12. std::queue and std::stack, problems that may require their usage.

Очередь:

> **Очередь (queue)** — это структура данных, работающая по принципу ***FIFO*** (First In, First Out) Первым пришёл — первым вышел.

Операции:

- `push(value)` — добавляет элемент в конец.
- `pop()` — удаляет элемент из начала.
- `front()` — возвращает элемент в начале.
- `empty()` — проверяет, пуста ли очередь.

Реализация в STL:

В C++ очередь реализована в виде `std::queue` , которая обычно (по дефолту) использует `std::deque` для внутреннего хранения данных:

```
#include <queue>
#include <iostream>
int main() {
        std::queue<int> q;
        q.push(10);
        q.push(20);
        q.push(30);
        while (!q.empty()) {
                 std::cout << q.front() << " ";
                 q.pop();
        }
         return 0;
}
```

```
пусть дан лабиринт, 0 - можно ходить, 1 - нет, 2 - посетили этот элемент.
пусть хотим от верхнего левого угла достичь до правого нижнего. используется волновой
алгоритм. в queue добавляются соседи данного элемента

#include <iostream>
#include <queue>
#include <list>
#include <vector>

bool isLabyrinthPassable(std::vector<std::string> labyrinth)
{
        using Cell = std::pair<int, int>;
        std::queue<Cell, std::list<Cell>> wave;
        wave.push({ 0, 0 });
        labyrinth[0][0] = '2';
        while (!wave.empty())
        {
                auto cell = wave.front();
```

```
                ... ну сделайте сами
        }
}
```

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

bool isLabyrinthPassable(std::vector<std::string> labyrinth) {
    using Cell = std::pair<int, int>;
    std::queue<Cell> wave;

    int rows = labyrinth.size();
    int cols = labyrinth[0].size();

    // Directions for neighbors: right, down, left, up
    std::vector<std::pair<int, int>> directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

    // Push the starting cell
    wave.push({0, 0});
    labyrinth[0][0] = '2'; // Mark as visited

    while (!wave.empty()) {
        auto [x, y] = wave.front(); // Get the front of the queue
        wave.pop();

        // Check if we reached the bottom-right corner
        if (x == rows - 1 && y == cols - 1) {
            return true; // Path exists
        }

        // Explore neighbors
        for (const auto& [dx, dy] : directions) {
            int nx = x + dx;
            int ny = y + dy;

            // Check bounds and if the cell is walkable
            if (nx >= 0 && ny >= 0 && nx < rows && ny < cols && labyrinth[nx][ny] == '0')
{
                wave.push({nx, ny});
                labyrinth[nx][ny] = '2'; // Mark as visited
            }
        }
    }

    return false; // No path found
}

int main() {
    std::vector<std::string> labyrinth{
        "010",
        "000",
    };

    if (isLabyrinthPassable(labyrinth)) {
        std::cout << "The labyrinth is passable!" << std::endl;
```

```
    } else {
        std::cout << "The labyrinth is not passable!" << std::endl;
    }

    return 0;
}
```

- `0` : Walkable cell.
- `1` : Wall (not walkable).
- `2` : Visited cell.
- Start: Top-left `(0, 0)` .
- Goal: Bottom-right `(1, 2)` .

Steps

Step	Current Cell	Queue After Step	Labyrinth State
1	(0, 0)	[(1, 0)]	`210` `000`
2	(1, 0)	[(1, 1)]	`210` `200`
3	(1, 1)	(1, 2)	`210` `220`
5	(1, 2)	Goal Reached	`210` `222`

Стек:

> **Стек (stack)** — это структура данных, работающая по принципу ***LIFO*** (Last In, First Out). Последним пришёл — первым вышел.

Операции:

- `push(value)` — добавляет элемент на вершину стека.
- `pop()` — удаляет элемент с вершины стека.
- `top()` — возвращает элемент с вершины стека.
- `empty()` — проверяет, пуст ли стек.

Реализация в STL:

В C++ стек реализован в виде `std::stack` , который обычно использует `std::deque` или `std::vector` для внутреннего хранения данных. Можно также использовать `std::list` . Но по дефолту стек использует дек.

- Когда вы добавляете элементы в `std::deque` , он не всегда размещает их в одном непрерывном блоке, как `std::vector` . Вместо этого:
  - Новый блок (chunk) выделяется, когда предыдущий блок заполнен.
  - Эти блоки *стараются быть расположены близко друг к другу* в памяти, но это не гарантируется.
- Таким образом, доступ к элементам в `std::deque` может быть чуть медленнее, чем в `std::vector` , из-за дополнительного уровня индирекции (указателей).

Почему используется `std::deque` для `std::stack` :

- `std::deque` предоставляет быструю вставку и удаление элементов с обоих концов.

- `std::stack` требует только операций **push**, **pop**, и **top** (вставка/удаление/доступ к последнему элементу).
- `std::deque` идеально подходит для этих операций, так как они выполняются за O(1) по времени.

```cpp
#include <stack>
#include <iostream>

// Simple stack demonstration
int main() {
    std::stack<int> s;
    s.push(10);
    s.push(20);
    s.push(30);
    while (!s.empty()) {
        std::cout << s.top() << " ";
        s.pop();
    }
    return 0;
}
```

Task: Check for a correct parenthesis sequence

```cpp
#include <iostream>
#include <vector>
#include <forward_list>
#include <algorithm>
#include <numeric>
#include <stack>
#include <list>
#include <deque>
#include <queue>

// Function to check if a parenthesis sequence is correct
bool isCorrectParenthesisSequence(const std::string& parenthesis) {
    int count = 0;
    for (char symbol : parenthesis) {
        if (symbol == '(')
            ++count;

        if (symbol == ')')
            --count;

        if (count < 0)
            return false;
    }
    return count == 0;
}

// Customizable Stack class template
// The second template parameter allows specifying a custom container (default: deque)
template <typename T, typename Container = std::deque<T>>
class Stack {
public:
    void push(const T& val) {
        elements.push_back(val);
    }

    void pop() {
```

```
        elements.pop_back();
    }

    const T& top() const {
        return elements.back();
    }

    bool empty() const {
        return elements.empty();
    }

private:
    Container elements;
};

// Function to check if a string contains multiple types of correct parenthesis sequences
bool isMultipleParenthesisSeq(const std::string& parenthesis) {
    std::stack<char> openingParenthesis;

    auto bracketMatcher = [](char symbol) {
        switch (symbol) {
            case ']': return '[';
            case '}': return '{';
            case ')': return '(';
            default: return '\0';
        }
    };

    for (char symbol : parenthesis) {
        switch (symbol) {
            case '(':
            case '{':
            case '[':
                openingParenthesis.push(symbol);
                break;
            case ')':
            case ']':
            case '}':
                if (openingParenthesis.empty() || openingParenthesis.top() !=
bracketMatcher(symbol))
                    return false;
                openingParenthesis.pop();
                break;
            default:
                break;
        }
    }
    return openingParenthesis.empty();
}
```