Stacks — Scenario Implementations (C++ Group Assignment)

Members:

Student's Name	Student's ID
Thang Saoly	IDTB110257
Phon Sovan Ey	IDTB110218
Noy Sokbolen	IDTB110181
Rith Vichet	IDTB110140

➢ Our Solution:

- We use **ArrayStack<char> solution** to solve the problem of **Balanced Brackets Linter.**
- Because we aren't allowed to use the default stack library, we need to define our own stack, a header file called "stack.hpp".
- In stack.hpp: Simple templated Stack implementation.
  - Template parameter T: element type stored in the stack.
  - Internally uses a dynamic array; top index tracks the current top element.
  - Provides push, pop, peek (inspect top), and empty.
  - pop() and peek() print an error and return default T on underflow; callers must check empty() first.
  - Fixed initial buffer (10); no automatic growth and no destructor/copy-safety in this example.

- In main.cpp:
  - We have checkBalanced function. It verifies bracket pairing in a string.
  - Uses a custom Stack<pair<char,int>> to track opening brackets and their positions.
  - Skips contents of single-quoted character literals (handles escaped characters).
  - Returns "OK" for balanced input or "ERROR pos=<1-based> reason=<...>" for the problem.
  - We Store both the opening bracket and its index so we can report the position of unclosed brackets.
  - When encountering a single-quoted literal, advance past the entire literal so any brackets inside the literal don't affect balancing logic.
  - Positions in error messages are 1-based to match typical user-facing indexing.
  - Escaped chars inside single quotes are handled: a backslash skips the next char.

```cpp
int main(){
    string s = "()[]{}";
    // string s = "([{}])";
    // string s = "([)]";
    // string s = "(((";
    // string s = "()))";
    // string s = "if(a[0] == '{')";

    cout << checkBalanced(s) << endl;

    return 0;
}
```

```cpp
template <typename T>

class Stack
{
private:
    T *arr = new T[10]; // fixed-size buffer (example). Stores stack elements.
    int top = -1;       // index of the top element (-1 means empty)

public:
    void push(T value)
    {
        // overflow guard: simple check for fixed buffer size
        if (top == 9)
        {
            cout << "Stack Overflow" << endl;
            return;
        }
        arr[++top] = value; // increment top then store value
    }

    T pop()
    {
        T temp;
        // underflow guard: no elements to pop
        if (top == -1)
        {
            cout << "Stack Underflow" << endl;
            return temp; // return default-constructed T
        }
        return arr[top--]; // return top element then decrement top
        return temp;       // unreachable but harmless (keeps previous structure)
    }

    T peek()
    {
        T temp;
        if (top == -1)
        {
            cout << "Stack Underflow" << endl;
            return temp; // return default-constructed T if empty
        }
        T val = arr[top]; // read top value
        return arr[top];  // return top element (do not modify top)
        return temp;      // unreachable (left from original code)
    }

    bool empty()
    {
        return top == -1; // true when no elements are present
    }
};
```

```cpp
string checkBalanced(const string &s)
{
    Stack<pair<char, int>> st; // stack of (opening-bracket, index)

    for (int i = 0; i < (int)s.length(); ++i)
    {
        char ch = s[i];

        // If we hit a single-quoted literal, skip everything until the closing quote.
        // This prevents characters like '{' inside literals from being interpreted as code.
        if (ch == '\'')
        {
            ++i; // move past opening quote
            while (i < (int)s.length())
            {
                if (s[i] == '\\')
                { // escaped char inside literal: skip it and the next char
                    ++i;
                    if (i < (int)s.length())
                        ++i;
                    continue; // continue scanning inside the literal
                }
                if (s[i] == '\'')
                    break; // found closing quote of the literal
                ++i;
            }
            continue; // continue main loop after the literal is skipped
        }

        // If an opening bracket, push it with its index.
        if (ch == '(' || ch == '{' || ch == '[')
        {
            st.push({ch, i});
        }
        // If a closing bracket, check matching with top of stack.
        else if (ch == ')' || ch == '}' || ch == ']')
        {
            if (st.empty())
            {
                // No opening bracket for this closing one.
                return "ERROR pos=" + to_string(i + 1) + " reason=extra-closing";
            }
            auto top = st.peek();  // get the top pair<char,int> without removing
            char open = top.first; // the opening bracket char
            int pos = top.second;  // its index (0-based)
            st.pop();              // remove the matched opening
            // If types don't match, it's a mismatch error at this closing bracket.
            if ((ch == ')' && open != '(') ||
                (ch == '}' && open != '{') ||
                (ch == ']' && open != '['))
            {
                return "ERROR pos=" + to_string(i + 1) + " reason=mismatch";
            }
        }
    }

    // After scanning: any remaining opening bracket is unclosed.
    if (!st.empty())
    {
        auto top = st.peek();
        // Report position as 1-based index of the unmatched opening bracket.
        return "ERROR pos=" + to_string(top.second + 1) + " reason=unclosed";
    }
    return "OK";
}
```

Screenshot of the Result in Terminal: