Chapter 8: Stack and Queue Applications

Data Structures and Algorithms with C

Maze · Backtracking · Operator precedence · Infix / postfix / prefix · Postfix evaluation · Bracket matching

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/dsa_c/slides/Chapter_08_Stack_and_Queue_Applications.html

Learning Outcomes

Code backtracking

Search a maze with a stack and trace how it backs out of dead ends.

Build the worst case

Design and analyze the worst-case input for an algorithm.

Convert notations

Convert among infix, prefix, and postfix by hand and with a stack.

Evaluate postfix

Run `eval` step by step with its operand stack.

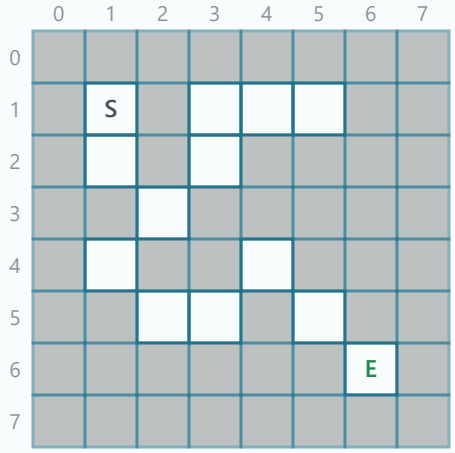
Encode precedence

Write the infix → postfix routine using the isp/icp tables.

Generalize the stack

Use stacks for bracket matching, undo/redo, and non-recursive traversal.

The Maze Problem



In an m×p matrix, **0**s are open paths and **1**s are walls. The entrance is [1][1] (S) and the exit [6][6] (E): we want a path from entrance to exit.

- Moves are horizontal, vertical, and **diagonal**: 8 directions.
- To avoid checking "am I leaving the matrix?" at every step, the matrix is **surrounded by a border of 1s**.
- Visited cells are marked in the `mark` matrix, so no cell is entered twice.
- At a dead end we **back up** to the most recent fork: last in, first out → a stack.

Eight Directions and the `move[]` Table

dir	move[dir].vert	move[dir].horiz
0	−1	0
1	−1	1
2	0	1
3	1	1
4	1	0
5	1	−1
6	0	−1
7	−1	−1



Direction 0 is north; 1…7 go clockwise. New position: [i + vert][j + horiz].

An eight-entry **look-up table** saves writing eight separate `if` s: the direction becomes a loop variable.

Definitions for the Maze

```
#define MAX_STACK_SIZE 100
#define EXIT_ROW 6
#define EXIT_COL 6
typedef struct { short int vert, horiz; } offsets;
offsets move[8] = {{-1, 0}, {-1, 1}, {0, 1}, {1, 1},
                   {1, 0}, {1, -1}, {0, -1}, {-1, -1}};
typedef struct { short int row, col, dir; } element;
element stack[MAX_STACK_SIZE];
int top = -1;
short int mark[8][8];
short int maze[8][8] = {{1,1,1,1,1,1,1,1},
    {1,0,1,0,0,0,1,1}, {1,0,1,0,1,1,1,1},
    {1,1,0,1,1,1,1,1}, {1,0,1,1,0,1,1,1},
    {1,1,0,0,1,0,1,1}, {1,1,1,1,1,1,0,1},
    {1,1,1,1,1,1,1,1}};
```

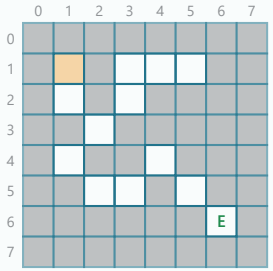
- `element` : the record pushed on the stack: **which cell** (row, col) and, if we come back, **which direction** to try next (dir).
- `mark` : visited cells (zero at the start because it is global).
- `maze` : an 8×8 matrix with a border of 1s; the interior is 6×6.
- The stack never holds more entries than there are open cells (13 here).

The `path` Routine

```
void path(void) {
  short int row, col, next_row, next_col;
  int i, dir, found = FALSE;
  element position;
  stack[0].row = stack[0].col = 1;
  stack[0].dir = 1;     /* push the start */
  mark[1][1] = 1;  top = 0;
  while (top > -1 && !found) {
    position = pop(&top);     /* next one */
    row = position.row;
    col = position.col;
    dir = position.dir;       /* which way */
    while (dir < 8 && !found) {
      next_row = row + move[dir].vert;
      next_col = col + move[dir].horiz;
      if (next_row == EXIT_ROW &&
          next_col == EXIT_COL)
        found = TRUE;         /* the exit? */
      else if (!maze[next_row][next_col] &&
               !mark[next_row][next_col]) {
        mark[next_row][next_col] = 1;
        position.row = row;   /* came from */
        position.col = col;
        position.dir = ++dir; /* resume dir */
        push(&top, position);
        row = next_row;  col = next_col;
        dir = 0;
      } else ++dir;           /* try another */
    }                         /* back up */
  }
```

If `found`, the stack positions (0…top), the last position and the exit are printed; otherwise “The maze does not have a path”. (Lines split for readability; logic unchanged.)

Code Trace: Backtracking with a Stack



Start: the stack holds **(1,1) dir 1**. `stack`: row,col,dir records (bottom to top).

Step 1 / 18

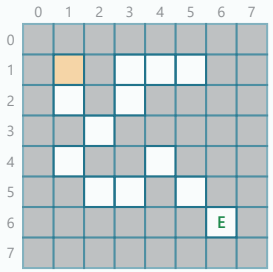
VARIABLES

`found = 0` `stack = 1,1,1`

OUTPUT

```
while (top > -1 && !found) {
  position = pop(&top);
  row = position.row;  col = position.col;
  dir = position.dir;
  while (dir < 8 && !found) {
    nr = row + move[dir].vert;
    nc = col + move[dir].horiz;
    if (nr == EXIT_ROW && nc == EXIT_COL)
      found = TRUE;
    else if (!maze[nr][nc] &&
             !mark[nr][nc]) {
      mark[nr][nc] = 1;
      position = (element){row, col, ++dir};
      push(&top, position);
      row = nr;  col = nc;  dir = 0;
    } else ++dir;
} }
```

Code Trace: Backtracking with a Stack



First **pop**: (1,1), starting at dir 1 (dir 0 is a wall anyway).

Step 2 / 18

VARIABLES

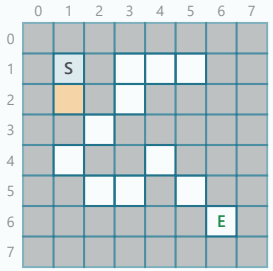
`row = 1` `col = 1` `dir = 1`

`stack = empty`

OUTPUT

```
while (top > -1 && !found) {
  position = pop(&top);
  row = position.row;  col = position.col;
  dir = position.dir;
  while (dir < 8 && !found) {
    nr = row + move[dir].vert;
    nc = col + move[dir].horiz;
    if (nr == EXIT_ROW && nc == EXIT_COL)
      found = TRUE;
    else if (!maze[nr][nc] &&
             !mark[nr][nc]) {
      mark[nr][nc] = 1;
      position = (element){row, col, ++dir};
      push(&top, position);
      row = nr;  col = nc;  dir = 0;
    } else ++dir;
} }
```

Code Trace: Backtracking with a Stack



From (1,1), dir 4 (S) is open → move to **(2,1)**; (1,1, dir 5) pushed.

Step 3 / 18

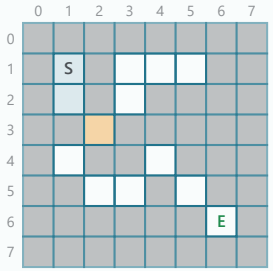
VARIABLES

`row = 2` `col = 1` `dir = 0` `stack = 1,1,5`

OUTPUT

```
while (top > -1 && !found) {
  position = pop(&top);
  row = position.row;  col = position.col;
  dir = position.dir;
  while (dir < 8 && !found) {
    nr = row + move[dir].vert;
    nc = col + move[dir].horiz;
    if (nr == EXIT_ROW && nc == EXIT_COL)
      found = TRUE;
    else if (!maze[nr][nc] &&
             !mark[nr][nc]) {
      mark[nr][nc] = 1;
      position = (element){row, col, ++dir};
      push(&top, position);
      row = nr;  col = nc;  dir = 0;
    } else ++dir;
} }
```

Code Trace: Backtracking with a Stack



From (2,1), dir 3 (SE) is open → move to **(3,2)**; (2,1, dir 4) pushed.

Step 4 / 18

VARIABLES

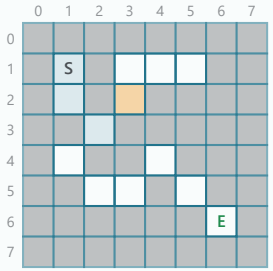
`row = 3` `col = 2` `dir = 0`

`stack = 1,1,5 2,1,4`

OUTPUT

```
while (top > -1 && !found) {
  position = pop(&top);
  row = position.row;  col = position.col;
  dir = position.dir;
  while (dir < 8 && !found) {
    nr = row + move[dir].vert;
    nc = col + move[dir].horiz;
    if (nr == EXIT_ROW && nc == EXIT_COL)
      found = TRUE;
    else if (!maze[nr][nc] &&
             !mark[nr][nc]) {
      mark[nr][nc] = 1;
      position = (element){row, col, ++dir};
      push(&top, position);
      row = nr;  col = nc;  dir = 0;
    } else ++dir;
} }
```

Code Trace: Backtracking with a Stack



From (3,2), dir 1 (NE) is open → move to **(2,3)**; (3,2, dir 2) pushed.

Step 5 / 18

VARIABLES

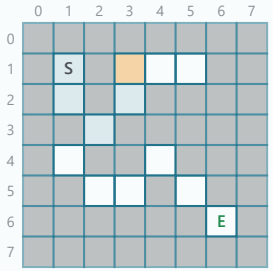
`row = 2` `col = 3` `dir = 0`

`stack = 1,1,5 2,1,4 3,2,2`

OUTPUT

```
while (top > -1 && !found) {
  position = pop(&top);
  row = position.row;  col = position.col;
  dir = position.dir;
  while (dir < 8 && !found) {
    nr = row + move[dir].vert;
    nc = col + move[dir].horiz;
    if (nr == EXIT_ROW && nc == EXIT_COL)
      found = TRUE;
    else if (!maze[nr][nc] &&
             !mark[nr][nc]) {
      mark[nr][nc] = 1;
      position = (element){row, col, ++dir};
      push(&top, position);
      row = nr;  col = nc;  dir = 0;
    } else ++dir;
} }
```

Code Trace: Backtracking with a Stack



From (2,3), dir 0 (N) is open → move to **(1,3)**; (2,3, dir 1) pushed.

Step 6 / 18

VARIABLES

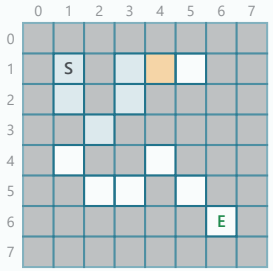
`row = 1` `col = 3` `dir = 0`

`stack = 1,1,5 2,1,4 3,2,2 2,3,1`

OUTPUT

```
while (top > -1 && !found) {
  position = pop(&top);
  row = position.row;  col = position.col;
  dir = position.dir;
  while (dir < 8 && !found) {
    nr = row + move[dir].vert;
    nc = col + move[dir].horiz;
    if (nr == EXIT_ROW && nc == EXIT_COL)
      found = TRUE;
    else if (!maze[nr][nc] &&
             !mark[nr][nc]) {
      mark[nr][nc] = 1;
      position = (element){row, col, ++dir};
      push(&top, position);
      row = nr;  col = nc;  dir = 0;
    } else ++dir;
} }
```

Code Trace: Backtracking with a Stack



From (1,3), dir 2 (E) is open → move to **(1,4)**; (1,3, dir 3) pushed.

Step 7 / 18

VARIABLES

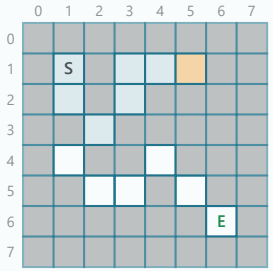
`row = 1` `col = 4` `dir = 0`

`stack = 1,1,5 2,1,4 3,2,2 2,3,1 1,3,3`

OUTPUT

```
1  while (top > -1 && !found) {
2    position = pop(&top);
3    row = position.row;  col = position.col;
4    dir = position.dir;
5    while (dir < 8 && !found) {
6      nr = row + move[dir].vert;
7      nc = col + move[dir].horiz;
8      if (nr == EXIT_ROW && nc == EXIT_COL)
9        found = TRUE;
10     else if (!maze[nr][nc] &&
11              !mark[nr][nc]) {
▶        mark[nr][nc] = 1;
▶        position = (element){row, col, ++dir};
▶        push(&top, position);
▶        row = nr;  col = nc;  dir = 0;
16     } else ++dir;
17 } }
```

Code Trace: Backtracking with a Stack



From (1,4), dir 2 (E) is open → move to **(1,5)**; (1,4, dir 3) pushed.

Step 8 / 18

VARIABLES

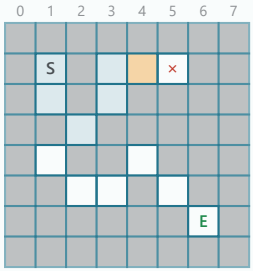
`row = 1` `col = 5` `dir = 0`

`stack = 1,1,5 2,1,4 3,2,2 2,3,1 1,3,3 1,4,3`

OUTPUT

```
1  while (top > -1 && !found) {
2    position = pop(&top);
3    row = position.row;  col = position.col;
4    dir = position.dir;
5    while (dir < 8 && !found) {
6      nr = row + move[dir].vert;
7      nc = col + move[dir].horiz;
8      if (nr == EXIT_ROW && nc == EXIT_COL)
9        found = TRUE;
10     else if (!maze[nr][nc] &&
11              !mark[nr][nc]) {
▶        mark[nr][nc] = 1;
▶        position = (element){row, col, ++dir};
▶        push(&top, position);
▶        row = nr;  col = nc;  dir = 0;
16     } else ++dir;
17 } }
```

Code Trace: Backtracking with a Stack



Back up / pop: return to (1,4) and keep trying from dir 3 (SE).

Step 9 / 18

VARIABLES

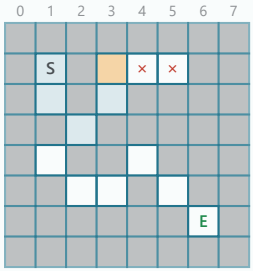
`row = 1` `col = 4` `dir = 3`

`stack = 1,1,5 2,1,4 3,2,2 2,3,1 1,3,3`

OUTPUT

```
1 while (top > -1 && !found) {
    position = pop(&top);
    row = position.row;  col = position.col;
    dir = position.dir;
5   while (dir < 8 && !found) {
6     nr = row + move[dir].vert;
7     nc = col + move[dir].horiz;
8     if (nr == EXIT_ROW && nc == EXIT_COL)
9       found = TRUE;
10    else if (!maze[nr][nc] &&
11             !mark[nr][nc]) {
12      mark[nr][nc] = 1;
13      position = (element){row, col, ++dir};
14      push(&top, position);
15      row = nr;  col = nc;  dir = 0;
16    } else ++dir;
17 } }
```

Code Trace: Backtracking with a Stack



Back up / pop: return to (1,3) and keep trying from dir 3 (SE).

Step 10 / 18

VARIABLES

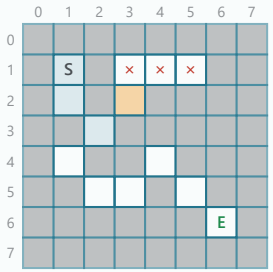
`row = 1` `col = 3` `dir = 3`

`stack = 1,1,5 2,1,4 3,2,2 2,3,1`

OUTPUT

```
while (top > -1 && !found) {
  position = pop(&top);
  row = position.row;  col = position.col;
  dir = position.dir;
  while (dir < 8 && !found) {
    nr = row + move[dir].vert;
    nc = col + move[dir].horiz;
    if (nr == EXIT_ROW && nc == EXIT_COL)
      found = TRUE;
    else if (!maze[nr][nc] &&
             !mark[nr][nc]) {
      mark[nr][nc] = 1;
      position = (element){row, col, ++dir};
      push(&top, position);
      row = nr;  col = nc;  dir = 0;
    } else ++dir;
} }
```

Code Trace: Backtracking with a Stack



Back up / pop: return to (2,3) and keep trying from dir 1 (NE).

Step 11 / 18

VARIABLES

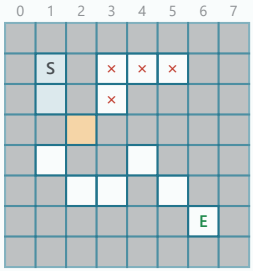
`row = 2` `col = 3` `dir = 1`

`stack = 1,1,5 2,1,4 3,2,2`

OUTPUT

```
1 while (top > -1 && !found) {
▶   position = pop(&top);
▶   row = position.row;  col = position.col;
▶   dir = position.dir;
5   while (dir < 8 && !found) {
6     nr = row + move[dir].vert;
7     nc = col + move[dir].horiz;
8     if (nr == EXIT_ROW && nc == EXIT_COL)
9       found = TRUE;
10    else if (!maze[nr][nc] &&
11             !mark[nr][nc]) {
12      mark[nr][nc] = 1;
13      position = (element){row, col, ++dir};
14      push(&top, position);
15      row = nr;  col = nc;  dir = 0;
16    } else ++dir;
17 } }
```

Code Trace: Backtracking with a Stack



Back up / pop: return to (3,2) and keep trying from dir 2 (E).

Step 12 / 18

VARIABLES

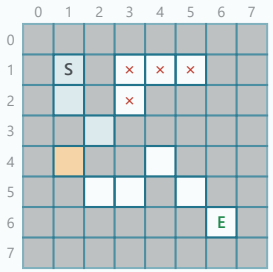
`row = 3` `col = 2` `dir = 2`

`stack = 1,1,5 2,1,4`

OUTPUT

```
while (top > -1 && !found) {
  position = pop(&top);
  row = position.row;  col = position.col;
  dir = position.dir;
  while (dir < 8 && !found) {
    nr = row + move[dir].vert;
    nc = col + move[dir].horiz;
    if (nr == EXIT_ROW && nc == EXIT_COL)
      found = TRUE;
    else if (!maze[nr][nc] &&
             !mark[nr][nc]) {
      mark[nr][nc] = 1;
      position = (element){row, col, ++dir};
      push(&top, position);
      row = nr;  col = nc;  dir = 0;
    } else ++dir;
} }
```

Code Trace: Backtracking with a Stack



From (3,2), dir 5 (SW) is open → move to **(4,1)**; (3,2, dir 6) pushed.

Step 13 / 18

VARIABLES

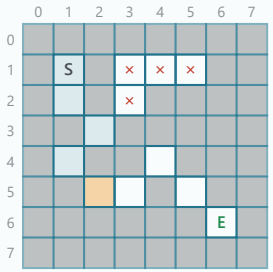
`row = 4` `col = 1` `dir = 0`

`stack = 1,1,5 2,1,4 3,2,6`

OUTPUT

```
while (top > -1 && !found) {
  position = pop(&top);
  row = position.row;  col = position.col;
  dir = position.dir;
  while (dir < 8 && !found) {
    nr = row + move[dir].vert;
    nc = col + move[dir].horiz;
    if (nr == EXIT_ROW && nc == EXIT_COL)
      found = TRUE;
    else if (!maze[nr][nc] &&
             !mark[nr][nc]) {
      mark[nr][nc] = 1;
      position = (element){row, col, ++dir};
      push(&top, position);
      row = nr;  col = nc;  dir = 0;
    } else ++dir;
} }
```

Code Trace: Backtracking with a Stack



From (4,1), dir 3 (SE) is open → move to **(5,2)**; (4,1, dir 4) pushed.

Step 14 / 18

VARIABLES

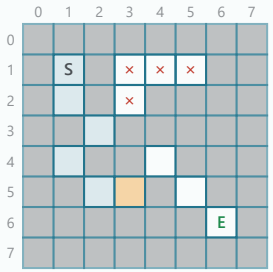
`row = 5` `col = 2` `dir = 0`

`stack = 1,1,5 2,1,4 3,2,6 4,1,4`

OUTPUT

```
while (top > -1 && !found) {
  position = pop(&top);
  row = position.row;  col = position.col;
  dir = position.dir;
  while (dir < 8 && !found) {
    nr = row + move[dir].vert;
    nc = col + move[dir].horiz;
    if (nr == EXIT_ROW && nc == EXIT_COL)
      found = TRUE;
    else if (!maze[nr][nc] &&
             !mark[nr][nc]) {
      mark[nr][nc] = 1;
      position = (element){row, col, ++dir};
      push(&top, position);
      row = nr;  col = nc;  dir = 0;
    } else ++dir;
} }
```

Code Trace: Backtracking with a Stack



From (5,2), dir 2 (E) is open → move to **(5,3)**; (5,2, dir 3) pushed.

Step 15 / 18

VARIABLES

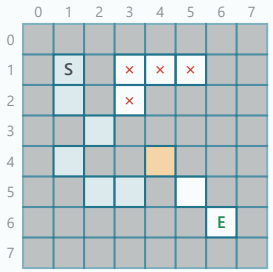
`row = 5` `col = 3` `dir = 0`

`stack = 1,1,5 2,1,4 3,2,6 4,1,4 5,2,3`

OUTPUT

```
1  while (top > -1 && !found) {
2    position = pop(&top);
3    row = position.row;  col = position.col;
4    dir = position.dir;
5    while (dir < 8 && !found) {
6      nr = row + move[dir].vert;
7      nc = col + move[dir].horiz;
8      if (nr == EXIT_ROW && nc == EXIT_COL)
9        found = TRUE;
10     else if (!maze[nr][nc] &&
11              !mark[nr][nc]) {
▶        mark[nr][nc] = 1;
▶        position = (element){row, col, ++dir};
▶        push(&top, position);
▶        row = nr;  col = nc;  dir = 0;
16     } else ++dir;
17 } }
```

Code Trace: Backtracking with a Stack



From (5,3), dir 1 (NE) is open → move to **(4,4)**; (5,3, dir 2) pushed.

Step 16 / 18

VARIABLES

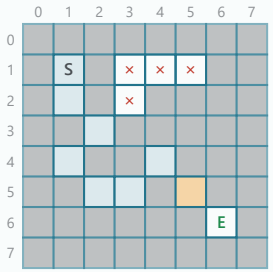
`row = 4` `col = 4` `dir = 0`

`stack = 1,1,5 2,1,4 3,2,6 4,1,4 5,2,3 5,3,2`

OUTPUT

```
while (top > -1 && !found) {
  position = pop(&top);
  row = position.row;  col = position.col;
  dir = position.dir;
  while (dir < 8 && !found) {
    nr = row + move[dir].vert;
    nc = col + move[dir].horiz;
    if (nr == EXIT_ROW && nc == EXIT_COL)
      found = TRUE;
    else if (!maze[nr][nc] &&
             !mark[nr][nc]) {
      mark[nr][nc] = 1;
      position = (element){row, col, ++dir};
      push(&top, position);
      row = nr;  col = nc;  dir = 0;
    } else ++dir;
} }
```

Code Trace: Backtracking with a Stack



From (4,4), dir 3 (SE) is open → move to **(5,5)**; (4,4, dir 4) pushed.

Step 17 / 18

VARIABLES

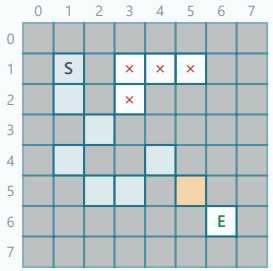
`row = 5` `col = 5` `dir = 0`

```
stac  1,1,5 2,1,4 3,2,6 4,1,4
k =   5,2,3 5,3,2 4,4,4
```

OUTPUT

```
while (top > -1 && !found) {
  position = pop(&top);
  row = position.row;  col = position.col;
  dir = position.dir;
  while (dir < 8 && !found) {
    nr = row + move[dir].vert;
    nc = col + move[dir].horiz;
    if (nr == EXIT_ROW && nc == EXIT_COL)
      found = TRUE;
    else if (!maze[nr][nc] &&
             !mark[nr][nc]) {
      mark[nr][nc] = 1;
      position = (element){row, col, ++dir};
      push(&top, position);
      row = nr;  col = nc;  dir = 0;
    } else ++dir;
} }
```

Code Trace: Backtracking with a Stack



Dir 3 (SE) is the **exit**! Path: the stack + (5,5) + (6,6).

Step 18 / 18

VARIABLES

`found = 1`

```
stac  1,1,5 2,1,4 3,2,6 4,1,4
k =   5,2,3 5,3,2 4,4,4
```

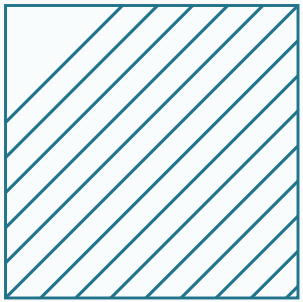
OUTPUT

```
11 21 32 41 52 53 44 55 66
```

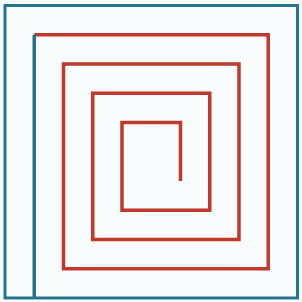
```
1  while (top > -1 && !found) {
2    position = pop(&top);
3    row = position.row;  col = position.col;
4    dir = position.dir;
5    while (dir < 8 && !found) {
6      nr = row + move[dir].vert;
7      nc = col + move[dir].horiz;
8      if (nr == EXIT_ROW && nc == EXIT_COL)
▶        found = TRUE;
10     else if (!maze[nr][nc] &&
11              !mark[nr][nc]) {
12       mark[nr][nc] = 1;
13       position = (element){row, col, ++dir};
14       push(&top, position);
15       row = nr;  col = nc;  dir = 0;
16     } else ++dir;
17 } }
```

Assessment and the Worst Case

- The eight-entry **look-up table** shortened the code and made the job easier.
- The program does a **brute-force** search of the maze with a stack.
- A worst-case analysis needs the **worst-case input**; the order in which directions are scanned decides it.
- **Input 1**: the first path tried is long but correct → only pushes, no pops.
- **Input 2**: the first path tried is long and **wrong** → it is followed to the end and the stack is emptied one by one; the solution is reached only after every cell off the correct path has been visited. This is the worst-case input.



Input 1



Input 2

INSTRUCTOR “Don't turn your nose up at an algorithm when you can't even analyze it!”

Complexity, and a Comparison with a Queue

Time

Each open cell is pushed at most once (`mark`), and at most 8 directions are tried per cell: **O(m·p)** for an m×p maze.

Memory

The stack grows at most to the number of open cells: **O(m·p)**; `mark` is O(m·p) too.

Stack (DFS)

The path found need not be the shortest. In this maze it went into the (2,3)→(1,3)→(1,4)→(1,5) dead end and came back; the path it found has 9 cells.

Queue (BFS)

Expands cells in order of distance, so it guarantees the **shortest** path. Here it finds the same 9-cell path, but in general the DFS path can be longer.

INSTRUCTOR "Only the person who wrote a piece of code can examine it in full detail. Nobody else can analyze it completely."

INSTRUCTOR "If there were 30–40 students instead of 120–130, and I had 4–5 assistants, I could go through the assignments in much more detail."

Evaluating Expressions

Compilers also use stacks to evaluate logical and arithmetic expressions. How does a compiler cope with expressions like these?

Precedence

`x = a / b - c + d * e - a * c`

Value

`((4 / 2) - 2) + (3 * 3) - (4 * 2)` = 1

Parentheses

`a / (b - c) + d * (e - a) * c`

- Reading left to right, we can't apply an operator right away: a higher-precedence operator may follow it.
- The solution has two phases: first convert the expression to **postfix** according to precedence, then **evaluate** the postfix in a single pass. Both use a stack.

INSTRUCTOR "Research the problem first; others have solved it many times. Find the fastest way. Only if it's unsolved does discovery begin."

Operator Precedence in C (Summary)

Precedence	Operators	Associativity
highest	`()` `[]` `→` `.`	left to right
	unary `-` `!` `++` `--` `*` `&`	right to left
	`*` `/` `%`	left to right
	`+` `-`	left to right
	`<` `≤` `>` `≥`	left to right
	`==` `≠`	left to right
	`&&`	left to right
lowest	`||`	left to right

- Precedence comes first; on a tie, **associativity** decides: `a - b - c` = `(a - b) - c`.
- Parentheses override everything.
- Example: `a / b - c + d * e - a * c` = `(((a / b) - c) + (d * e)) - (a * c)`.
- The algorithms in this chapter handle only `+ - * / %` and parentheses (Horowitz–Sahni, Figure 3.12).

EXPRESSION NOTATIONS

Infix, Prefix, Postfix

- **Infix**: the operator sits **between** its operands, `a + b`. Both parentheses and precedence must be taken into account.
- **Prefix** (Polish notation): the operator comes **first**, `+ a b`.
- **Postfix** (reverse Polish): the operator comes **last**, `a b +`. Prefix and postfix **need no parentheses** and are evaluated without thinking about precedence.

#	Prefix	Infix	Postfix
1	`*34`	`3*4`	`34*`
2	`+2*34`	`2+3*4`	`234*+`
3	`-*+abc/da`	`(a+b)*c-d/a`	`ab+c*da/-`
4	`**/a+-bcd-eac`	`(a/(b-c+d))*(e-a)*c`	`abc-d+/ea-*c*`

Converting by Hand: Parenthesize, Move, Erase

```
a / b − c + d * e − a * c
```

The starting expression.

Step 1 / 4

The **order of the operands never changes**; only the operators move. That observation is the basis of the stack algorithm: operands go straight to the output, and operators wait on the stack until their turn comes.

Converting by Hand: Parenthesize, Move, Erase

```
((((a / b) - c) + (d * e)) - (a * c))
```

1. **Fully parenthesize** by precedence and associativity.

Step 2 / 4

The **order of the operands never changes**; only the operators move. That observation is the basis of the stack algorithm: operands go straight to the output, and operators wait on the stack until their turn comes.

Converting by Hand: Parenthesize, Move, Erase

```
((((a b /) c -) (d e *) +) (a c *) -)
```

2. Move each operator to the **right** parenthesis that closes it.

Step 3 / 4

The **order of the operands never changes**; only the operators move. That observation is the basis of the stack algorithm: operands go straight to the output, and operators wait on the stack until their turn comes.

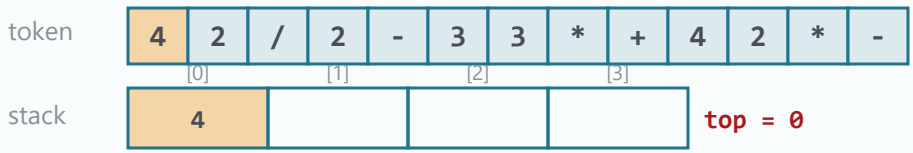
Converting by Hand: Parenthesize, Move, Erase

```
a b / c − d e * + a c * −
```

3. Erase the parentheses: **postfix**. (For prefix, move each operator to its **left** parenthesis: `− + − / a b c * d e * a c`.)

The **order of the operands never changes**; only the operators move. That observation is the basis of the stack algorithm: operands go straight to the output, and operators wait on the stack until their turn comes.

Postfix → Infix: `42/2-33*+42*-`

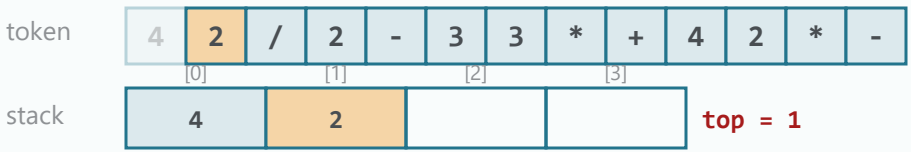


4 operand → push.

Step 1 / 13

After the `*`, cell [1] is not empty: it holds `(3*3)`, then `(4*2)` (top = 1).

Postfix → Infix: `42/2-33*+42*-`

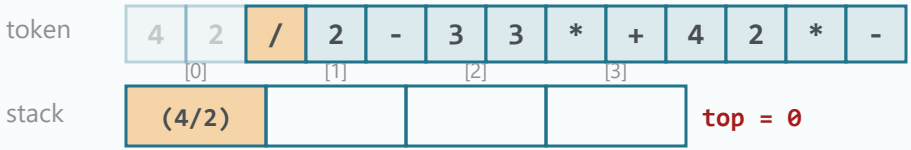


2 operand → push.

Step 2 / 13

After the `*`, cell [1] is not empty: it holds `(3*3)`, then `(4*2)` (top = 1).

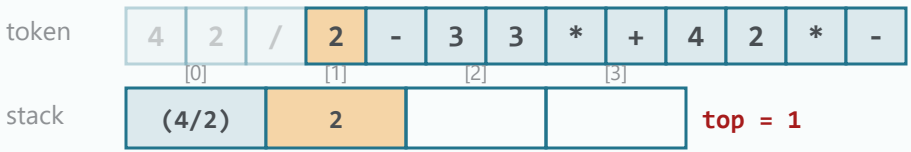
Postfix → Infix: `42/2-33*+42*-`



/ operator → pop two (4, 2) and push back `(4/2)`.

After the `*`, cell [1] is not empty: it holds `(3*3)`, then `(4*2)` (top = 1).

Postfix → Infix: `42/2−33*+42*−`

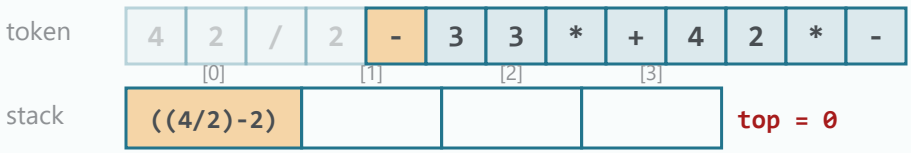


2 operand → push.

Step 4 / 13

After the `*`, cell [1] is not empty: it holds `(3*3)`, then `(4*2)` (top = 1).

Postfix → Infix: `42/2−33*+42*−`

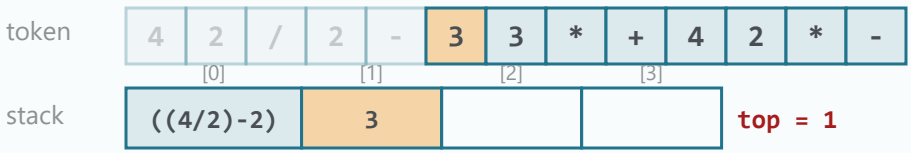


- operator → pop two ((4/2), 2) and push back `((4/2)−2)`.

Step 5 / 13

After the `*`, cell [1] is not empty: it holds `(3*3)`, then `(4*2)` (top = 1).

Postfix → Infix: `42/2−33*+42*−`

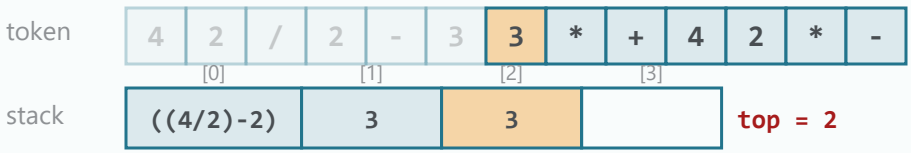


3 operand → push.

Step 6 / 13

After the `*`, cell [1] is not empty: it holds `(3*3)`, then `(4*2)` (top = 1).

Postfix → Infix: `42/2-33*+42*-`

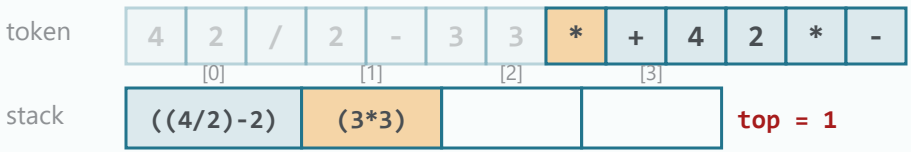


3 operand → push.

Step 7 / 13

After the `*`, cell [1] is not empty: it holds `(3*3)`, then `(4*2)` (top = 1).

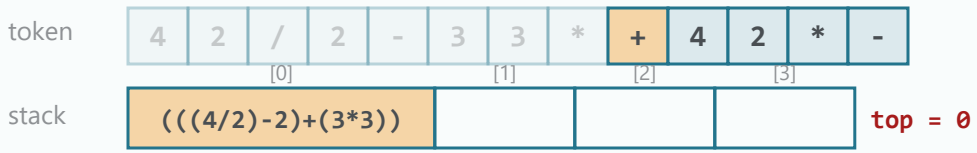
Postfix → Infix: `42/2-33*+42*-`



* operator → pop two (3, 3) and push back `(3*3)`.

After the `*`, cell [1] is not empty: it holds `(3*3)`, then `(4*2)` (top = 1).

Postfix → Infix: `42/2-33*+42*-`

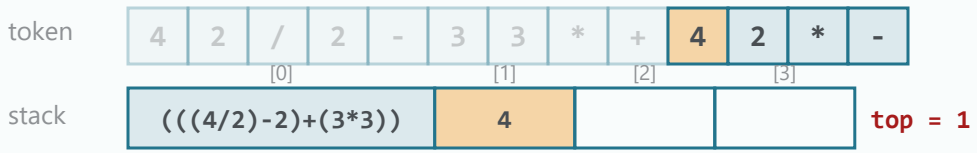


+ operator → pop two (((4/2)-2), (3*3)) and push back `(((4/2)-2)+(3*3))`.

Step 9 / 13

After the `*`, cell [1] is not empty: it holds `(3*3)`, then `(4*2)` (top = 1).

Postfix → Infix: `42/2−33*+42*−`

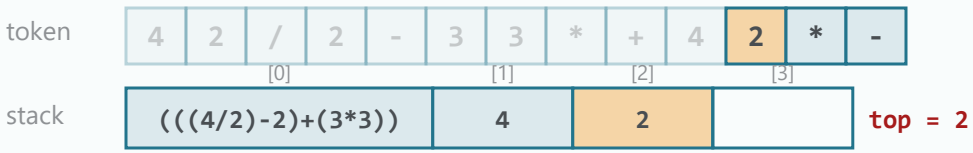


4 operand → push.

Step 10 / 13

After the `*`, cell [1] is not empty: it holds `(3*3)`, then `(4*2)` (top = 1).

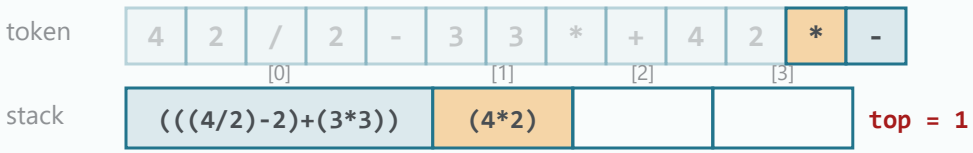
Postfix → Infix: `42/2-33*+42*-`



2 operand → push.

After the `*`, cell [1] is not empty: it holds `(3*3)`, then `(4*2)` (top = 1).

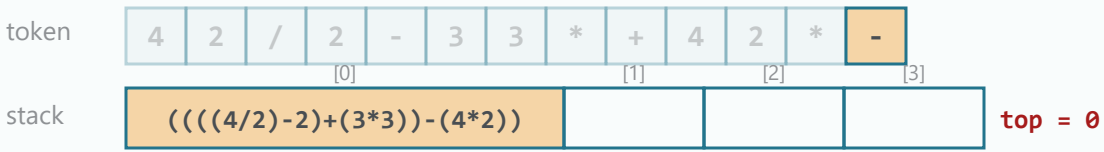
Postfix → Infix: `42/2-33*+42*-`



* operator → pop two (4, 2) and push back `(4*2)`.

After the `*`, cell [1] is not empty: it holds `(3*3)`, then `(4*2)` (top = 1).

Postfix → Infix: `42/2-33*+42*-`



- operator → pop two ((((4/2)-2)+(3*3)), (4*2)) and push back `((((4/2)-2)+(3*3))-(4*2))`. Result: `((((4/2)-2)+(3*3))-(4*2))`.

Step 13 / 13

After the `*`, cell [1] is not empty: it holds `(3*3)`, then `(4*2)` (top = 1).

Tokens: `get_token`

```
typedef enum { lparen, rparen, plus, minus,
    times, divide, mod, eos, operand } precedence;

precedence get_token(const char *expr, char *symbol, int *n) {
    *symbol = expr[(*n)++];
    switch (*symbol) {
        case '(':  return lparen;
        case ')':  return rparen;
        case '+':  return plus;
        case '-':  return minus;
        case '/':  return divide;
        case '*':  return times;
        case '%':  return mod;
        case '\0': return eos;
        default:   return operand;
    }
}
```

- The expression is read one character at a time; each character is a **token**.
- Operands are assumed to be single digits. To turn a digit into a number: `symbol - '0'`. The ASCII code of '2' is 50 and of '0' is 48: 50 − 48 = 2.
- The expression ends with `'\0'` → `eos`.
- `expr` could have been a global array; here it is passed as a parameter.

INSTRUCTOR "Labeling things with typedef enum { ... }; makes our job easier."

Postfix Evaluation: `eval` (Global Version)

```
float eval(void) {
    char symbol;  precedence token;
    float op1, op2;
    int n = 0, top = -1;
    token = get_token(expr, &symbol, &n);
    while (token != eos) {
        if (token == operand)
            push(&top, symbol - '0');
        else {
            op2 = pop(&top);   /* right operand first */
            op1 = pop(&top);
            push(&top, apply(token, op1, op2));
        }
        token = get_token(expr, &symbol, &n);
    }
    return pop(&top);   /* the one left is the result */
}
```

- An operand is pushed; an operator pops two operands, applies itself, and pushes the result.
- The first one popped is **op2** (the right operand): `6 2 -` means 6 − 2, not 2 − 6.
- At the end exactly one item must remain on the stack; otherwise the expression is malformed.
- `apply` picks among the five operators with a `switch`; `case mod: (int)op1 % (int)op2`, since `%` doesn't work on floats.

From Globals to Parameters

```
/* global version: external dependencies */
char expr[] = "42/2-33*+42*-";
float stack[MAX_SIZE];
float eval(void);
```

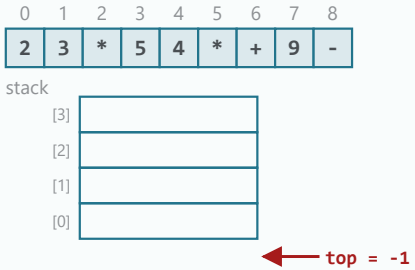
The expression and stack are global: you can't evaluate two expressions at once or reuse the routine in another program.

```
/* component: all parameters/locals */
float eval(const char *expr);
/* the stack is a local array in eval */
/* in main: */
float a = eval("23*54*+9-");
float b = eval("42/2-33*+42*-");
```

The same routine works anywhere, on any expression, with no side effects. The code traces use this version.

INSTRUCTOR "Write functions as if you were writing components. Don't create outside dependencies with globals, defines, and the like! That's not how components are written! Tomorrow, working in a team at some company, are you going to tell your project manager: I'm a very special programmer, I have my own style, put my variables above main, that's how I work, take me as I am or else...?"

Code Trace: `eval("23*54*+9-")`



Expression **23*54*+9-**; the stack is empty.

Step 1 / 11

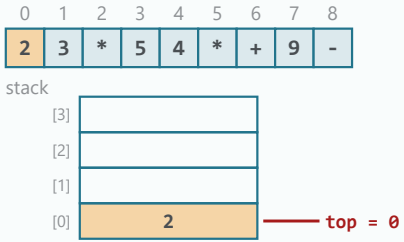
VARIABLES

`expr = "23*54*+9-"` `top = -1`

OUTPUT

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: `eval("23*54*+9-")`



'2' operand: push **'2' − '0' = 2**.

Step 2 / 11

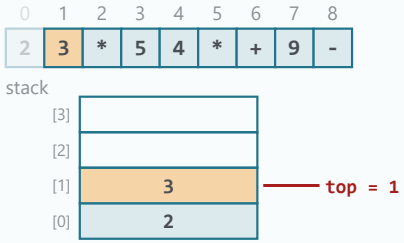
VARIABLES

`n = 0` `c = '2'` `top = 0`

OUTPUT

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: `eval("23*54*+9-")`



'3' operand: push **'3' − '0' = 3**.

Step 3 / 11

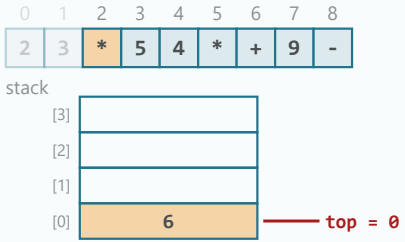
VARIABLES

`n = 1` `c = '3'` `top = 1`

OUTPUT

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: `eval("23*54*+9-")`



'*' operator: op2 = 3, op1 = 2 → push **2 * 3 = 6**.

Step 4 / 11

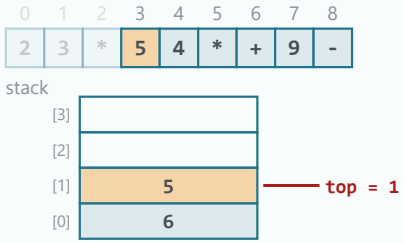
VARIABLES

`n = 2` `c = '*'` `op1 = 2` `op2 = 3` `top = 0`

OUTPUT

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: `eval("23*54*+9-")`



'5' operand: push **'5' − '0' = 5**.

Step 5 / 11

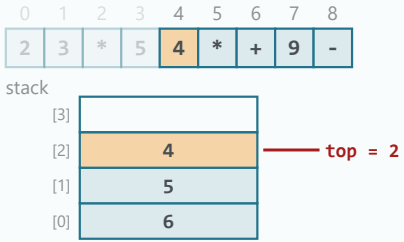
VARIABLES

`n = 3` `c = '5'` `top = 1`

OUTPUT

```
1  float eval(const char *expr) {
2    float y[MAX_SIZE], op1, op2;  int top = -1;
3    for (int n = 0; expr[n] != '\0'; n++) {
4      char c = expr[n];
▶      if (isdigit(c)) y[++top] = c - '0';
6      else {
7        op2 = y[top--];  op1 = y[top--];
8        switch (c) {
9          case '+': y[++top] = op1 + op2; break;
10         case '-': y[++top] = op1 - op2; break;
11         case '*': y[++top] = op1 * op2; break;
12         case '/': y[++top] = op1 / op2; break;
13       }
14     }
15   }
16   return y[top];
17 }
```

Code Trace: `eval("23*54*+9-")`



'4' operand: push **'4' − '0' = 4**.

Step 6 / 11

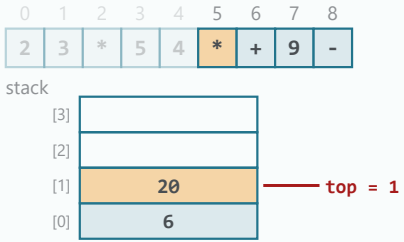
VARIABLES

`n = 4` `c = '4'` `top = 2`

OUTPUT

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: `eval("23*54*+9-")`



'*' operator: op2 = 4, op1 = 5 → push **5 * 4 = 20**.

Step 7 / 11

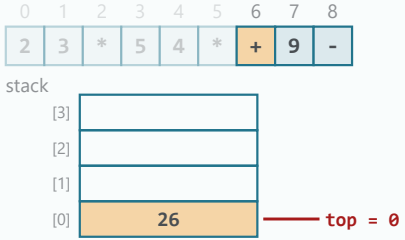
VARIABLES

`n = 5` `c = '*'` `op1 = 5` `op2 = 4` `top = 1`

OUTPUT

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: `eval("23*54*+9-")`



'+' operator: op2 = 20, op1 = 6 → push **6 + 20 = 26**.

Step 8 / 11

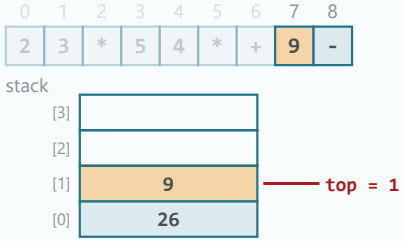
VARIABLES

`n = 6` `c = '+'` `op1 = 6` `op2 = 20` `top = 0`

OUTPUT

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: `eval("23*54*+9-")`



'9' operand: push **'9' − '0' = 9**.

Step 9 / 11

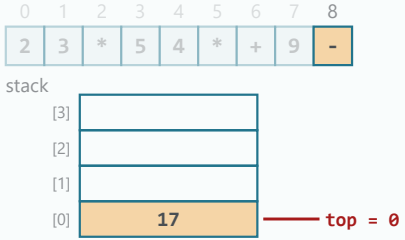
VARIABLES

`n = 7` `c = '9'` `top = 1`

OUTPUT

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: `eval("23*54*+9-")`



'-' operator: op2 = 9, op1 = 26 → push **26 - 9 = 17**.

Step 10 / 11

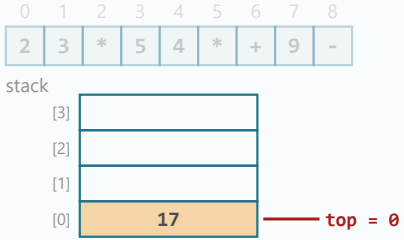
VARIABLES

`n = 8` `c = '-'` `op1 = 26` `op2 = 9` `top = 0`

OUTPUT

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: `eval("23*54*+9-")`



'\0': the loop ends and the one item on the stack, **17**, is returned. Each token is handled once → **O(n)**.

Step 11 / 11

VARIABLES

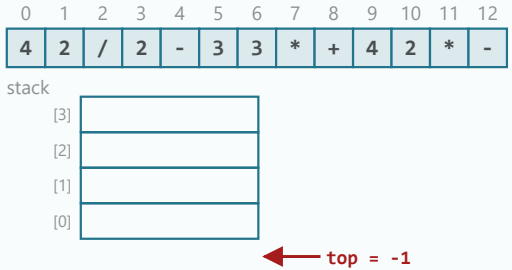
`top = 0` `result = 17`

OUTPUT

```
17
```

```
1  float eval(const char *expr) {
2    float y[MAX_SIZE], op1, op2;  int top = -1;
3    for (int n = 0; expr[n] != '\0'; n++) {
4      char c = expr[n];
5      if (isdigit(c)) y[++top] = c - '0';
6      else {
7        op2 = y[top--];  op1 = y[top--];
8        switch (c) {
9          case '+': y[++top] = op1 + op2; break;
10         case '-': y[++top] = op1 - op2; break;
11         case '*': y[++top] = op1 * op2; break;
12         case '/': y[++top] = op1 / op2; break;
13       }
14     }
15   }
▶  return y[top];
17 }
```

Code Trace: Another Expression



Expression **42/2-33*+42*-**; the stack is empty.

Step 1 / 15

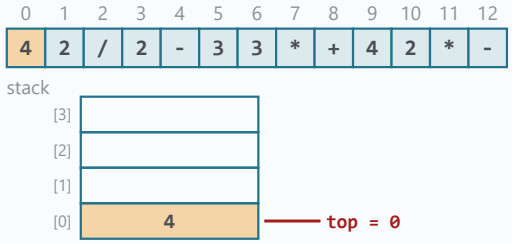
VARIABLES

`expr = "42/2−33*+42*−"` `top = −1`

OUTPUT

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: Another Expression



'4' operand: push **'4' − '0' = 4**.

Step 2 / 15

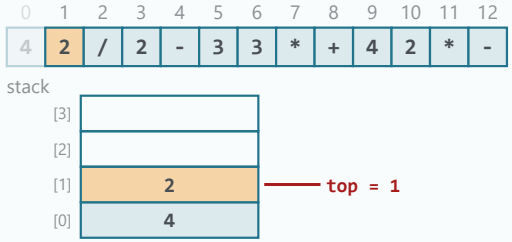
VARIABLES

`n = 0` `c = '4'` `top = 0`

OUTPUT

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: Another Expression



'2' operand: push **'2' − '0' = 2**.

Step 3 / 15

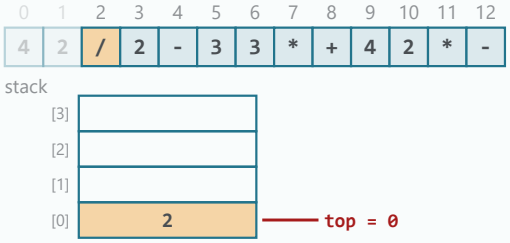
VARIABLES

`n = 1` `c = '2'` `top = 1`

OUTPUT

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: Another Expression



'/' operator: op2 = 2, op1 = 4 → push **4 / 2 = 2**.

Step 4 / 15

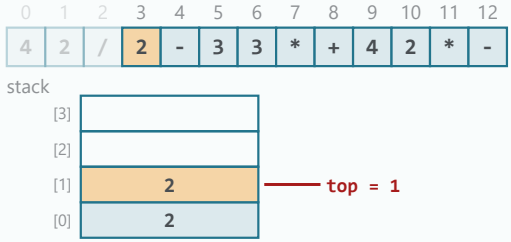
VARIABLES

`n = 2` `c = '/'` `op1 = 4` `op2 = 2` `top = 0`

OUTPUT

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: Another Expression



'2' operand: push **'2' − '0' = 2**.

Step 5 / 15

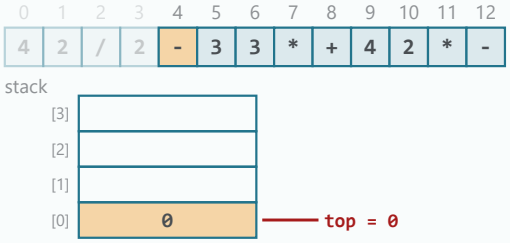
VARIABLES

`n = 3` `c = '2'` `top = 1`

OUTPUT

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: Another Expression



'-' operator: op2 = 2, op1 = 2 → push **2 - 2 = 0**.

Step 6 / 15

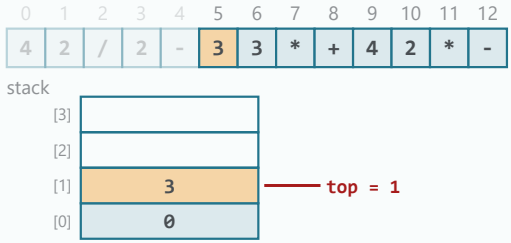
VARIABLES

`n = 4` `c = '-'` `op1 = 2` `op2 = 2` `top = 0`

OUTPUT

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: Another Expression



'3' operand: push **'3' − '0' = 3**.

Step 7 / 15

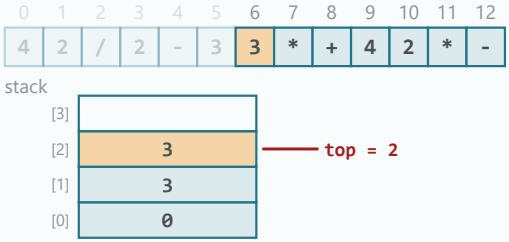
VARIABLES

`n = 5` `c = '3'` `top = 1`

OUTPUT

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: Another Expression



'3' operand: push **'3' − '0' = 3**.

Step 8 / 15

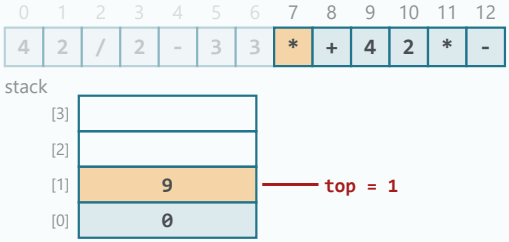
VARIABLES

`n = 6` `c = '3'` `top = 2`

OUTPUT

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: Another Expression



'*' operator: op2 = 3, op1 = 3 → push **3 * 3 = 9**.

Step 9 / 15

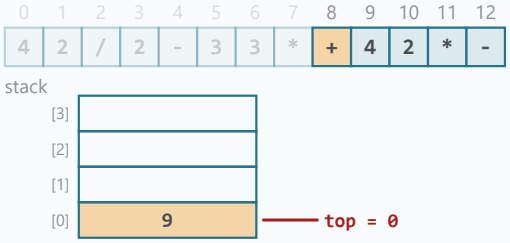
VARIABLES

`n = 7` `c = '*'` `op1 = 3` `op2 = 3` `top = 1`

OUTPUT

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: Another Expression



'+' operator: op2 = 9, op1 = 0 → push **0 + 9 = 9**.

Step 10 / 15

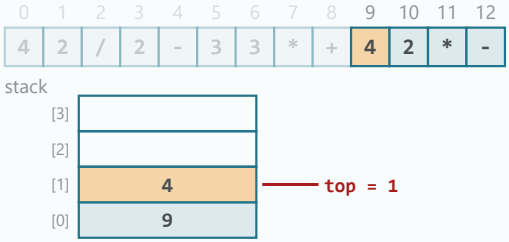
VARIABLES

`n = 8` `c = '+'` `op1 = 0` `op2 = 9` `top = 0`

OUTPUT

```float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: Another Expression



'4' operand: push **'4' − '0' = 4**.

Step 11 / 15

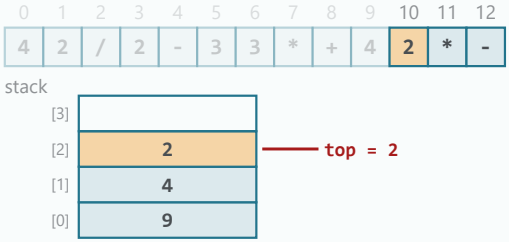
VARIABLES

`n = 9` `c = '4'` `top = 1`

OUTPUT

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: Another Expression



'2' operand: push **'2' − '0' = 2**.

Step 12 / 15

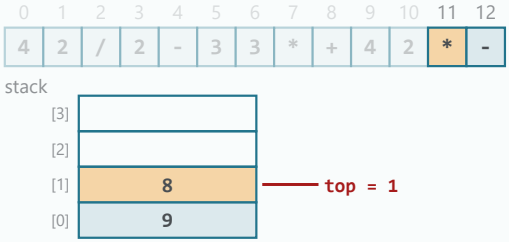
VARIABLES

`n = 10` `c = '2'` `top = 2`

OUTPUT

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: Another Expression



'*' operator: op2 = 2, op1 = 4 → push **4 * 2 = 8**.

Step 13 / 15

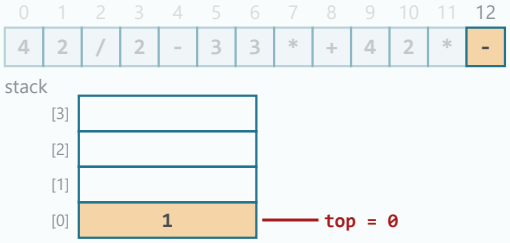
VARIABLES

`n = 11` `c = '*'` `op1 = 4` `op2 = 2` `top = 1`

OUTPUT

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: Another Expression



'-' operator: op2 = 8, op1 = 9 → push **9 - 8 = 1**.

Step 14 / 15

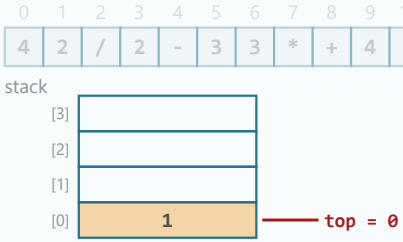
VARIABLES

`n = 12` `c = '-'` `op1 = 9` `op2 = 8` `top = 0`

OUTPUT

```
float eval(const char *expr) {
  float y[MAX_SIZE], op1, op2;  int top = -1;
  for (int n = 0; expr[n] != '\0'; n++) {
    char c = expr[n];
    if (isdigit(c)) y[++top] = c - '0';
    else {
      op2 = y[top--];  op1 = y[top--];
      switch (c) {
        case '+': y[++top] = op1 + op2; break;
        case '-': y[++top] = op1 - op2; break;
        case '*': y[++top] = op1 * op2; break;
        case '/': y[++top] = op1 / op2; break;
      }
    }
  }
  return y[top];
}
```

Code Trace: Another Expression



'\0': the loop ends and the one item on the stack, **1**, is returned. Each token is handled once → **O(n)**.

Step 15 / 15

VARIABLES

`top = 0` `result = 1`

OUTPUT

```
1
```

```
1  float eval(const char *expr) {
2    float y[MAX_SIZE], op1, op2;  int top = -1;
3    for (int n = 0; expr[n] != '\0'; n++) {
4      char c = expr[n];
5      if (isdigit(c)) y[++top] = c - '0';
6      else {
7        op2 = y[top--];  op1 = y[top--];
8        switch (c) {
9          case '+': y[++top] = op1 + op2; break;
10         case '-': y[++top] = op1 - op2; break;
11         case '*': y[++top] = op1 * op2; break;
12         case '/': y[++top] = op1 / op2; break;
13       }
14     }
15   }
▶    return y[top];
17 }
```

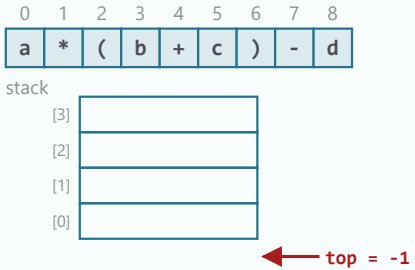
Infix → Postfix: The Algorithm

- **Operand**: written straight to the output.
- **Operator**: while the operator on the stack has precedence **greater than or equal to** the incoming one, pop it to the output; then push the incoming operator (popping on ties = left associativity).
- **(** : always pushed; once inside the stack it acts as the lowest precedence.
- **)**: pop to the output until (is seen; discard the (.
- **At the end**: pop everything left on the stack to the output.

Token	isp (in stack)	icp (incoming)
`(`	0	20
`* / %`	13	13
`+ −`	12	12
`#` (bottom marker)	−1	—

`(` has two precedences: highest when arriving (it goes in without popping anything) and lowest on the stack (any operator can sit on top of it). The `#` at the bottom of the stack stops the loop.

Code Trace: `a*(b+c)-d` → Postfix



Expression **a*(b+c)-d**. A **#** (isp = −1) sits at the bottom of the stack; it is not drawn.

Step 1 / 11

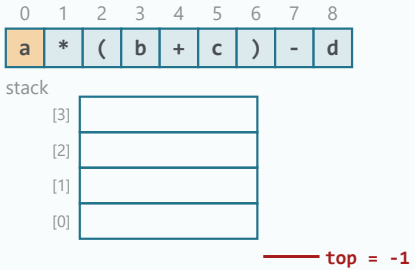
VARIABLES

`top = 0` `out = ""`

OUTPUT

```
1  void postfix(const char *e, char *out) {
▶    char y[MAX] = {'#'};  int top = 0, k = 0;
3    for (; *e; e++) {
4      if (isalnum(*e)) out[k++] = *e;
5      else if (*e == ')') {
6        while (y[top] != '('
7          out[k++] = y[top--];
8        top--;                /* drop '(' */
9      } else {
10       while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
12       y[++top] = *e;
13     }
14   }
15   while (top > 0) out[k++] = y[top--];
16   out[k] = '\0';
17 }
```

Code Trace: `a*(b+c)-d` → Postfix



a operand → output.

Step 2 / 11

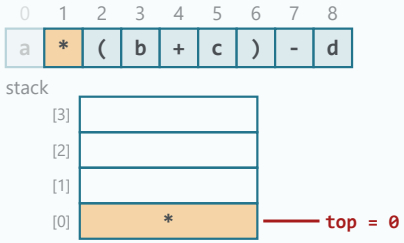
VARIABLES

`*e = 'a'` `top = 0` `out = "a"`

OUTPUT

```
1  void postfix(const char *e, char *out) {
2    char y[MAX] = {'#'};  int top = 0, k = 0;
3    for (; *e; e++) {
▶      if (isalnum(*e)) out[k++] = *e;
5      else if (*e == ')') {
6        while (y[top] != '(')
7          out[k++] = y[top--];
8        top--;                /* drop '(' */
9      } else {
10       while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
12       y[++top] = *e;
13     }
14   }
15   while (top > 0) out[k++] = y[top--];
16   out[k] = '\0';
17 }
```

Code Trace: `a*(b+c)-d` → Postfix



*: isp(#) = -1 < icp(*) = 13 → push *.

Step 3 / 11

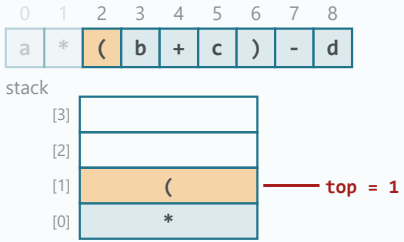
VARIABLES

`*e = '*'` `top = 1` `out = "a"`

OUTPUT

```
1  void postfix(const char *e, char *out) {
2    char y[MAX] = {'#'};  int top = 0, k = 0;
3    for (; *e; e++) {
4      if (isalnum(*e)) out[k++] = *e;
5      else if (*e == ')') {
6        while (y[top] != '(')
7          out[k++] = y[top--];
8        top--;                  /* drop '(' */
9      } else {
▶        while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
▶        y[++top] = *e;
13     }
14   }
15   while (top > 0) out[k++] = y[top--];
16   out[k] = '\0';
17 }
```

Code Trace: `a*(b+c)-d` → Postfix



(: isp(*) = 13 < icp(() = 20 → push (.

Step 4 / 11

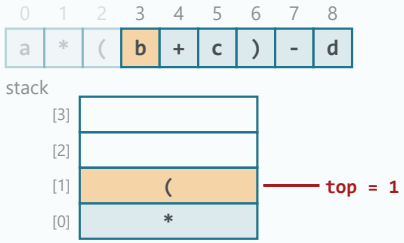
VARIABLES

`*e = '('` `top = 2` `out = "a"`

OUTPUT

```
1  void postfix(const char *e, char *out) {
2    char y[MAX] = {'#'};  int top = 0, k = 0;
3    for (; *e; e++) {
4      if (isalnum(*e)) out[k++] = *e;
5      else if (*e == ')') {
6        while (y[top] != '(')
7          out[k++] = y[top--];
8        top--;                  /* drop '(' */
9      } else {
▶        while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
▶        y[++top] = *e;
13     }
14   }
15   while (top > 0) out[k++] = y[top--];
16   out[k] = '\0';
17 }
```

Code Trace: `a*(b+c)-d` → Postfix



b operand → output.

Step 5 / 11

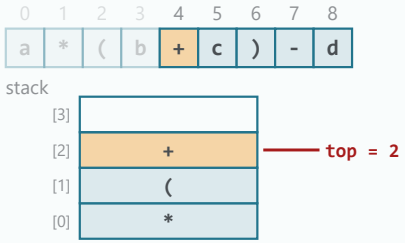
VARIABLES

`*e = 'b'` `top = 2` `out = "ab"`

OUTPUT

```
1  void postfix(const char *e, char *out) {
2    char y[MAX] = {'#'};  int top = 0, k = 0;
3    for (; *e; e++) {
▶      if (isalnum(*e)) out[k++] = *e;
5      else if (*e == ')') {
6        while (y[top] != '(')
7          out[k++] = y[top--];
8        top--;                /* drop '(' */
9      } else {
10       while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
12       y[++top] = *e;
13     }
14   }
15   while (top > 0) out[k++] = y[top--];
16   out[k] = '\0';
17 }
```

Code Trace: `a*(b+c)-d` → Postfix



+: isp(() = 0 < icp(+) = 12 → push +.

Step 6 / 11

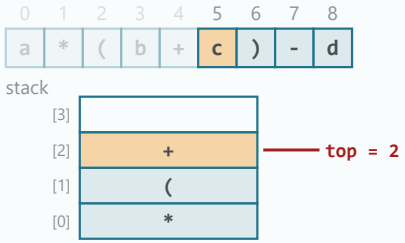
VARIABLES

`*e = '+'` `top = 3` `out = "ab"`

OUTPUT

```
1  void postfix(const char *e, char *out) {
2    char y[MAX] = {'#'};  int top = 0, k = 0;
3    for (; *e; e++) {
4      if (isalnum(*e)) out[k++] = *e;
5      else if (*e == ')') {
6        while (y[top] != '('
7          out[k++] = y[top--];
8        top--;                /* drop '(' */
9      } else {
▶        while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
▶        y[++top] = *e;
13     }
14   }
15   while (top > 0) out[k++] = y[top--];
16   out[k] = '\0';
17 }
```

Code Trace: `a*(b+c)-d` → Postfix



c operand → output.

Step 7 / 11

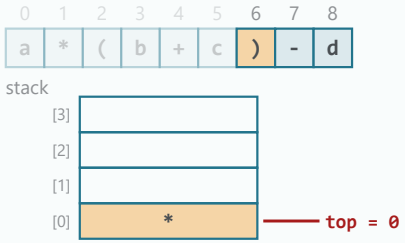
VARIABLES

`*e = 'c'` `top = 3` `out = "abc"`

OUTPUT

```
1  void postfix(const char *e, char *out) {
2    char y[MAX] = {'#'};  int top = 0, k = 0;
3    for (; *e; e++) {
▶      if (isalnum(*e)) out[k++] = *e;
5      else if (*e == ')') {
6        while (y[top] != '(')
7          out[k++] = y[top--];
8        top--;                /* drop '(' */
9      } else {
10       while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
12       y[++top] = *e;
13     }
14   }
15   while (top > 0) out[k++] = y[top--];
16   out[k] = '\0';
17 }
```

Code Trace: `a*(b+c)-d` → Postfix



): pop until (→ **+** to output; (discarded.

Step 8 / 11

VARIABLES

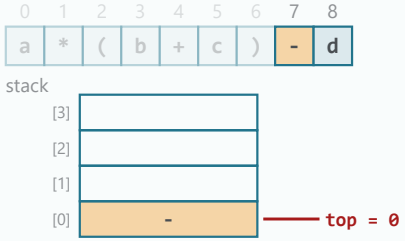
`*e = ')'` `top = 1` `out = "abc+"`

OUTPUT

```
```

```
1  void postfix(const char *e, char *out) {
2    char y[MAX] = {'#'};  int top = 0, k = 0;
3    for (; *e; e++) {
4      if (isalnum(*e)) out[k++] = *e;
5      else if (*e == ')') {
▶        while (y[top] != '('
▶          out[k++] = y[top--];
8        top--;                  /* drop '(' */
9      } else {
10       while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
12       y[++top] = *e;
13     }
14   }
15   while (top > 0) out[k++] = y[top--];
16   out[k] = '\0';
17 }
```

Code Trace: `a*(b+c)-d` → Postfix



-: isp(*) = 13 ≥ icp(-) = 12 → * to output; push -.

Step 9 / 11

VARIABLES

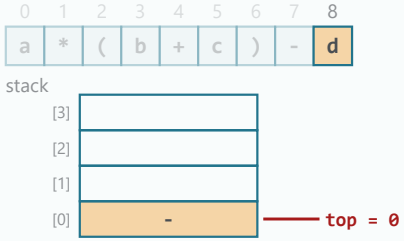
`*e = '-'` `top = 1` `out = "abc+*"`

OUTPUT

```
```

```
1  void postfix(const char *e, char *out) {
2    char y[MAX] = {'#'};  int top = 0, k = 0;
3    for (; *e; e++) {
4      if (isalnum(*e)) out[k++] = *e;
5      else if (*e == ')') {
6        while (y[top] != '(')
7          out[k++] = y[top--];
8        top--;                  /* drop '(' */
9      } else {
▶        while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
▶        y[++top] = *e;
13     }
14   }
15   while (top > 0) out[k++] = y[top--];
16   out[k] = '\0';
17 }
```

Code Trace: `a*(b+c)-d` → Postfix



d operand → output.

Step 10 / 11

VARIABLES

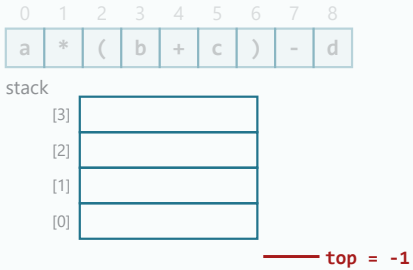
`*e = 'd'` `top = 1`

`out = "abc+*d"`

OUTPUT

```
1  void postfix(const char *e, char *out) {
2    char y[MAX] = {'#'};  int top = 0, k = 0;
3    for (; *e; e++) {
▶      if (isalnum(*e)) out[k++] = *e;
5      else if (*e == ')') {
6        while (y[top] != '(')
7          out[k++] = y[top--];
8        top--;                /* drop '(' */
9      } else {
10       while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
12       y[++top] = *e;
13     }
14   }
15   while (top > 0) out[k++] = y[top--];
16   out[k] = '\0';
17 }
```

Code Trace: `a*(b+c)-d` → Postfix



End of input: the remaining **-** goes to the output. Result: **abc+*d-**. Each token is pushed and popped at most once → **O(n)**.

Step 11 / 11

VARIABLES

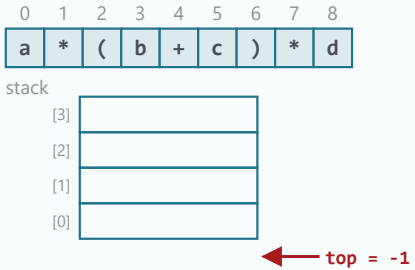
`top = 0` `out = "abc+*d-"`

OUTPUT

```
abc+*d-
```

```
1  void postfix(const char *e, char *out) {
2    char y[MAX] = {'#'};  int top = 0, k = 0;
3    for (; *e; e++) {
4      if (isalnum(*e)) out[k++] = *e;
5      else if (*e == ')') {
6        while (y[top] != '(')
7          out[k++] = y[top--];
8        top--;                /* drop '(' */
9      } else {
10       while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
12       y[++top] = *e;
13     }
14   }
▶    while (top > 0) out[k++] = y[top--];
▶    out[k] = '\0';
17 }
```

Code Trace: The `a*(b+c)*d` Example



Expression **a*(b+c)*d**. A **#** (isp = −1) sits at the bottom of the stack; it is not drawn.

Step 1 / 11

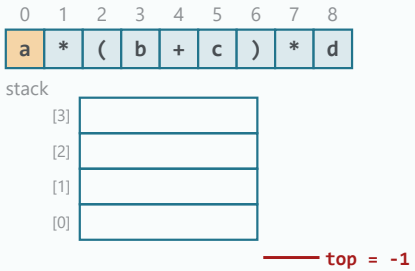
VARIABLES

`top = 0` `out = ""`

OUTPUT

```
1  void postfix(const char *e, char *out) {
▶    char y[MAX] = {'#'};  int top = 0, k = 0;
3    for (; *e; e++) {
4      if (isalnum(*e)) out[k++] = *e;
5      else if (*e == ')') {
6        while (y[top] != '('
7          out[k++] = y[top--];
8        top--;                /* drop '(' */
9      } else {
10       while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
12       y[++top] = *e;
13     }
14   }
15   while (top > 0) out[k++] = y[top--];
16   out[k] = '\0';
17 }
```

Code Trace: The `a*(b+c)*d` Example



a operand → output.

Step 2 / 11

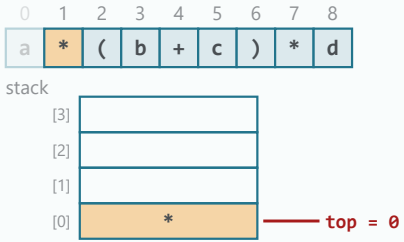
VARIABLES

`*e = 'a'` `top = 0` `out = "a"`

OUTPUT

```
1  void postfix(const char *e, char *out) {
2    char y[MAX] = {'#'};  int top = 0, k = 0;
3    for (; *e; e++) {
▶      if (isalnum(*e)) out[k++] = *e;
5      else if (*e == ')') {
6        while (y[top] != '(')
7          out[k++] = y[top--];
8        top--;                /* drop '(' */
9      } else {
10       while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
12       y[++top] = *e;
13     }
14   }
15   while (top > 0) out[k++] = y[top--];
16   out[k] = '\0';
17 }
```

Code Trace: The `a*(b+c)*d` Example



*: isp(#) = -1 < icp(*) = 13 → push *.

Step 3 / 11

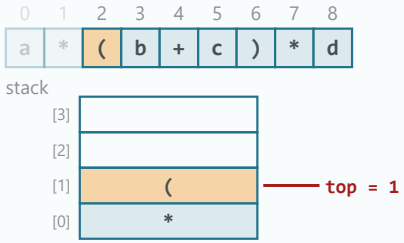
VARIABLES

`*e = '*'` `top = 1` `out = "a"`

OUTPUT

```
1  void postfix(const char *e, char *out) {
2    char y[MAX] = {'#'};  int top = 0, k = 0;
3    for (; *e; e++) {
4      if (isalnum(*e)) out[k++] = *e;
5      else if (*e == ')') {
6        while (y[top] != '(')
7          out[k++] = y[top--];
8        top--;                  /* drop '(' */
9      } else {
▶        while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
▶        y[++top] = *e;
13     }
14   }
15   while (top > 0) out[k++] = y[top--];
16   out[k] = '\0';
17 }
```

Code Trace: The `a*(b+c)*d` Example



(: isp(*) = 13 < icp(() = 20 → push **(**.

Step 4 / 11

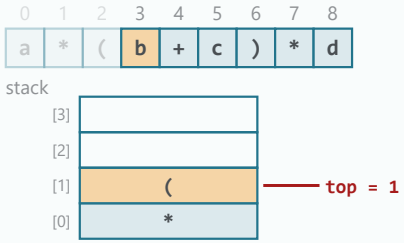
VARIABLES

`*e = '('` `top = 2` `out = "a"`

OUTPUT

```
1  void postfix(const char *e, char *out) {
2    char y[MAX] = {'#'};  int top = 0, k = 0;
3    for (; *e; e++) {
4      if (isalnum(*e)) out[k++] = *e;
5      else if (*e == ')') {
6        while (y[top] != '('
7          out[k++] = y[top--];
8        top--;                /* drop '(' */
9      } else {
▶        while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
▶        y[++top] = *e;
13     }
14   }
15   while (top > 0) out[k++] = y[top--];
16   out[k] = '\0';
17 }
```

Code Trace: The `a*(b+c)*d` Example



b operand → output.

Step 5 / 11

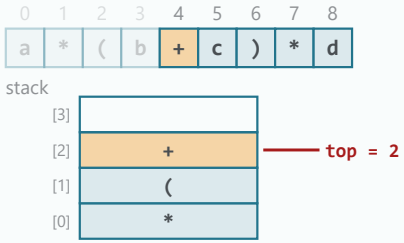
VARIABLES

`*e = 'b'` `top = 2` `out = "ab"`

OUTPUT

```
1  void postfix(const char *e, char *out) {
2    char y[MAX] = {'#'};  int top = 0, k = 0;
3    for (; *e; e++) {
▶      if (isalnum(*e)) out[k++] = *e;
5      else if (*e == ')') {
6        while (y[top] != '(')
7          out[k++] = y[top--];
8        top--;                /* drop '(' */
9      } else {
10       while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
12       y[++top] = *e;
13     }
14   }
15   while (top > 0) out[k++] = y[top--];
16   out[k] = '\0';
17 }
```

Code Trace: The `a*(b+c)*d` Example



+: isp(() = 0 < icp(+) = 12 → push +.

Step 6 / 11

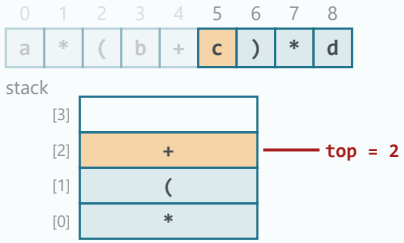
VARIABLES

`*e = '+'` `top = 3` `out = "ab"`

OUTPUT

```
1  void postfix(const char *e, char *out) {
2    char y[MAX] = {'#'};  int top = 0, k = 0;
3    for (; *e; e++) {
4      if (isalnum(*e)) out[k++] = *e;
5      else if (*e == ')') {
6        while (y[top] != '(')
7          out[k++] = y[top--];
8        top--;                /* drop '(' */
9      } else {
▶        while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
▶        y[++top] = *e;
13     }
14   }
15   while (top > 0) out[k++] = y[top--];
16   out[k] = '\0';
17 }
```

Code Trace: The `a*(b+c)*d` Example



c operand → output.

Step 7 / 11

VARIABLES

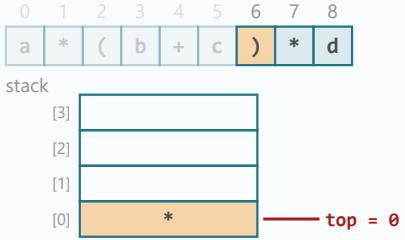
`*e = 'c'` `top = 3` `out = "abc"`

OUTPUT

```
```

```
1  void postfix(const char *e, char *out) {
2    char y[MAX] = {'#'};  int top = 0, k = 0;
3    for (; *e; e++) {
▶      if (isalnum(*e)) out[k++] = *e;
5      else if (*e == ')') {
6        while (y[top] != '(')
7          out[k++] = y[top--];
8        top--;                /* drop '(' */
9      } else {
10       while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
12       y[++top] = *e;
13     }
14   }
15   while (top > 0) out[k++] = y[top--];
16   out[k] = '\0';
17 }
```

Code Trace: The `a*(b+c)*d` Example



): pop until (→ **+** to output; (discarded.

Step 8 / 11

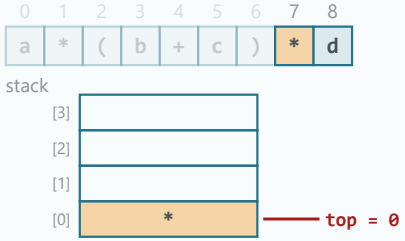
VARIABLES

`*e = ')'` `top = 1` `out = "abc+"`

OUTPUT

```
1  void postfix(const char *e, char *out) {
2    char y[MAX] = {'#'};  int top = 0, k = 0;
3    for (; *e; e++) {
4      if (isalnum(*e)) out[k++] = *e;
5      else if (*e == ')') {
▶        while (y[top] != '(')
▶          out[k++] = y[top--];
8        top--;                  /* drop '(' */
9      } else {
10       while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
12       y[++top] = *e;
13     }
14   }
15   while (top > 0) out[k++] = y[top--];
16   out[k] = '\0';
17 }
```

Code Trace: The `a*(b+c)*d` Example



*: isp(*) = 13 ≥ icp(*) = 13 → * to output; push *.

Step 9 / 11

VARIABLES

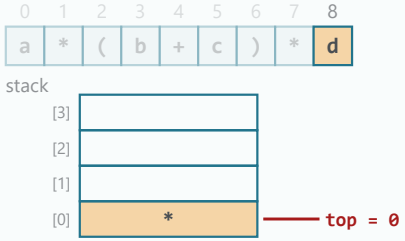
`*e = '*'` `top = 1` `out = "abc+*"`

OUTPUT

```
```

```
1  void postfix(const char *e, char *out) {
2    char y[MAX] = {'#'};  int top = 0, k = 0;
3    for (; *e; e++) {
4      if (isalnum(*e)) out[k++] = *e;
5      else if (*e == ')') {
6        while (y[top] != '(')
7          out[k++] = y[top--];
8        top--;                /* drop '(' */
9      } else {
▶        while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
▶        y[++top] = *e;
13     }
14   }
15   while (top > 0) out[k++] = y[top--];
16   out[k] = '\0';
17 }
```

Code Trace: The `a*(b+c)*d` Example



d operand → output.

Step 10 / 11

VARIABLES

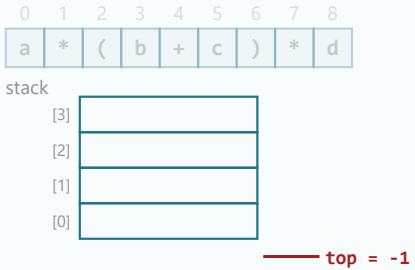
`*e = 'd'` `top = 1`

`out = "abc+*d"`

OUTPUT

```
1  void postfix(const char *e, char *out) {
2    char y[MAX] = {'#'};  int top = 0, k = 0;
3    for (; *e; e++) {
▶      if (isalnum(*e)) out[k++] = *e;
5      else if (*e == ')') {
6        while (y[top] != '(')
7          out[k++] = y[top--];
8        top--;                /* drop '(' */
9      } else {
10       while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
12       y[++top] = *e;
13     }
14   }
15   while (top > 0) out[k++] = y[top--];
16   out[k] = '\0';
17 }
```

Code Trace: The `a*(b+c)*d` Example



End of input: the remaining ***** goes to the output. Result: **abc+*d***. Each token is pushed and popped at most once → **O(n)**.

Step 11 / 11

VARIABLES

`top = 0` `out = "abc+*d*"`

OUTPUT

```
abc+*d*
```

```
1  void postfix(const char *e, char *out) {
2    char y[MAX] = {'#'};  int top = 0, k = 0;
3    for (; *e; e++) {
4      if (isalnum(*e)) out[k++] = *e;
5      else if (*e == ')') {
6        while (y[top] != '(')
7          out[k++] = y[top--];
8        top--;                /* drop '(' */
9      } else {
10       while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
12       y[++top] = *e;
13     }
14   }
▶    while (top > 0) out[k++] = y[top--];
▶    out[k] = '\0';
17 }
```

Trace Table: `a+b*c` → `abc*+`

token	[0]	[1]	top	Output
a			−1	a
+	+		0	a
b	+		0	ab
*	+	*	1	ab
c	+	*	1	abc
eos			−1	abc*+

- When `*` arrives, `+` is on top: isp(+) = 12 < icp(*) = 13 → `*` goes **on top of it**.
- At the end `*` comes out first, then `+`: the higher-precedence operator is applied first.

A common question: “Doesn't the last operand get lost?” No: `postfix` writes operands to the output immediately and empties the stack at the end, so the last operand is printed too.

Infix → Prefix: Reverse, Postfix, Reverse

```
(a+b)*c-d/a
```

One way to get infix → prefix: **reverse** the expression.

Step 1 / 5

A direct way: scan the expression **right to left**; operands are prepended to the output, and operators follow the same stack rule. Both ways are O(n).

Infix → Prefix: Reverse, Postfix, Reverse

```
a/d−c*)b+a(
```

Reversed. Now swap **(and)**.

Step 2 / 5

A direct way: scan the expression **right to left**; operands are prepended to the output, and operators follow the same stack rule. Both ways are O(n).

Infix → Prefix: Reverse, Postfix, Reverse

```
a/d-c*(b+a)
```

Parentheses fixed. Convert this to postfix, but **don't pop** on equal precedence (use >), because reversing the expression also reverses associativity.

A direct way: scan the expression **right to left**; operands are prepended to the output, and operators follow the same stack rule. Both ways are O(n).

Infix → Prefix: Reverse, Postfix, Reverse

```
ad/cba+*-
```

The postfix result.

A direct way: scan the expression **right to left**; operands are prepended to the output, and operators follow the same stack rule. Both ways are O(n).

Infix → Prefix: Reverse, Postfix, Reverse

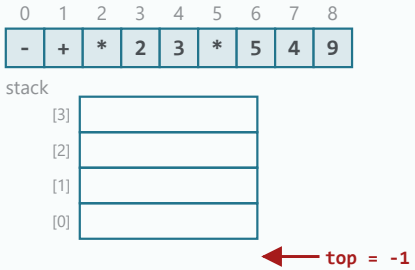
```
-*+abc/da
```

Reverse it again: prefix. Same as row 3 of the table.

Step 5 / 5

A direct way: scan the expression **right to left**; operands are prepended to the output, and operators follow the same stack rule. Both ways are O(n).

Code Trace: Prefix Evaluation (`-+*23*549`)



Prefix **-+*23*549** (infix: 2*3+5*4−9). Scanned **right to left**.

Step 1 / 11

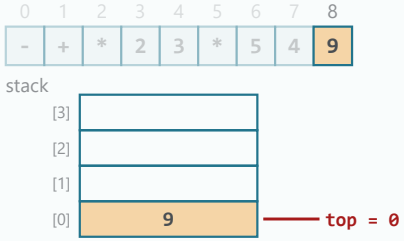
```
1  float prefix_eval(const char *e) {
2    float y[MAX], op1, op2;  int top = -1;
▶    for (int i = strlen(e) - 1; i >= 0; i--) {
4      if (isdigit(e[i])) y[++top] = e[i] - '0';
5      else {
6        op1 = y[top--];  op2 = y[top--];
7        y[++top] = apply(e[i], op1, op2);
8      }
9    }
10   return y[top];
11 }
```

VARIABLES

`i = 8` `top = -1`

OUTPUT

Code Trace: Prefix Evaluation (`-+*23*549`)



'9' operand → push.

Step 2 / 11

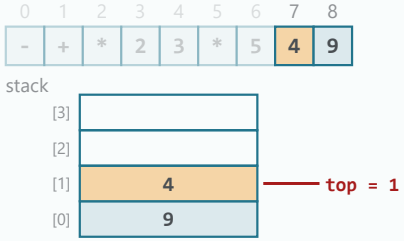
```
1  float prefix_eval(const char *e) {
2    float y[MAX], op1, op2;  int top = -1;
3    for (int i = strlen(e) - 1; i >= 0; i--) {
▶      if (isdigit(e[i])) y[++top] = e[i] - '0';
5      else {
6        op1 = y[top--];  op2 = y[top--];
7        y[++top] = apply(e[i], op1, op2);
8      }
9    }
10   return y[top];
11 }
```

VARIABLES

`i = 8` `e[i] = '9'` `top = 0`

OUTPUT

Code Trace: Prefix Evaluation (`-+*23*549`)



'4' operand → push.

Step 3 / 11

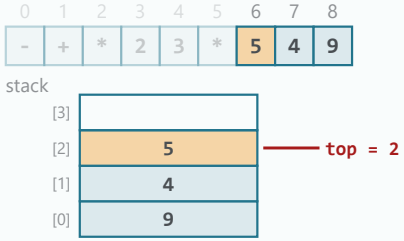
```
1  float prefix_eval(const char *e) {
2    float y[MAX], op1, op2;  int top = -1;
3    for (int i = strlen(e) - 1; i >= 0; i--) {
▶      if (isdigit(e[i])) y[++top] = e[i] - '0';
5      else {
6        op1 = y[top--];  op2 = y[top--];
7        y[++top] = apply(e[i], op1, op2);
8      }
9    }
10   return y[top];
11 }
```

VARIABLES

`i = 7` `e[i] = '4'` `top = 1`

OUTPUT

Code Trace: Prefix Evaluation (`-+*23*549`)



'5' operand → push.

Step 4 / 11

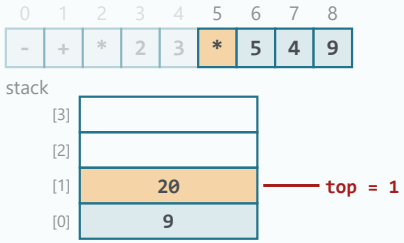
```
1  float prefix_eval(const char *e) {
2    float y[MAX], op1, op2;  int top = -1;
3    for (int i = strlen(e) - 1; i >= 0; i--) {
▶      if (isdigit(e[i])) y[++top] = e[i] - '0';
5      else {
6        op1 = y[top--];  op2 = y[top--];
7        y[++top] = apply(e[i], op1, op2);
8      }
9    }
10   return y[top];
11 }
```

VARIABLES

`i = 6` `e[i] = '5'` `top = 2`

OUTPUT

Code Trace: Prefix Evaluation (`-+*23*549`)



'*': the first popped is **op1** = 5 (the left operand!), op2 = 4 → **5 * 4 = 20**.

Step 5 / 11

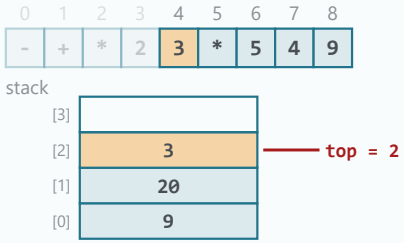
```
1  float prefix_eval(const char *e) {
2    float y[MAX], op1, op2;  int top = -1;
3    for (int i = strlen(e) - 1; i >= 0; i--) {
4      if (isdigit(e[i])) y[++top] = e[i] - '0';
5      else {
▶        op1 = y[top--];  op2 = y[top--];
▶        y[++top] = apply(e[i], op1, op2);
8      }
9    }
10   return y[top];
11 }
```

VARIABLES

`i = 5` `e[i] = '*'` `op1 = 5` `op2 = 4` `top = 1`

OUTPUT

Code Trace: Prefix Evaluation (`-+*23*549`)



'3' operand → push.

Step 6 / 11

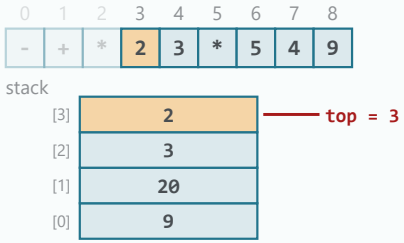
```
1  float prefix_eval(const char *e) {
2    float y[MAX], op1, op2;  int top = -1;
3    for (int i = strlen(e) - 1; i >= 0; i--) {
▶      if (isdigit(e[i])) y[++top] = e[i] - '0';
5      else {
6        op1 = y[top--];  op2 = y[top--];
7        y[++top] = apply(e[i], op1, op2);
8      }
9    }
10   return y[top];
11 }
```

VARIABLES

`i = 4` `e[i] = '3'` `top = 2`

OUTPUT

Code Trace: Prefix Evaluation (`-+*23*549`)



'2' operand → push.

Step 7 / 11

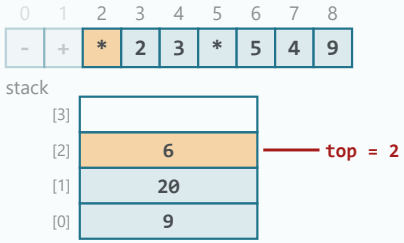
```
1  float prefix_eval(const char *e) {
2    float y[MAX], op1, op2;  int top = -1;
3    for (int i = strlen(e) - 1; i >= 0; i--) {
▶      if (isdigit(e[i])) y[++top] = e[i] - '0';
5      else {
6        op1 = y[top--];  op2 = y[top--];
7        y[++top] = apply(e[i], op1, op2);
8      }
9    }
10   return y[top];
11 }
```

VARIABLES

`i = 3` `e[i] = '2'` `top = 3`

OUTPUT

Code Trace: Prefix Evaluation (`-+*23*549`)



'*': the first popped is **op1** = 2 (the left operand!), op2 = 3 → **2 * 3 = 6**.

Step 8 / 11

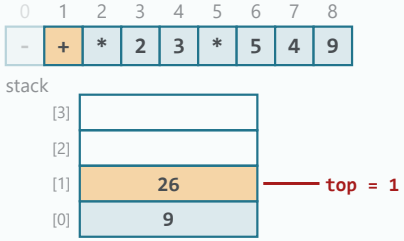
```
1  float prefix_eval(const char *e) {
2    float y[MAX], op1, op2;  int top = -1;
3    for (int i = strlen(e) - 1; i >= 0; i--) {
4      if (isdigit(e[i])) y[++top] = e[i] - '0';
5      else {
▶        op1 = y[top--];  op2 = y[top--];
▶        y[++top] = apply(e[i], op1, op2);
8      }
9    }
10   return y[top];
11 }
```

VARIABLES

`i = 2` `e[i] = '*'` `op1 = 2` `op2 = 3` `top = 2`

OUTPUT

Code Trace: Prefix Evaluation (`-+*23*549`)



'+': the first popped is **op1** = 6 (the left operand!), op2 = 20 → **6 + 20 = 26**.

Step 9 / 11

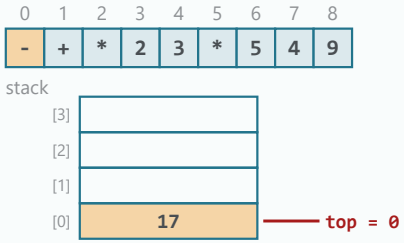
```
1  float prefix_eval(const char *e) {
2    float y[MAX], op1, op2;  int top = -1;
3    for (int i = strlen(e) - 1; i >= 0; i--) {
4      if (isdigit(e[i])) y[++top] = e[i] - '0';
5      else {
▶        op1 = y[top--];  op2 = y[top--];
▶        y[++top] = apply(e[i], op1, op2);
8      }
9    }
10   return y[top];
11 }
```

VARIABLES

`i = 1` `e[i] = '+'` `op1 = 6` `op2 = 20` `top = 1`

OUTPUT

Code Trace: Prefix Evaluation (`-+*23*549`)



'-': the first popped is **op1** = 26 (the left operand!), op2 = 9 → **26 - 9 = 17**.

Step 10 / 11

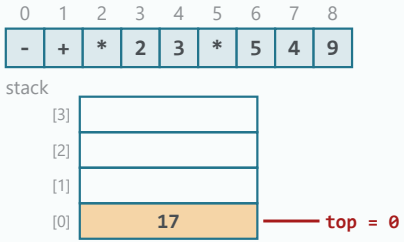
```
1  float prefix_eval(const char *e) {
2    float y[MAX], op1, op2;  int top = -1;
3    for (int i = strlen(e) - 1; i >= 0; i--) {
4      if (isdigit(e[i])) y[++top] = e[i] - '0';
5      else {
▶        op1 = y[top--];  op2 = y[top--];
▶        y[++top] = apply(e[i], op1, op2);
8      }
9    }
10   return y[top];
11 }
```

VARIABLES

`i = 0` `e[i] = '-'` `op1 = 26` `op2 = 9` `top = 0`

OUTPUT

Code Trace: Prefix Evaluation (`-+*23*549`)



Result **17**: the same expression as the postfix example (23*54*+9−). In postfix the first pop was the right operand; in prefix it is the left one.

Step 11 / 11

VARIABLES

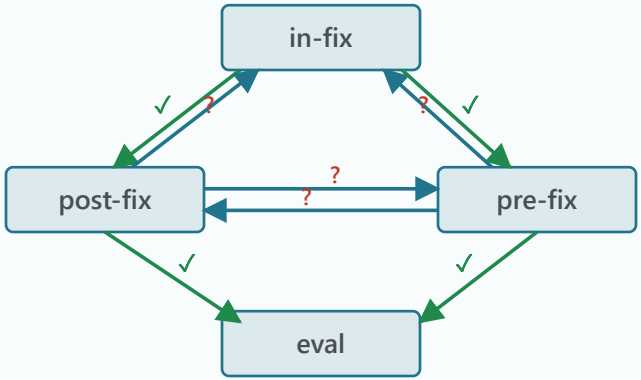
`top = 0` `result = 17`

OUTPUT

```
17
```

```
1  float prefix_eval(const char *e) {
2    float y[MAX], op1, op2;  int top = -1;
3    for (int i = strlen(e) - 1; i >= 0; i--) {
4      if (isdigit(e[i])) y[++top] = e[i] - '0';
5      else {
6        op1 = y[top--];  op2 = y[top--];
7        y[++top] = apply(e[i], op1, op2);
8      }
9    }
▶    return y[top];
11 }
```

Exercise: The Other Conversions

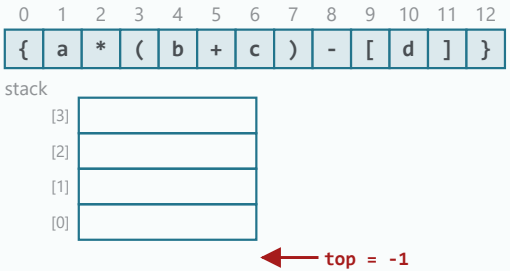


✓: done in this chapter · ?: conversions left to design.

EXERCISE: Design the remaining conversion and evaluation algorithms for expression parsing.

- postfix → infix (done by hand in this chapter; write it in C)
- postfix → prefix and prefix → postfix
- infix → prefix (directly, scanning right to left)
- prefix → infix
- evaluating infix directly (two stacks: operands and operators)

Code Trace: Balanced Brackets `{a*(b+c)-[d]}`



Input **{a*(b+c)-[d]}**.

Step 1 / 15

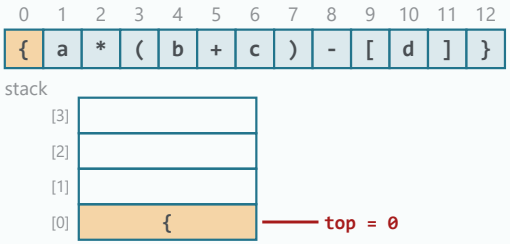
VARIABLES

`top = -1`

OUTPUT

```
bool balanced(const char *e) {
  char y[MAX];  int top = -1;
  for (; *e; e++) {
    if (*e=='(' || *e=='[' || *e=='{')
      y[++top] = *e;
    else if (*e==')' || *e==']' || *e=='}') {
      if (top < 0 || !matches(y[top], *e))
        return false;  /* extra/wrong */
      top--;
    }
  }
  return top == -1;       /* none left open? */
}
```

Code Trace: Balanced Brackets `{a*(b+c)-[d]}`



{ opens → push.

Step 2 / 15

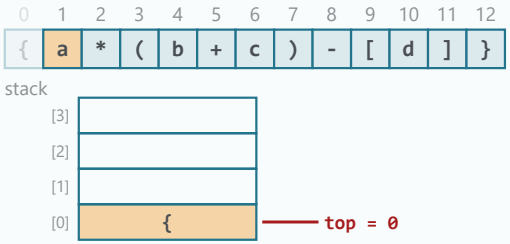
```
1  bool balanced(const char *e) {
2    char y[MAX];  int top = -1;
3    for (; *e; e++) {
4      if (*e=='(' || *e=='[' || *e=='{')
▶        y[++top] = *e;
6      else if (*e==')' || *e==']' || *e=='}') {
7        if (top < 0 || !matches(y[top], *e))
8          return false;  /* extra/wrong */
9        top--;
10     }
11   }
12   return top == -1;        /* none left open? */
13 }
```

VARIABLES

`*e = '{'` `top = 0`

OUTPUT

Code Trace: Balanced Brackets `{a*(b+c)-[d]}`



'a' is not a bracket; skip it.

Step 3 / 15

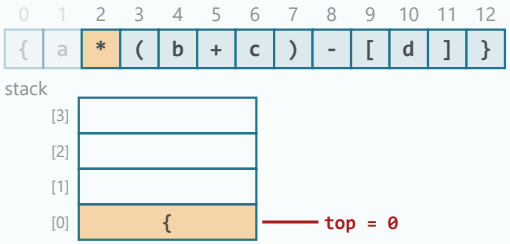
VARIABLES

`*e = 'a'` `top = 0`

OUTPUT

```bool balanced(const char *e) {
  char y[MAX];  int top = -1;
  for (; *e; e++) {
    if (*e=='(' || *e=='[' || *e=='{')
      y[++top] = *e;
    else if (*e==')' || *e==']' || *e=='}') {
      if (top < 0 || !matches(y[top], *e))
        return false;  /* extra/wrong */
      top--;
    }
  }
  return top == -1;       /* none left open? */
}
```

Code Trace: Balanced Brackets `{a*(b+c)-[d]}`



'*' is not a bracket; skip it.

Step 4 / 15

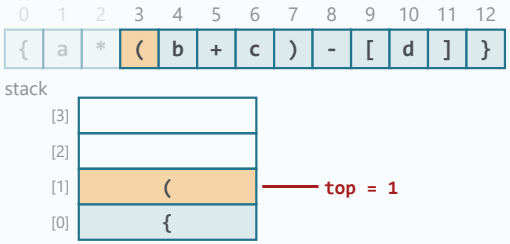
VARIABLES

`*e = '*'` `top = 0`

OUTPUT

```bool balanced(const char *e) {
  char y[MAX];  int top = -1;
  for (; *e; e++) {
    if (*e=='(' || *e=='[' || *e=='{')
      y[++top] = *e;
    else if (*e==')' || *e==']' || *e=='}') {
      if (top < 0 || !matches(y[top], *e))
        return false;  /* extra/wrong */
      top--;
    }
  }
  return top == -1;        /* none left open? */
}
```

Code Trace: Balanced Brackets `{a*(b+c)-[d]}`



(opens → push.

Step 5 / 15

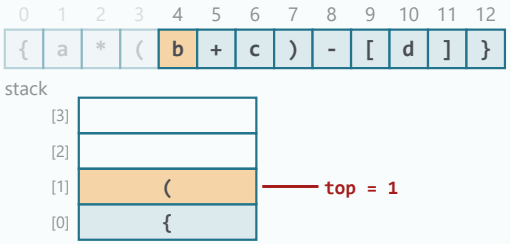
VARIABLES

`*e = '('` `top = 1`

OUTPUT

```bool balanced(const char *e) {
  char y[MAX];  int top = -1;
  for (; *e; e++) {
    if (*e=='(' || *e=='[' || *e=='{')
      y[++top] = *e;
    else if (*e==')' || *e==']' || *e=='}') {
      if (top < 0 || !matches(y[top], *e))
        return false;  /* extra/wrong */
      top--;
    }
  }
  return top == -1;       /* none left open? */
}
```

Code Trace: Balanced Brackets `{a*(b+c)-[d]}`



'b' is not a bracket; skip it.

Step 6 / 15

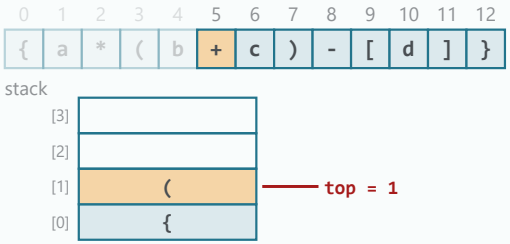
```
1  bool balanced(const char *e) {
2    char y[MAX];  int top = -1;
▶    for (; *e; e++) {
4      if (*e=='(' || *e=='[' || *e=='{')
5        y[++top] = *e;
6      else if (*e==')' || *e==']' || *e=='}') {
7        if (top < 0 || !matches(y[top], *e))
8          return false;  /* extra/wrong */
9        top--;
10     }
11   }
12   return top == -1;        /* none left open? */
13 }
```

VARIABLES

`*e = 'b'` `top = 1`

OUTPUT

Code Trace: Balanced Brackets `{a*(b+c)-[d]}`



'+' is not a bracket; skip it.

Step 7 / 15

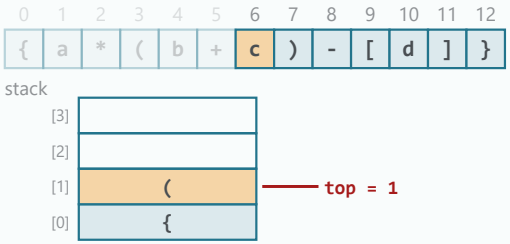
VARIABLES

`*e = '+'` `top = 1`

OUTPUT

```
bool balanced(const char *e) {
  char y[MAX];  int top = -1;
  for (; *e; e++) {
    if (*e=='(' || *e=='[' || *e=='{')
      y[++top] = *e;
    else if (*e==')' || *e==']' || *e=='}') {
      if (top < 0 || !matches(y[top], *e))
        return false;  /* extra/wrong */
      top--;
    }
  }
  return top == -1;       /* none left open? */
}
```

Code Trace: Balanced Brackets `{a*(b+c)-[d]}`



'c' is not a bracket; skip it.

Step 8 / 15

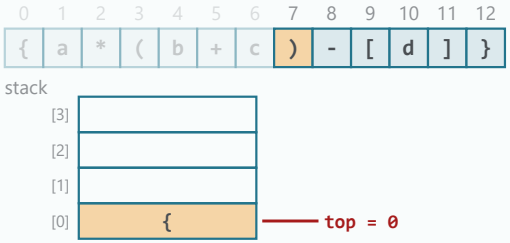
VARIABLES

`*e = 'c'` `top = 1`

OUTPUT

```
bool balanced(const char *e) {
  char y[MAX];  int top = -1;
  for (; *e; e++) {
    if (*e=='(' || *e=='[' || *e=='{')
      y[++top] = *e;
    else if (*e==')' || *e==']' || *e=='}') {
      if (top < 0 || !matches(y[top], *e))
        return false;  /* extra/wrong */
      top--;
    }
  }
  return top == -1;       /* none left open? */
}
```

Code Trace: Balanced Brackets `{a*(b+c)-[d]}`



) closes and matches the (on top → pop.

Step 9 / 15

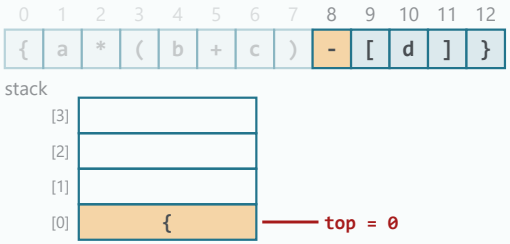
VARIABLES

`*e = ')'` `top = 0`

OUTPUT

```
bool balanced(const char *e) {
  char y[MAX];  int top = -1;
  for (; *e; e++) {
    if (*e=='(' || *e=='[' || *e=='{')
      y[++top] = *e;
    else if (*e==')' || *e==']' || *e=='}') {
      if (top < 0 || !matches(y[top], *e))
        return false;  /* extra/wrong */
      top--;
    }
  }
  return top == -1;        /* none left open? */
}
```

Code Trace: Balanced Brackets `{a*(b+c)-[d]}`



'-' is not a bracket; skip it.

Step 10 / 15

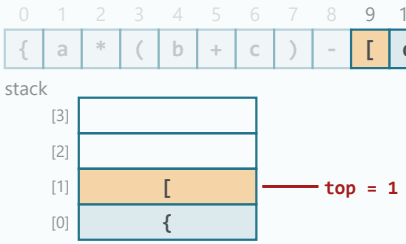
VARIABLES

`*e = '-'` `top = 0`

OUTPUT

```bool balanced(const char *e) {
  char y[MAX];  int top = -1;
  for (; *e; e++) {
    if (*e=='(' || *e=='[' || *e=='{')
      y[++top] = *e;
    else if (*e==')' || *e==']' || *e=='}') {
      if (top < 0 || !matches(y[top], *e))
        return false;  /* extra/wrong */
      top--;
    }
  }
  return top == -1;        /* none left open? */
}
```

Code Trace: Balanced Brackets `{a*(b+c)-[d]}`



[opens → push.

Step 11 / 15

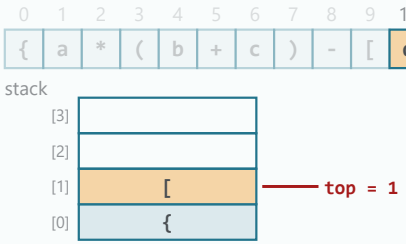
```
1  bool balanced(const char *e) {
2    char y[MAX];  int top = -1;
3    for (; *e; e++) {
4      if (*e=='(' || *e=='[' || *e=='{')
▶        y[++top] = *e;
6      else if (*e==')' || *e==']' || *e=='}') {
7        if (top < 0 || !matches(y[top], *e))
8          return false;  /* extra/wrong */
9        top--;
10     }
11   }
12   return top == -1;        /* none left open? */
13 }
```

VARIABLES

`*e = '['` `top = 1`

OUTPUT

Code Trace: Balanced Brackets `{a*(b+c)-[d]}`



'd' is not a bracket; skip it.

Step 12 / 15

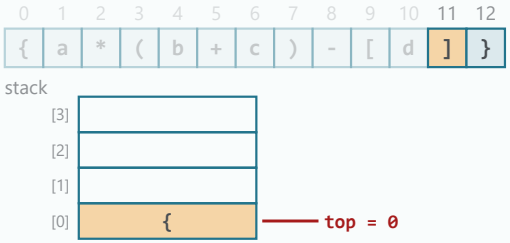
```
1  bool balanced(const char *e) {
2    char y[MAX];  int top = -1;
▶    for (; *e; e++) {
4      if (*e=='(' || *e=='[' || *e=='{')
5        y[++top] = *e;
6      else if (*e==')' || *e==']' || *e=='}') {
7        if (top < 0 || !matches(y[top], *e))
8          return false;  /* extra/wrong */
9        top--;
10     }
11   }
12   return top == -1;        /* none left open? */
13 }
```

VARIABLES

`*e = 'd'` `top = 1`

OUTPUT

Code Trace: Balanced Brackets `{a*(b+c)-[d]}`



] closes and matches the [on top → pop.

Step 13 / 15

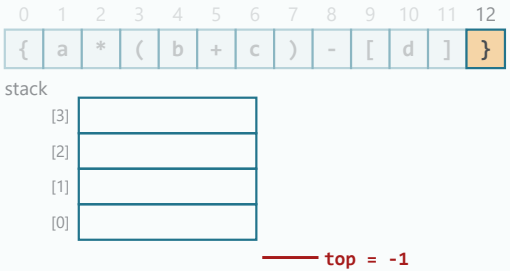
```
1  bool balanced(const char *e) {
2    char y[MAX];  int top = -1;
3    for (; *e; e++) {
4      if (*e=='(' || *e=='[' || *e=='{')
5        y[++top] = *e;
6      else if (*e==')' || *e==']' || *e=='}') {
▶        if (top < 0 || !matches(y[top], *e))
8          return false;  /* extra/wrong */
▶        top--;
10     }
11   }
12   return top == -1;        /* none left open? */
13 }
```

VARIABLES

`*e = ']'` `top = 0`

OUTPUT

Code Trace: Balanced Brackets `{a*(b+c)-[d]}`



} closes and matches the { on top → pop.

Step 14 / 15

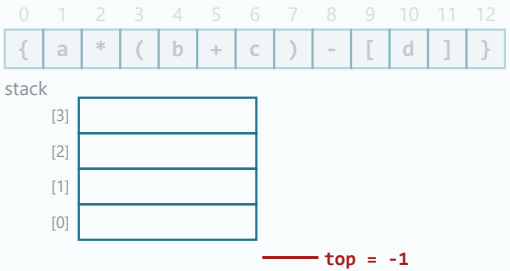
VARIABLES

`*e = '}'` `top = -1`

OUTPUT

```
bool balanced(const char *e) {
  char y[MAX];  int top = -1;
  for (; *e; e++) {
    if (*e=='(' || *e=='[' || *e=='{')
      y[++top] = *e;
    else if (*e==')' || *e==']' || *e=='}') {
      if (top < 0 || !matches(y[top], *e))
        return false;  /* extra/wrong */
      top--;
    }
  }
  return top == -1;        /* none left open? */
}
```

Code Trace: Balanced Brackets `{a*(b+c)-[d]}`



End of input and the stack is **empty** → **true**: balanced.

Step 15 / 15

VARIABLES

`top = -1`

OUTPUT

```
true
```

```
1  bool balanced(const char *e) {
2    char y[MAX];  int top = -1;
3    for (; *e; e++) {
4      if (*e=='(' || *e=='[' || *e=='{')
5        y[++top] = *e;
6      else if (*e==')' || *e==']' || *e=='}') {
7        if (top < 0 || !matches(y[top], *e))
8          return false;  /* extra/wrong */
9        top--;
10     }
11   }
▶    return top == -1;        /* none left open? */
13 }
```

Bracket Matching: Three Kinds of Error

Input	What happens?	Error
`[(a+b]*c)`	`]` arrives with `(` on top	Mismatch
`(a+b))`	The stack is empty when the second `)` arrives	Extra closer
`((a+b)`	`(` is left on the stack at the end	Unclosed opener

- A counter is enough for a single kind of bracket; with **several kinds** you need to know “the most recently opened” → a stack.
- Compilers, HTML/XML validators, and the matching-bracket highlighting in editors all use this algorithm.
- Time O(n); memory O(n) in the worst case (all openers).

OTHER APPLICATIONS

More Applications of Stacks and Queues

Undo / redo

Two stacks: each action goes on the undo stack, each undone action on the redo stack; a browser's back/forward buttons work the same way.

Non-recursive traversal

Inorder traversal becomes a loop with an explicit stack of node addresses: O(n) time, O(h) stack.

Next greater element

The first larger element to the right of each item, in one pass: a decreasing stack, O(n).

Sliding-window maximum

A deque finds the maximum of every window in O(n).

Shortest path with BFS

Use a queue in the maze and the first path found is a shortest one.

Hanoi, n-queens, sudoku

Recursive backtracking, or backtracking with an explicit stack.

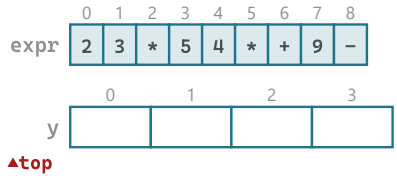
Lab: Postfix Evaluation

23*54*+9-

Input (postfix expression, single-digit numbers) 23*54*+9− Apply

CODE

```
 1 float eval(const char *expr) {
 2   float y[MAX_SIZE], op1, op2; int top = -1;
 3   for (int n = 0; expr[n] != '\0'; n++) {
 4     char c = expr[n];
 5     if (isdigit(c)) y[++top] = c - '0';
 6     else {
 7       op2 = y[top--];  op1 = y[top--];
 8       switch (c) {
 9       case '+': y[++top] = op1 + op2; break;
10       case '-': y[++top] = op1 - op2; break;
11       case '*': y[++top] = op1 * op2; break;
12       case '/': y[++top] = op1 / op2; break;
13       }
14     }
```



n − c − top −1

OUTPUT (none yet)

`eval("23*54*+9−")` : the expression is read left to right. Operands are pushed; for an operator the top two values are popped and the result is pushed back.

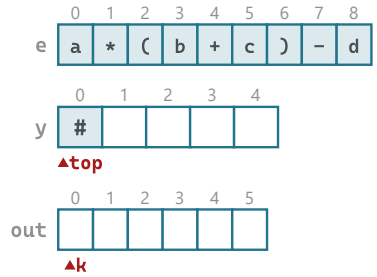
Lab: Infix → Postfix

a*(b+c)-d

Input (infix expression, e.g. a*(b+c)-d) a*(b+c)-d Apply

CODE

```
 1 void postfix(const char *e, char *out) {
 2   char y[MAX] = {'#'};  int top = 0, k = 0;
 3   for (; *e; e++) {
 4     if (isalnum(*e)) out[k++] = *e;
 5     else if (*e == ')') {
 6       while (y[top] != '(')
 7         out[k++] = y[top--];
 8       top--;                /* drop '(' */
 9     } else {
10       while (isp(y[top]) >= icp(*e))
11         out[k++] = y[top--];
12       y[++top] = *e;
13     }
14   }
```



*e '\0' top 0 k 0

OUTPUT (none yet)

postfix("a*(b+c)-d", out) : the bottom of the stack holds '#' (isp = −1); operands go straight to the output, operators wait on the stack according to their precedence.

SUMMARY

Complexity Summary

Algorithm	Structure	Time	Extra memory
Maze (path)	stack	O(m·p)	O(m·p)
Maze shortest path	queue	O(m·p)	O(m·p)
Postfix / prefix evaluation	stack	O(n)	O(n)
Infix → postfix / prefix	stack	O(n)	O(n)
Bracket matching	stack	O(n)	O(n)
Non-recursive inorder	stack	O(n)	O(h)
Level-order traversal	queue	O(n)	O(n)

n: expression length or number of nodes · m×p: maze size · h: tree height.

SUMMARY

Common Mistakes

1. Swapping `op1` and `op2` in postfix: `6 2 -` becomes −4.
2. Using the postfix order in prefix: there the first pop is the **left** operand.
3. Not popping on equal precedence in infix → postfix: `a-b-c` converts wrongly.
4. When `)` arrives, leaving the `(` on the stack or writing it to the output.
5. Forgetting the `mark` in the maze and looping forever.
6. Running off the array because the maze has no border of 1s.

SUMMARY

Exercises

1. Convert `a/(b-c)+d*(e-a)*c` to postfix and prefix.
2. Evaluate the postfix expression `6 2 3 + - 3 8 2 / + *`.
3. Write a C routine that converts postfix to infix.
4. Make `path` mark the path it finds in the maze.
5. Find the shortest path through the maze with a queue.
6. Write non-recursive preorder and postorder traversals.

▶ **Hints / answers**

SUMMARY

Chapter Summary

Backtracking
Every fork in the maze goes on the stack; at a dead end, pop back to the last fork.

Worst case
A long, wrong first path: the whole wrong region gets explored.

Notations
Infix needs precedence; postfix/prefix are evaluated without parentheses or precedence.

Postfix evaluation
Push operands; on an operator pop op2, then op1; push the result. O(n).

Infix → postfix
Pop while isp ≥ icp; (is high when arriving, low on the stack.

General principle
Wherever you see "last opened, first closed," use a stack; wherever you see "in order," use a queue.

INSTRUCTOR "When I lay your assignment over mine and hold them up to the light… (I'd better not see a difference.)"

SUMMARY

References and Next Steps

- E. Horowitz, S. Sahni, S. Anderson-Freed, *Fundamentals of Data Structures in C*, 2nd ed., Sections 3.5 (mazes) and 3.6 (evaluation of expressions).
- R. Sedgewick, *Algorithms in C*, Parts 1–4, Sections 4.3 (postfix) and Chapter 5 (recursion and stacks).

← Chapter 7: Queues

Course page

Chapter 9: The List ADT and Generalized Lists

The End

Chapter 8: Stack and Queue Applications

Next chapter: Chapter 9: The List ADT and Generalized Lists · Course page