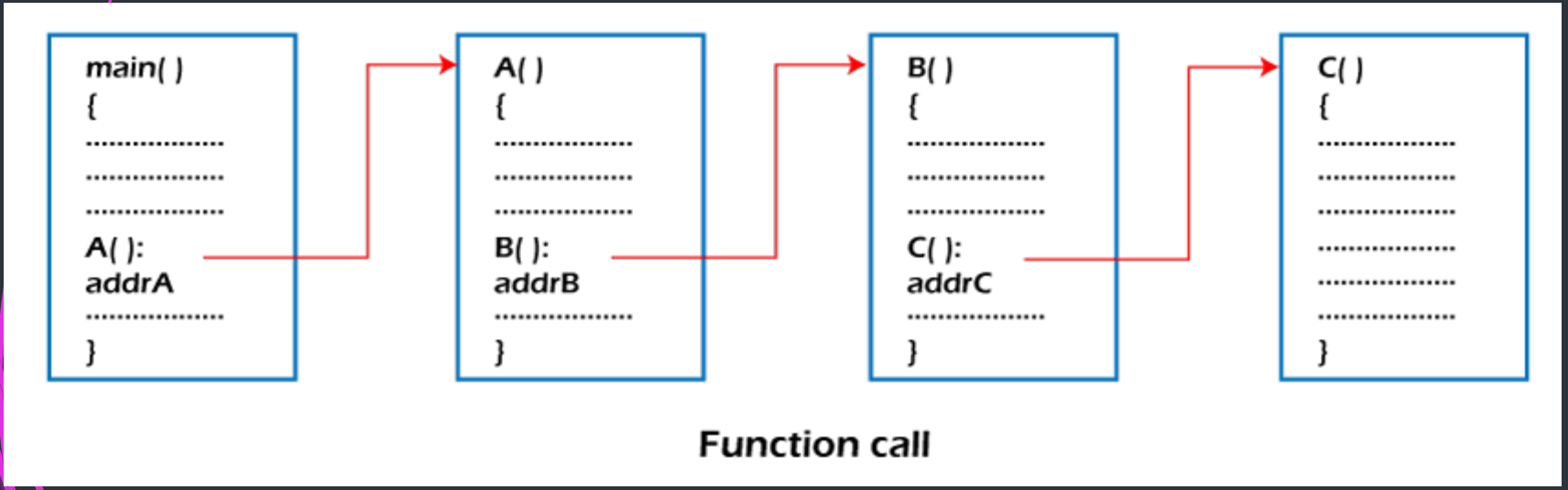
Converting Decimal to Binary:

- **Example: Converting 14 number Decimal to Binary:**

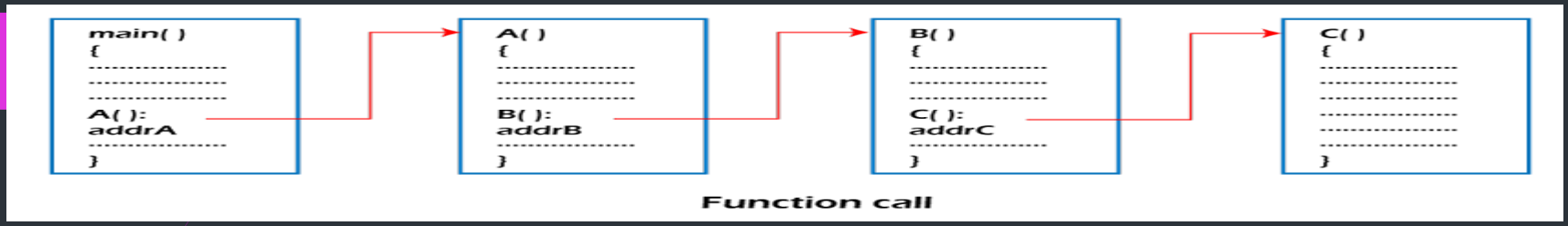


Processing Function Calls:

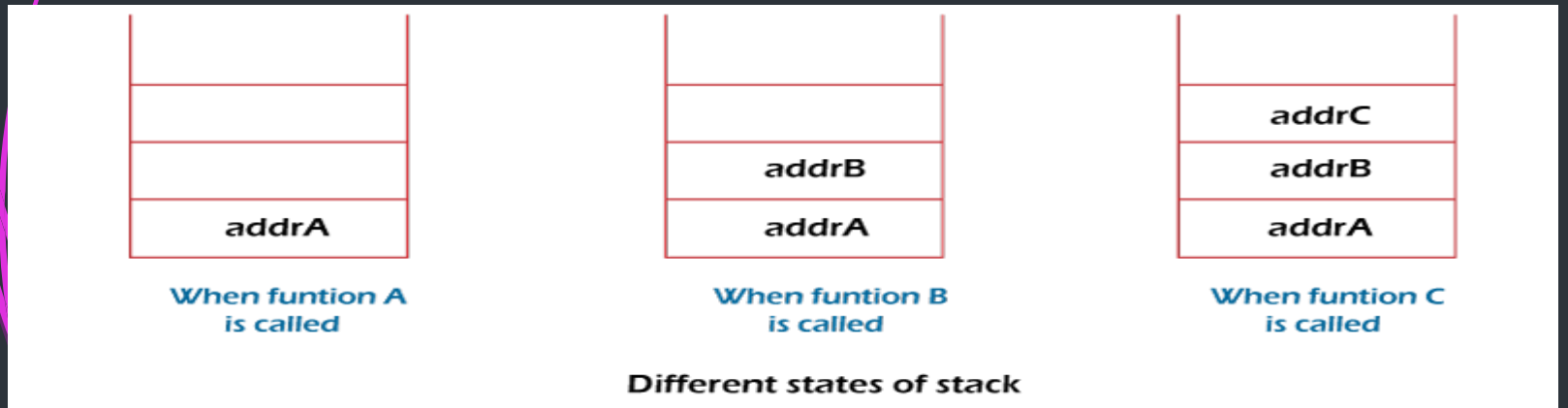
- Stack plays an important role in programs that call several functions in succession.
- Suppose we have a program containing three functions: A, B, and C. function A invokes function B, which invokes the function C.



Processing Function Calls:



- When we invoke function A, which contains a call to function B, then its processing will not be completed until function B has completed its execution and returned.
- Similarly for function B and C. So we observe that function A will only be completed after function B is completed and function B will only be completed after function C is completed.
- Therefore, function A is first to be started and last to be completed. To conclude, the above function activity matches the last in first out behavior and can easily be handled using Stack.



Stack Applications

- Call Stack in C:
  - Primary task of Function Call Stack in C is to manage the function calls and how they pass parameters to each other.
  - It is also called an execution stack or machine stack. More often, it is simply known as a stack.
  - Now let's look at how function calls are actually organized in a stack: Let's assume we have two functions f1() and f2() along with main().

```c
#include<stdio.h>

void f2() {
  return;
}

void f1() {
  f2();  //calling f2()
  return;
}

//This is main function
int main() {
  f1();  // calling f1()
}
```

On running the program, the main() is called, so an activation record for main() is created and added to the stack. Now, main() calls f1(), which creates an activation record for f1() on top of stack and f1() calls f2() by adding activation record for f2() on top of stack.

When f2() terminates, its activation record is removed (popped) from the stack. After completing the execution of f1(), it returns by removing the activation record from the stack. At this stage, we are back to our main() which removes its activation record leading to the termination of the program.