Data Structures and Algorithms

Mr. Tahir Iqbal

tahir.iqbal@bahria.edu.pk

Lecture 06: Infix and Postfix

Infix, Postfix and Prefix Notations

Evaluating arithmetic expressions.

Prefix: + a b

Infix: a + b

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 Notation

- 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)

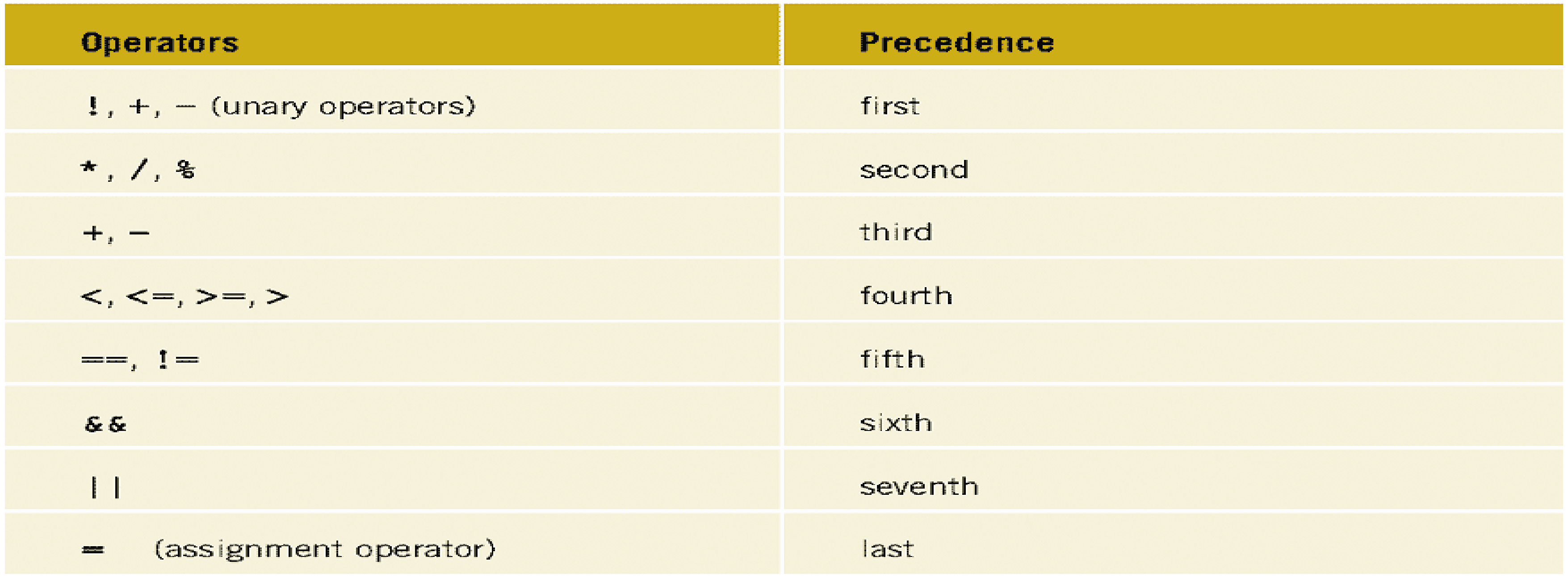
  or 13 – (5 / 3) + 1

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

- 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



Operators	Precedence
!, +, – (unary operators)	first
*, /, %	second
+, –	third
<, <=, >=, >	fourth
==, !=	fifth
&&	sixth
||	seventh
= (assignment operator)	last

Self Practice

- Solve the following infix expressions:
  - A) x = 7 + 3 * 6 / 2 – 1;
  - B) x = 2 % 2 + 2 * 2 – 2 / 2;
  - C) x = (3 * 9 * (3 + (9 * (3 / 3))));
  - D) x = 2^3+5*2^2-12/6

Prefix Notation

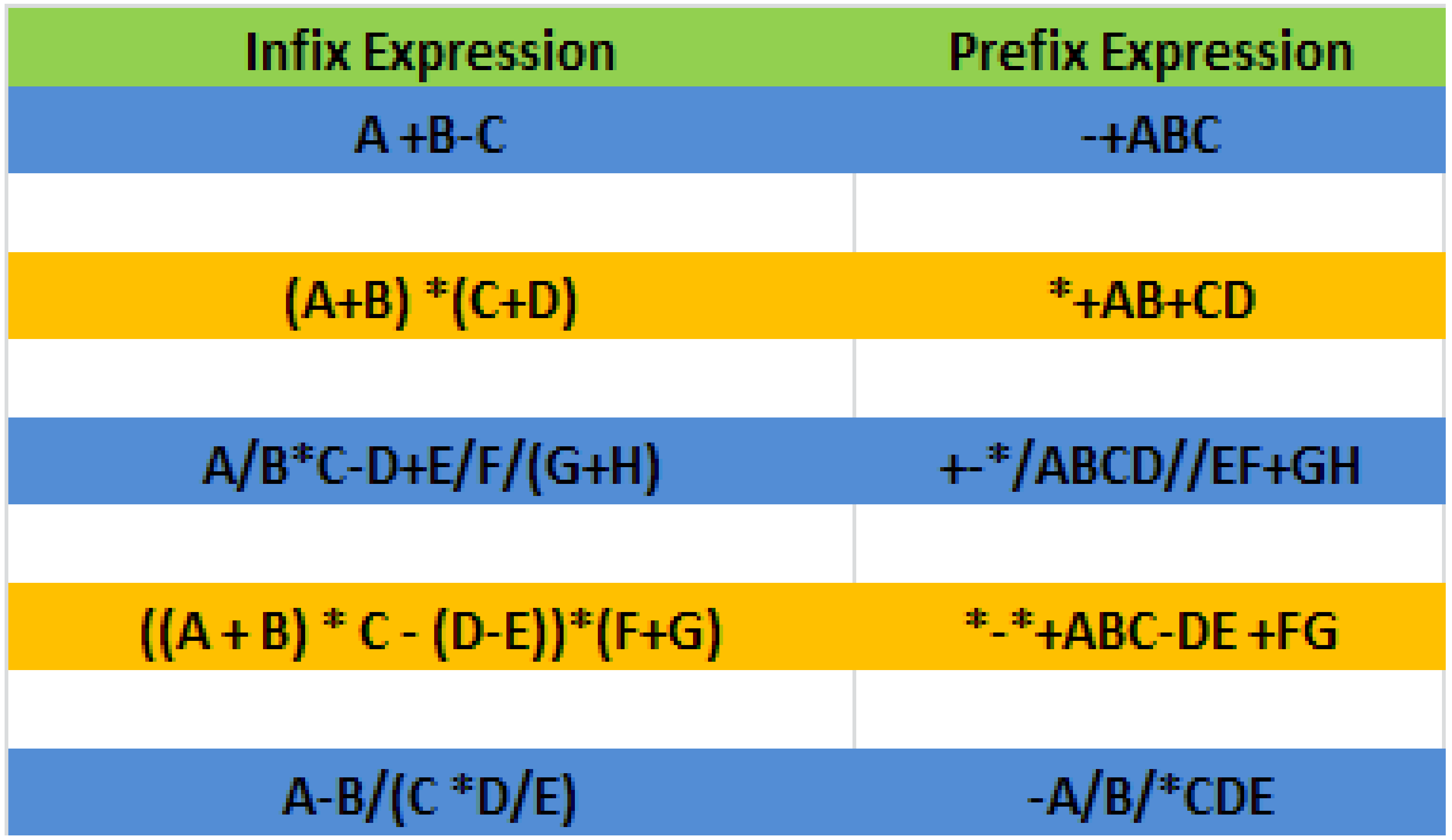
- It is a notation for writing arithmetic expressions in which operators symbol appears **before** the operands.
- E.g. A + B is written as + A B in prefix notation
- There are no precedence rules to be learnt in it.
- Parentheses are never needed.
- This is also called the “Polish Notation or PN”.

Simple Infix to prefix conversion

- Apply the rules of precedence and convert the portion of the expression that is evaluated first.

- ((A+B)*C-(D-E))^(F+G)
- (+AB*C-(D-E))^(F+G)
- (+AB*C--DE)^(F+G)
- (*+ABC--DE)^(F+G)
- -*+ABC-DE^(F+G)
- -*+ABC-DE^+FG
- ^ -*+ABC-DE+FG

Examples of infix to prefix conversion



Infix Expression	Prefix Expression
A +B-C	-+ABC
(A+B) *(C+D)	*+AB+CD
A/B*C-D+E/F/(G+H)	+-*/ABCD//EF+GH
((A + B) * C - (D-E))*(F+G)	*-*+ABC-DE +FG
A-B/(C *D/E)	-A/B/*CDE

Self Practice

- Convert the following infix expressions into prefix:
  - A+B-C
  - (A+B)*(C-D)^E*F
  - (A+B)*(C^(D-E)+F)-G
  - A+(((B-C)*(D-E)+F)/G)^(H-J)

Postfix Notation

- It is a notation for writing arithmetic expressions in which operators appear **after** the operands.
- 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”

Simple Infix to postfix conversion

- Apply the rules of precedence and convert the portion of the expression that is evaluated first.

- ((A+B)*C-(D-E))^(F+G)
- (AB+*C-(D-E))^(F+G)
- (AB+*C-DE-)^(F+G)
- (AB+C*-DE-)^(F+G)
- AB+C*DE--^(F+G)
- AB+C*DE--^FG+
- AB+C*DE--FG+^ ______________ ^ -*+ABC-DE+FG

(Not necessarily the mirror image of PN)

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 * /

Self Practice

- Convert the following infix expressions into postfix:
  - A+B-C
  - (A+B)*(C-D)^E*F
  - (A+B)*(C^(D-E)+F)-G
  - A+(((B-C)*(D-E)+F)/G)^(H-J)

Expression representation examples

Infix	Prefix	Postfix
a + b	+ a b	a b +
a + b * c	+ a * b c	a b c * +
(a + b) * (c - d)	* + a b - c d	a b + c d - *

Postfix Evaluation

Postfix Evaluation vs Infix Evaluation

- 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

Evaluation of Postfix Expression

- After an infix expression is converted to postfix, its evaluation is a simple affair and Stack comes in handy, AGAIN

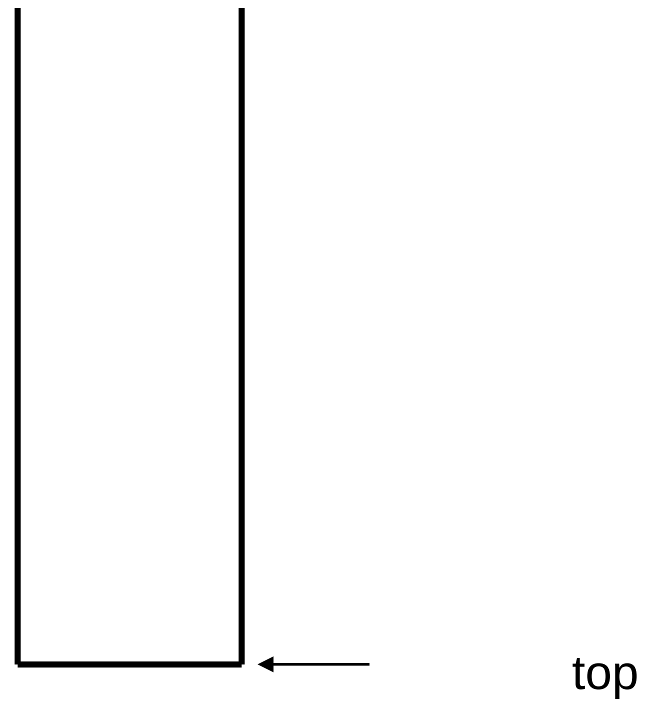
- <u>The Algorithm</u>

```
1. Create STACK.
2. While end of expression, Scan postfix expression RP from left to right and
repeat following:
        2.1 If an Operand is encountered, then:
                2.1.1 PUSH(STACK, Operand).
        2.2 Else if an Operator is encountered, then:
                2.2.1 SET OP2=POP(STACK).
                2.2.2 SET OP1=POP(STACK).
                2.2.3 SET Value = Evaluate(OP1, OP2, Operator).
                2.2.4 PUSH(STACK, Value).
3. Return POP(STACK).
```

Stack in Action ….

Postfix Expression

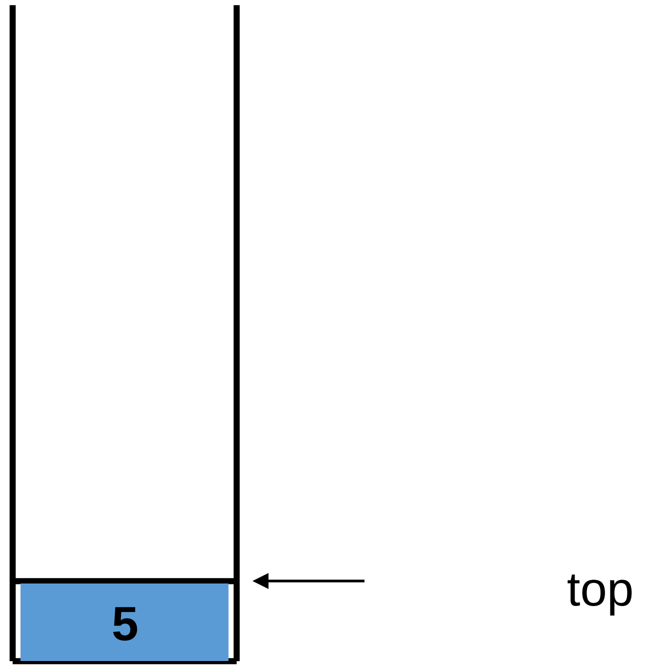
5 7 + 6 2 - *



Stack in Action ….

Postfix Expression

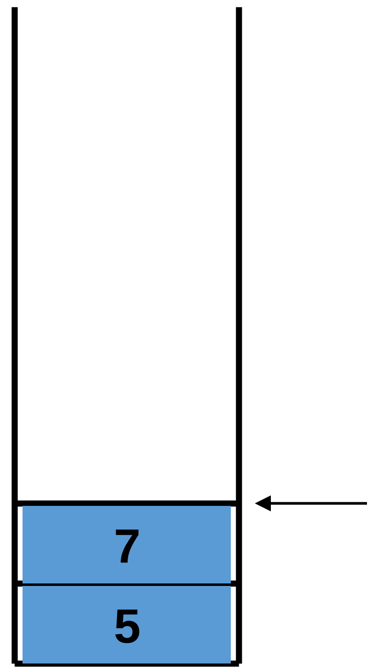
5 7 + 6 2 - *



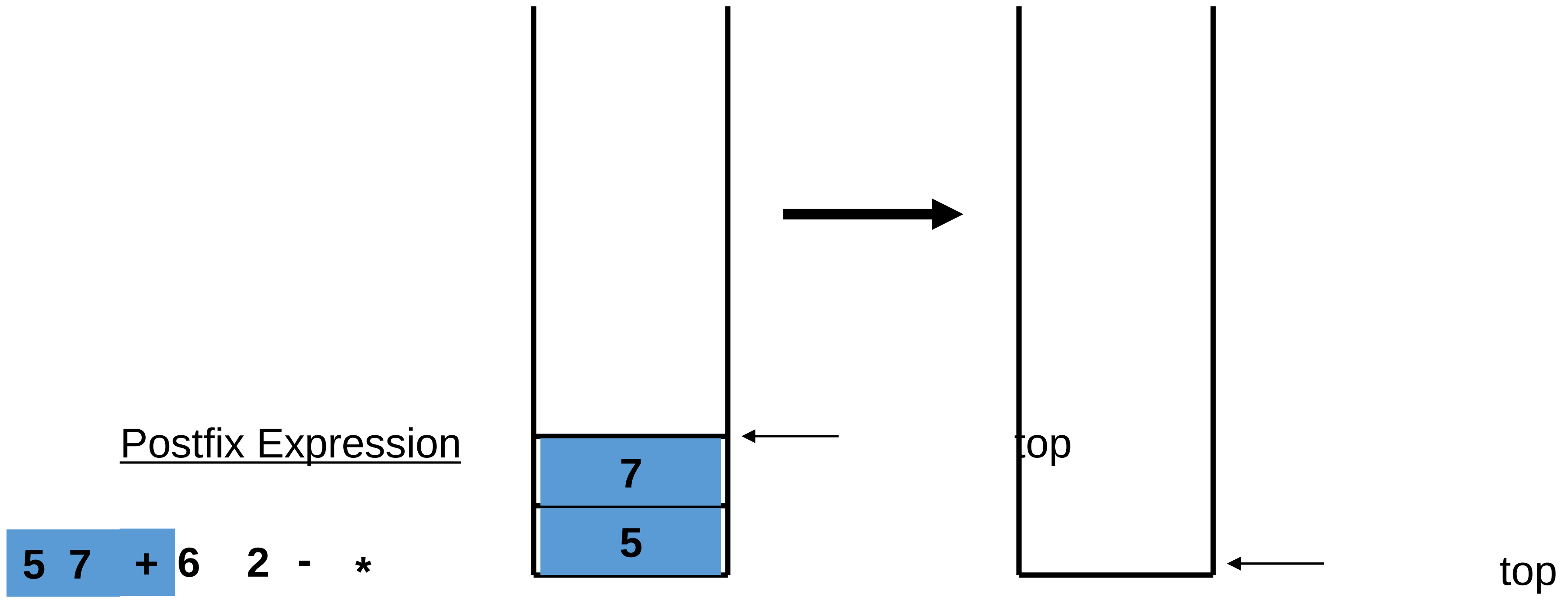
Stack in Action ….

Postfix Expression

5 7 + 6 2 - *



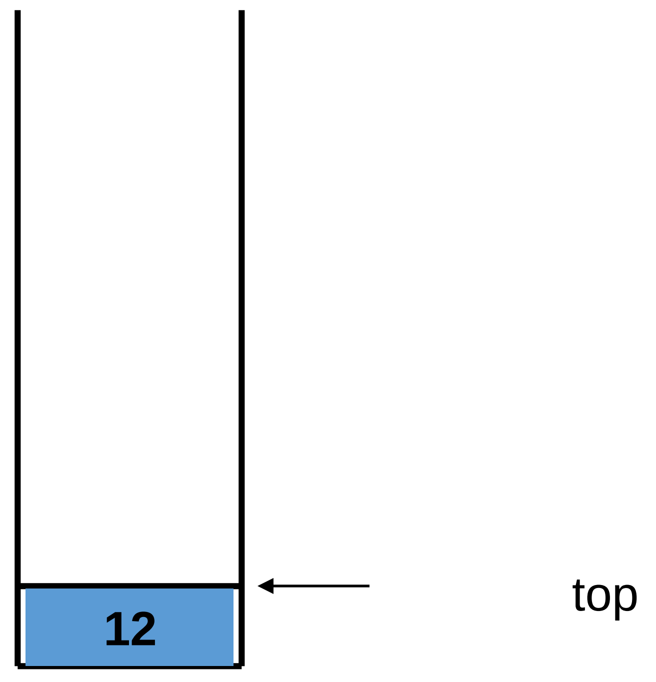
Stack in Action ….



Result = Pop( ) "+" Pop( )

Push (Result)

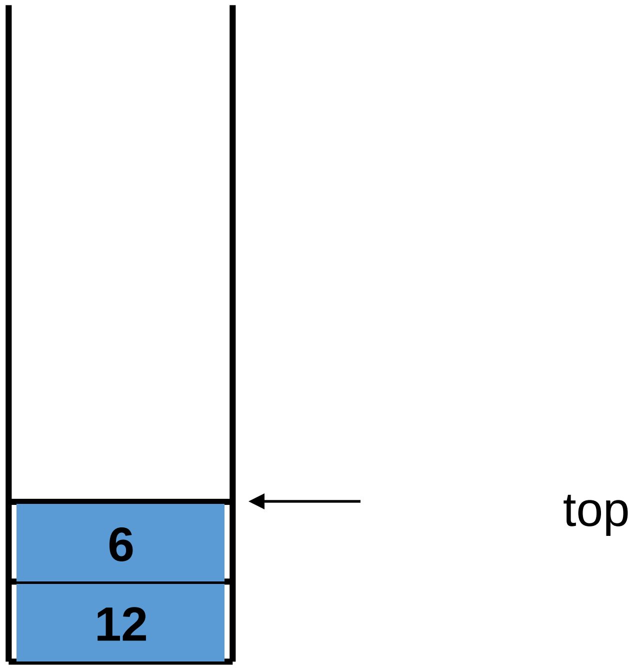
Stack in Action ….



Postfix Expression

5 7 + 6 2 - *

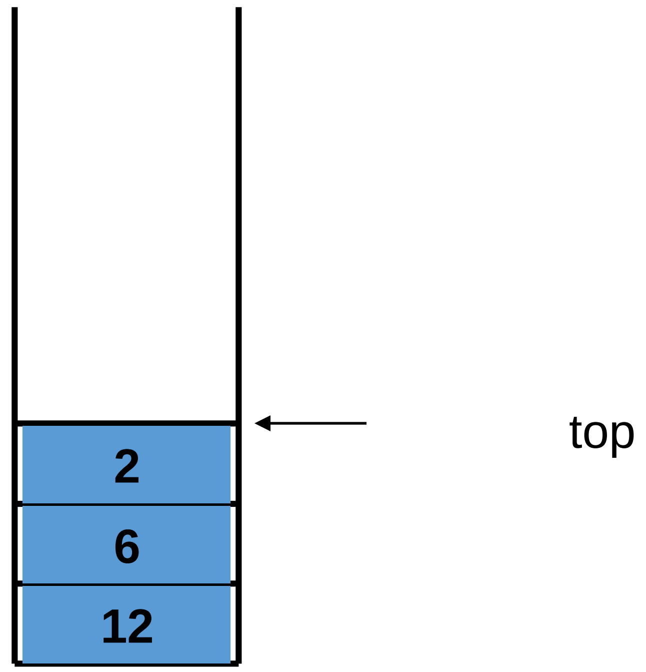
Stack in Action ….



Postfix Expression

5 7 + 6 2 - *

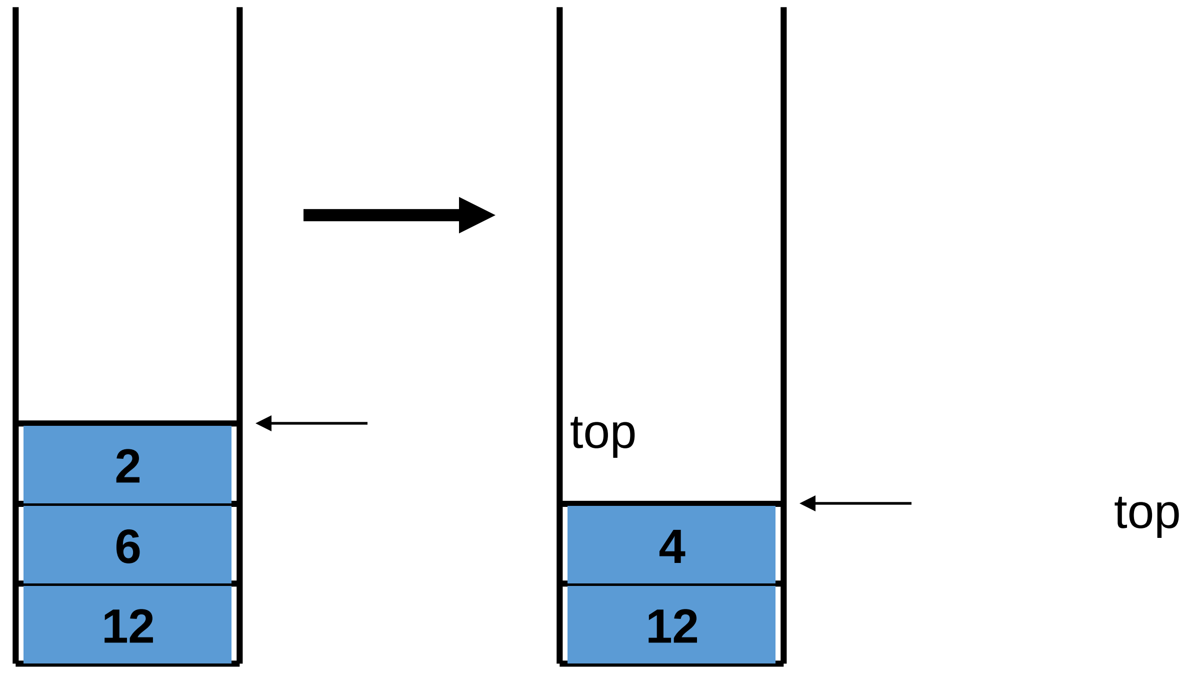
Stack in Action ….



Postfix Expression

5 7 + 6 2 - *

Stack in Action ….



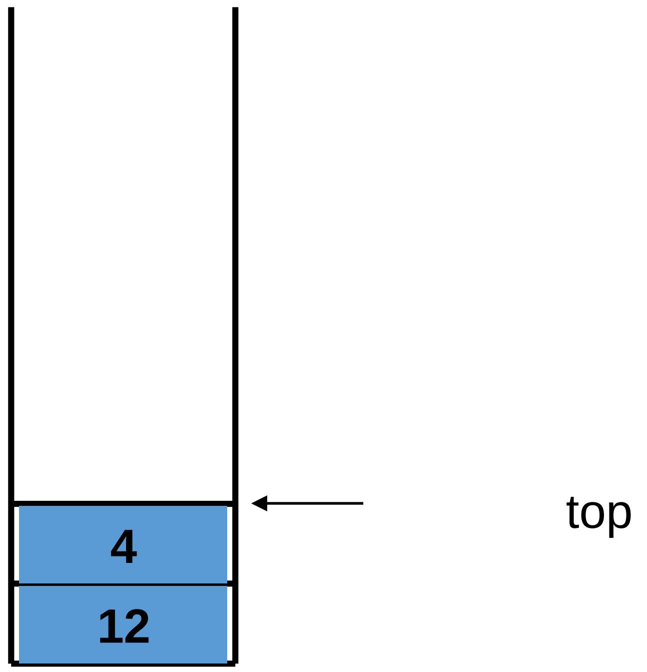
Postfix Expression

5 7 + 6 2 - *

Result = Pop( ) “-” Pop( )

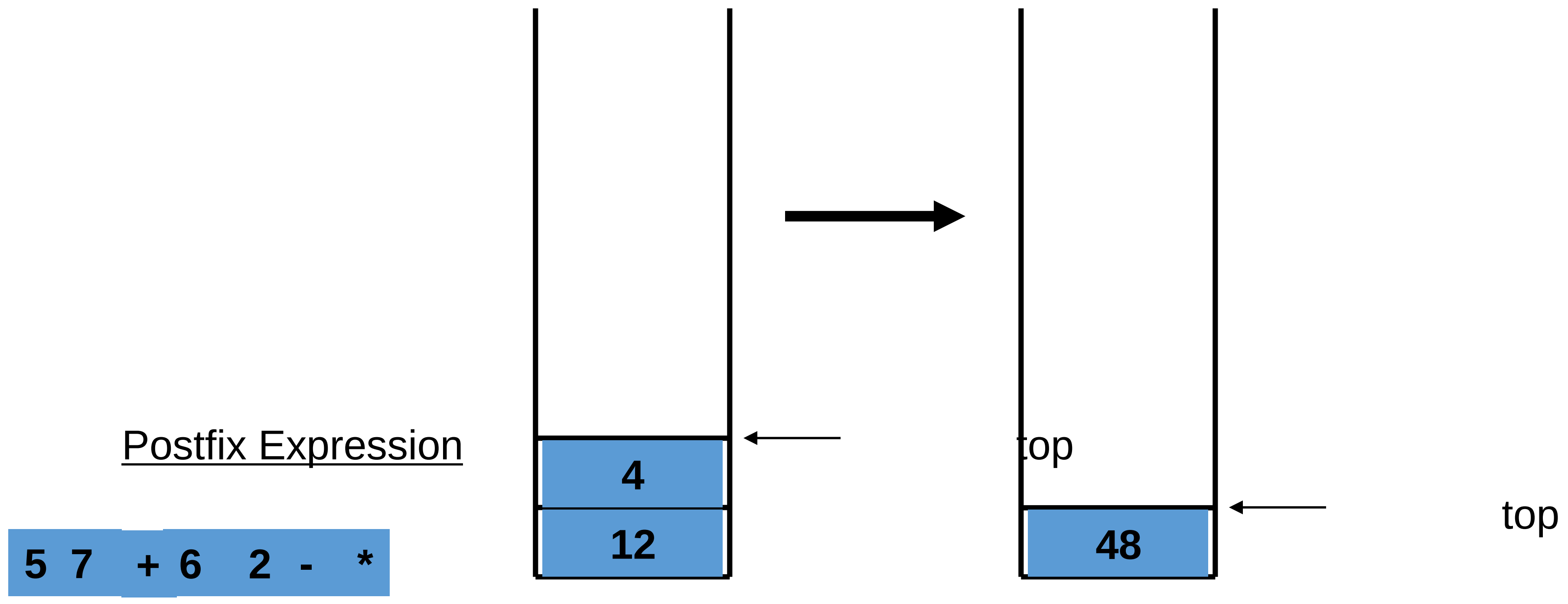
Push (Result)

Stack in Action ….

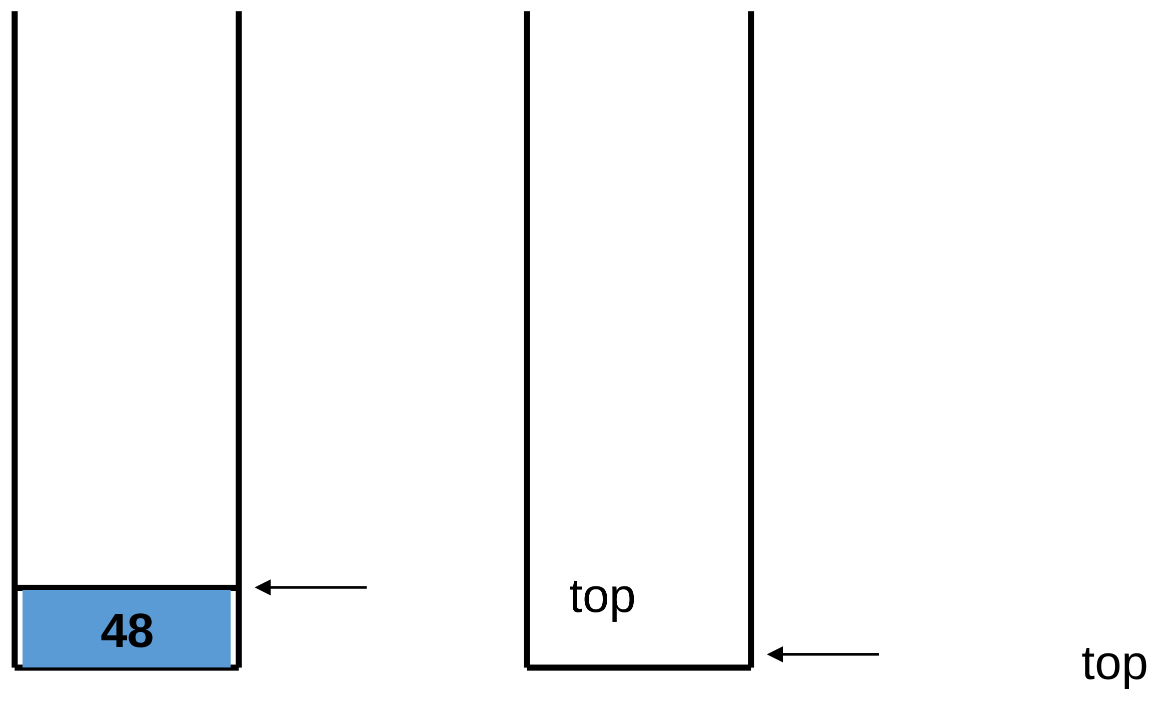


Postfix Expression

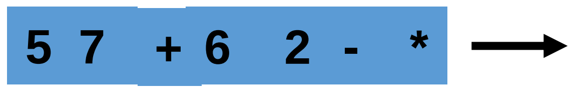
5 7 + 6 2 - *



Result = Pop( ) " * " Pop( )   Push (Result)



Postfix Expression



Result = Pop( )

Result = 48

Self Practice

- Applying the Postfix evaluation algorithm solve the following expressions:
- Assume A= 1, B= 2, C=3
    - AB+C-BA+C^-
    - ABC+*CBA-+*

Stack class for Evaluation

```
#include<iostream>
#include<conio.h>
#include<math.h>
using namespace std;
#define size 100
class Evaluation
{
        private:
            int stack[size];
            int top;
        public:
            Evaluation()
          {
           top=-1;
          }
            void push(char);
            char pop();
            int pEvaluation();
};
```

Push & pop functions

```
void Evaluation::push(char item)
{
        if (top == size-1)
            cout<<"\nThe Stack Is Full";
        else
            stack[++top]=item;
}
```

```
char Evaluation::pop()
{
        char item;
        if (top == -1)
            cout<<"\nThe Stack Is Empty, Invalid
            Infix expression";
        else
            item=stack[top--];
        return item ;
}
```

Evaluation function

```
int Evaluation::pEvaluation()
{
    int x, y, temp, length;
    char postfix[size];
    cout<<"\n\nEnter the Postfix expression = ";
    cin>>postfix;
    length=strlen(postfix);
    for(int i=0;i<length;i++)
    {
            if (postfix[i]>='0'&& postfix[i]<='9')
                    push((int)postfix[i]-'0');
            else
            {
                    x=pop();
                    y=pop();
```

```
    switch(postfix[i])
    {
            case'+':
                temp = y+x; break;
            case'-':
                temp = y-x; break;
            case'*':
                temp = y*x; break;
            case'/':
                temp = y/x; break;
            case'%':
                temp = y%x; break;
            case '^':
                temp = pow(y,x); break;
    }
    push(temp);
}
}
    return pop() ;
}
```

Main function

```
void main()
{
    Evaluation pfix;
    int Result;
    Result=pfix.pEvaluation();
    cout<<"\n\nThe postfix evaluation is = "<<Result;
    getch();
}
```

Infix to Postfix Conversion

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

Conversion from Infix to Postfix Notation

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

Infix to Postfix Conversion Algorithm

```
1. Create STACK.
2. PUSH(STACK,“(” )
3. Add(Q, “)”) (to the end of the infix expression Q).
4. While STACK is not empty, scan Q from left to right and repeat following
Steps:
    4.1 If an Operand is encountered, then:
        4.1.1 Add it to postfix expression P.
    4.2 Else if a “(” is encountered, then:
        4.2.1 PUSH(STACK, “(” ).
    4.3 Else if an Operator is encountered, then:
        4.3.1 While (Operator<STACK[top])
            4.3.1.1 Add(P, POP(STACK)).
            4.3.1.2 PUSH(STACK, Operator).
        4.3.2 PUSH(STACK, Operator).
    4.4 Else if “)” is encountered, then:
```

Infix to Postfix Conversion Algorithm

```
        4.4.1 While ( !“(” ), repeat following:
              4.4.1.1 Add(P, POP(STACK)).
        4.4.2 Delete( “(” ).
5. Return.
```

Infix to Postfix Example

infix	stack	postfix
a) A	(	A
b) +	(+	A
c) (	(+ (	A
d) B	(+ (	A B
e) *	(+ (*	A B
f) C	(+ (*	A B C
g) -	(+ (-	A B C *
h) (	(+ (- (	A B C *
i) D	(+ (- (	A B C * D
j) /	(+ (- (/	A B C * D
k) E	(+ (- (/	A B C * D E
l) ^	(+ (- (/ ^	A B C * D E
m) F	(+ (- (/ ^	A B C * D E F
n))	(+ (-	A B C * D E F ^ /
o) *	(+ (- *	A B C * D E F ^ /
p) G	(+ (- *	A B C * D E F ^ / G
q))	(+	A B C * D E F ^ / G * -
r) *	(+ *	A B C * D E F ^ / G * -
s) H	(+ *	A B C * D E F ^ / G * - H
t))		A B C * D E F ^ / G * - H * +

Self Practice

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

Stack class for conversion

```
#define size 100
class conversion
{
          private:
               int stack[size];
               int top;
          public:
               conversion()
            {
             top=-1;
            }
               void push(char);
               char pop();
            int p(char);
           void iTop();
};
```

Push & pop functions

```
void conversion::push(char item)
{
        if (top == size-1)
            cout<<"\nThe Stack Is Full";
        else
            stack[++top]=item;
}
```

```
char conversion::pop()
{
        char item;
        if (top == -1)
            cout<<"\nThe Stack Is Empty, Invalid
            Infix expression";
        else
            item=stack[top--];
        return item ;
}
```

Conversion function

```
int conversion::p(char a)
{
    switch(a)
    {
            case'(':
                    return 1 ;
            case')':
                    return 2;
            case'+‘: case'-':
                    return 3 ;
            case'*‘: case'/‘: case'%':
                    return 4;
            case'^':
                    return 5 ;
            default:
                    return 0 ;
    }
}
```

```
void conversion::iTop()
{
        int length, priority;
        char infix[size], postfix[size], ch;
        cout<<"\n\nEnter the infix expression = : ";
        cin>>infix;
        length=strlen(infix);
        infix[length++]=')';
        push('(');
        for(int i=0,j=0; i<length ;i++)
    {
          switch(p(infix[i]))
          {
              case 1:
                    push(infix[i]);
                    break;
              case 2:
                    ch=pop();
```

Conversion function

```
        while(ch != '(')
        {
                postfix[j++]=ch;
                ch=pop();
            }
            Break;
    case 3:
            ch=pop();
            while(p(ch) >= 3)
            {
                postfix[j++]=ch;
                ch=pop();
            }
            push(ch);
            push(infix[i]);
            break;
   case 4:
            ch=pop();
            while(p(ch) >= 4)
            {postfix[j++]=ch;
            ch=pop();
            }
```

```
            push(ch);
            push(infix[i]);
            break;
    case 5:
            ch=pop();
            while(p(ch) == 5)
            {
            postfix[j++]=ch;
            ch=pop();
            }
            push(ch);
            push(infix[i]);
            break;
    default:
            postfix[j++]=infix[i];
            break;
    }
}
cout<<"\n Postfix expression is = ";
for(int i = 0; i<length-1; i++)
cout<<postfix[i];
}
```

Main function

```
void main()
{
    conversion in;
    in.iTop();
    getch();
}
```

Prefix/Postfix to infix conversion (Self Practice)

- Convert the following prefix notations into infix:
  - +-ABC
  - ++A-*^BCD/+EF*GHI

- Convert the following postfix notations into infix:
  - ABC+-
  - ABCDE-+^*EF*-