**MTH 4300: Algorithms, Computers and Programming II**

**Fall 2025**

**Section: STRA**

# Problem Set 2 - Solutions

**Due Date:** September 12, 2025

## Problem 1: Valid Parentheses Checker (10 points)

Given a string containing only the characters `( ) [ ] { }` determine if the input string has valid parentheses.

An input string is valid if:

1. Open brackets are closed by the same type of brackets
2. Open brackets are closed in the correct order
3. Every close bracket has a corresponding open bracket

```
Example 1:

Input: "()"
Output: true

Example 2:

Input: "()[]{}"
Output: true

Example 3:

Input: "(]"
Output: false
```

Requirements:

- Write a function `bool is_valid(const std::string& s)`
- Use a `std::vector<char>` as a stack to track opening brackets
- Use if-else statements to check bracket types
- Use a for loop to iterate through the string
- Return true if valid, false otherwise

Constraints:

- `1 ≤ s.length ≤ 10⁴`
- `s` consists of parentheses only: `( ) [ ] { }`

### Solution

```cpp
#include <iostream>
#include <string>
#include <vector>

bool is_valid(const std::string& s) {
    std::vector<char> stack;

    for (char c : s) {
        if (c == '(' || c == '[' || c == '{') {
            stack.push_back(c);
        } else {
            if (stack.empty()) {
                return false;
            }

            char top = stack.back();
            stack.pop_back();

            if ((c == ')' && top != '(') ||
                (c == ']' && top != '[') ||
                (c == '}' && top != '{')) {
                return false;
            }
        }
    }

    return stack.empty();
```

```cpp
}

int main() {
    std::string test1 = "()";
    std::string test2 = "()[]{}" ;
    std::string test3 = "(]";

    std::cout << "Test 1: " << test1 << " -> " << (is_valid(test1) ? "true" : "false") << std::endl;
    std::cout << "Test 2: " << test2 << " -> " << (is_valid(test2) ? "true" : "false") << std::endl;
    std::cout << "Test 3: " << test3 << " -> " << (is_valid(test3) ? "true" : "false") << std::endl;

    return 0;
}
```

## Problem 2: Direction Path Validator (10 points)

You are given a string representing a series of movement commands. Determine if following these commands will bring you back to the starting position (origin).

The commands are:

- `'N'` = Move North (y+1)
- `'S'` = Move South (y-1)
- `'E'` = Move East (x+1)
- `'W'` = Move West (x-1)

### Examples

```
Example 1:

Input: "NSEW"
Output: true
Explanation: North->South cancels out, East->West cancels out. Back at origin.

Example 2:

Input: "NNS"
Output: false
Explanation: Moved 2 North, 1 South. Final position is (0,1), not origin.

Example 3:

Input: "NNSSWEEW"
Output: true
Explanation: All movements cancel out perfectly.
```

### Requirements

- Create an enum `Direction` with values: `NORTH`, `SOUTH`, `EAST`, `WEST`
- Write a function `bool returns_to_origin(const std::string& commands)`
- Use a switch statement to convert characters to enum values
- Use the enum values to track x,y position changes
- Use a for loop to process each character in the string
- Return true if final position is (0,0), false otherwise

### Constraints

- $1 \leq$ `commands.length` $\leq 10^4$
- commands consists of only 'N', 'S', 'E', 'W' characters
- Starting position is always (0,0)

### Solution

```cpp
#include <iostream>
#include <string>

enum Direction {
    NORTH,
    SOUTH,
    EAST,
    WEST
};

bool returns_to_origin(const std::string& commands) {
    int x = 0, y = 0;

    for (char c : commands) {
        Direction dir;

        switch (c) {
            case 'N':
                dir = NORTH;
                break;
            case 'S':
                dir = SOUTH;
                break;
            case 'E':
                dir = EAST;
                break;
            case 'W':
```

```
                dir = WEST;
                break;
        }

        switch (dir) {
            case NORTH:
                y++;
                break;
            case SOUTH:
                y--;
                break;
            case EAST:
                x++;
                break;
            case WEST:
                x--;
                break;
        }
    }

    return (x == 0 && y == 0);
}

int main() {
    std::string test1 = "NSEW";
    std::string test2 = "NNS";
    std::string test3 = "NNSSWEEW";

    std::cout << "Test 1: " << test1 << " -> " << (returns_to_origin(test1) ? "true" : "false") << std::endl;
    std::cout << "Test 2: " << test2 << " -> " << (returns_to_origin(test2) ? "true" : "false") << std::endl;
    std::cout << "Test 3: " << test3 << " -> " << (returns_to_origin(test3) ? "true" : "false") << std::endl;

    return 0;
}
```

## Problem 3: Array Element Swapper (10 points)

Given a vector of integers and a series of swap operations, perform the swaps and return the final array state.

Each swap operation is represented by two indices. You must swap the elements at those positions using pointer operations.

### Examples

```
Example 1:

Input: nums = [1,2,3,4,5], swaps = [[0,4], [1,3]]
Output: [5,4,3,2,1]
Explanation:
- Initial: [1,2,3,4,5]
- Swap indices 0,4: [5,2,3,4,1]
- Swap indices 1,3: [5,4,3,2,1]

Example 2:

Input: nums = [10,20,30], swaps = [[0,2], [0,1]]
Output: [20,30,10]
Explanation:
- Initial: [10,20,30]
- Swap indices 0,2: [30,20,10]
- Swap indices 0,1: [20,30,10]
```

### Requirements

- Write a function `void perform_swaps(std::vector<int>& nums, const std::vector<std::vector<int>>& swaps)`
- Create a helper function `void swap_elements(int* a, int* b)` that swaps values using pointers
- Use references to modify the original vector (no copying)
- Use pointers to access and swap array elements
- Use for loops to process each swap operation
- The function should modify the input vector directly

### Constraints

- `1 ≤ nums.length ≤ 100`
- `1 ≤ swaps.length ≤ 100`
- `0 ≤ swaps[i][0], swaps[i][1] < nums.length`
- `swaps[i][0] ≠ swaps[i][1]` (no self-swaps)

### Solution

```cpp
#include <iostream>
#include <vector>

void swap_elements(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

void perform_swaps(std::vector<int>& nums, const std::vector<std::vector<int>>& swaps) {
    for (const std::vector<int>& swap_pair : swaps) {
        int index1 = swap_pair[0];
        int index2 = swap_pair[1];

        swap_elements(&nums[index1], &nums[index2]);
    }
}

int main() {
    // Test 1
    std::vector<int> nums1 = {1, 2, 3, 4, 5};
    std::vector<std::vector<int>> swaps1 = {{0, 4}, {1, 3}};

    std::cout << "Test 1 - Before: ";
    for (int num : nums1) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    perform_swaps(nums1, swaps1);

    std::cout << "Test 1 - After: ";
```

```
    for (int num : nums1) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    // Test 2
    std::vector<int> nums2 = {10, 20, 30};
    std::vector<std::vector<int>> swaps2 = {{0, 2}, {0, 1}};

    std::cout << "Test 2 - Before: ";
    for (int num : nums2) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    perform_swaps(nums2, swaps2);

    std::cout << "Test 2 - After: ";
    for (int num : nums2) {
        std::cout << num << " ";
    }
    std::cout << std::endl;

    return 0;
}
```