UNIT 3 · STACKS

11. Stack Applications: Expression Conversion

CSC211 · Algorithms and Data Structures

COMSATS University Islamabad, Lahore Campus

Agenda

1. Real applications of stacks: calls, backtracking, undo/redo
2. Infix, prefix, and postfix notation
3. Operator precedence and associativity
4. Converting infix to postfix using a stack, traced token by token
5. Evaluating a postfix expression using a stack

Applications of Stacks

Function call management

Every call pushes a stack frame (locals, return address) onto the call stack; returning pops it. This is why it's called a call stack.

Backtracking

Algorithms that try a path and “undo” it if it fails (maze solving, Sudoku) push each decision, then pop back to the last decision point on a dead end.

Undo / redo

Every action is pushed onto an undo stack; undoing pops the most recent action and reverses it.

Expression Notation

Notation	Operator position	Example: “3 plus 4”
Infix	Between the two operands	3 + 4
Prefix	Before the two operands	+ 3 4
Postfix	After the two operands	3 4 +

Infix is what humans write naturally — but it's hard for a computer to evaluate directly, since it must know precedence and parentheses. Postfix needs neither: evaluated left to right with just a stack.

Operator Precedence and Associativity

Operator	Precedence	Associativity
^ (exponent)	Highest	Right to left
*, /	Middle	Left to right
+, -	Lowest	Left to right

Associativity is the tie-breaker for equal precedence.

`10 - 3 - 2` evaluates left to right: `(10 - 3) - 2 = 5`, because `-` is left-associative.

Infix → Postfix: The Algorithm

`operand`	Append directly to output
`'('`	Push onto the stack
`')'`	Pop + output until '(' is popped, then discard it
`operator`	Pop + output while stack top has ≥ precedence, then push this operator

After scanning the whole expression, pop and output any remaining operators.

Tracing 3+4*2, Token by Token

Token	Action	Stack (bottom→top)	Output
3	operand: append	(empty)	3
+	stack empty, push	+	3
4	operand: append	+	3 4
*	'+' has lower precedence — don't pop, push	+ *	3 4
2	operand: append	+ *	3 4 2
(end)	pop everything remaining: *, then +	(empty)	3 4 2 * +

'' ends up before '+' in the postfix result even though '+' appears first: '*' was pushed after '+' (binds tighter), so it's popped and output first once the stack unwinds.*

The Conversion Loop

infix_to_postfix.cpp — core loop

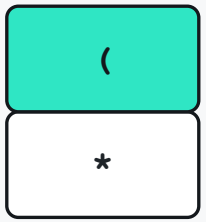
```
for (char token : infix) {
    if (isdigit(token)) {
        postfix << token;
    } else if (token == '(') {
        operators.push(token);
    } else if (token == ')') {
        while (!operators.empty() && operators.top() != '(') {
            postfix << ' ' << operators.top();  operators.pop();
        }
        operators.pop();   // discard the matching '('
    } else {
        while (!operators.empty() &&
               precedence(operators.top()) >= precedence(token)) {
            postfix << operators.top() << ' ';  operators.pop();
        }
        operators.push(token);
    }
}
```

precedence('(') returns 0 — the lowest possible — so an operator never pops past an open paren while scanning.

A Harder Trace: 5*(3+2)-8/4^2

Exercises every rule at once: a (...) pair that must fully resolve first, a lower-precedence '-' waiting behind a completed multiplication, and a right-associative '^'.

After '(' (step 3)



'+' pushed inside, then ')' pops it and discards '('

After ')' resolves (step 7)

'' left sitting from step 2, waiting through the (...) group*

After '-' pushes (step 8)

'' has higher precedence than '-': popped + output first*

`Final result: 5 3 2 + * 8 4 2 ^ / -  — matches the compiled program's real output exactly.`

Postfix Expression Evaluation

Scan left to right, push every operand; on an operator, pop the top two, apply it, push the result back.

postfix_eval.cpp — evaluatePostfix

```
while (tokens >> token) {
    if (isdigit(token[0])) {
        values.push(stoi(token));
    } else {
        int b = values.top(); values.pop();
        int a = values.top(); values.pop();
        int result = 0;
        switch (token[0]) {
            case '+': result = a + b; break;
            case '-': result = a - b; break;   // order matters!
            // ... '*' and '/' the same way
        }
        values.push(result);
    }
}
return values.top();
```

Operand order matters for '-' and '/': b was pushed last, so a - b (not b - a) is the correct order.

Tracing 5 1 2 + 4 * + 3 -

Token	Action	Stack (bottom→top)
5, 1, 2	push operands	5 1 2
+	pop 2,1 → 1+2=3, push 3	5 3
4	push operand	5 3 4
*	pop 4,3 → 3*4=12, push 12	5 12
+	pop 12,5 → 5+12=17, push 17	17
3	push operand	17 3
-	pop 3,17 → 17-3=14, push 14	14

14

final stack holds exactly one value — the answer to 5 + (1 + 2) * 4 - 3

Key Takeaways

- Prefix, infix, and postfix are three ways to write the same expression, differing only in where the operator sits relative to its operands.
- Postfix can be evaluated left to right using only a stack, with no precedence rules or parentheses needed — exactly why compilers convert to it internally.
- Infix-to-postfix conversion holds operators on a stack until a lower-or-equal precedence operator forces earlier ones out — parentheses act as a hard wall nothing can pop past.
- A harder expression like 5*(3+2)-8/4^2 shows parentheses, mixed precedence, and right-associativity all at once — tracing the stack token by token makes it concrete.
- Postfix evaluation is the mirror operation: push operands, and on each operator pop two, apply it in the right order, and push the result back.