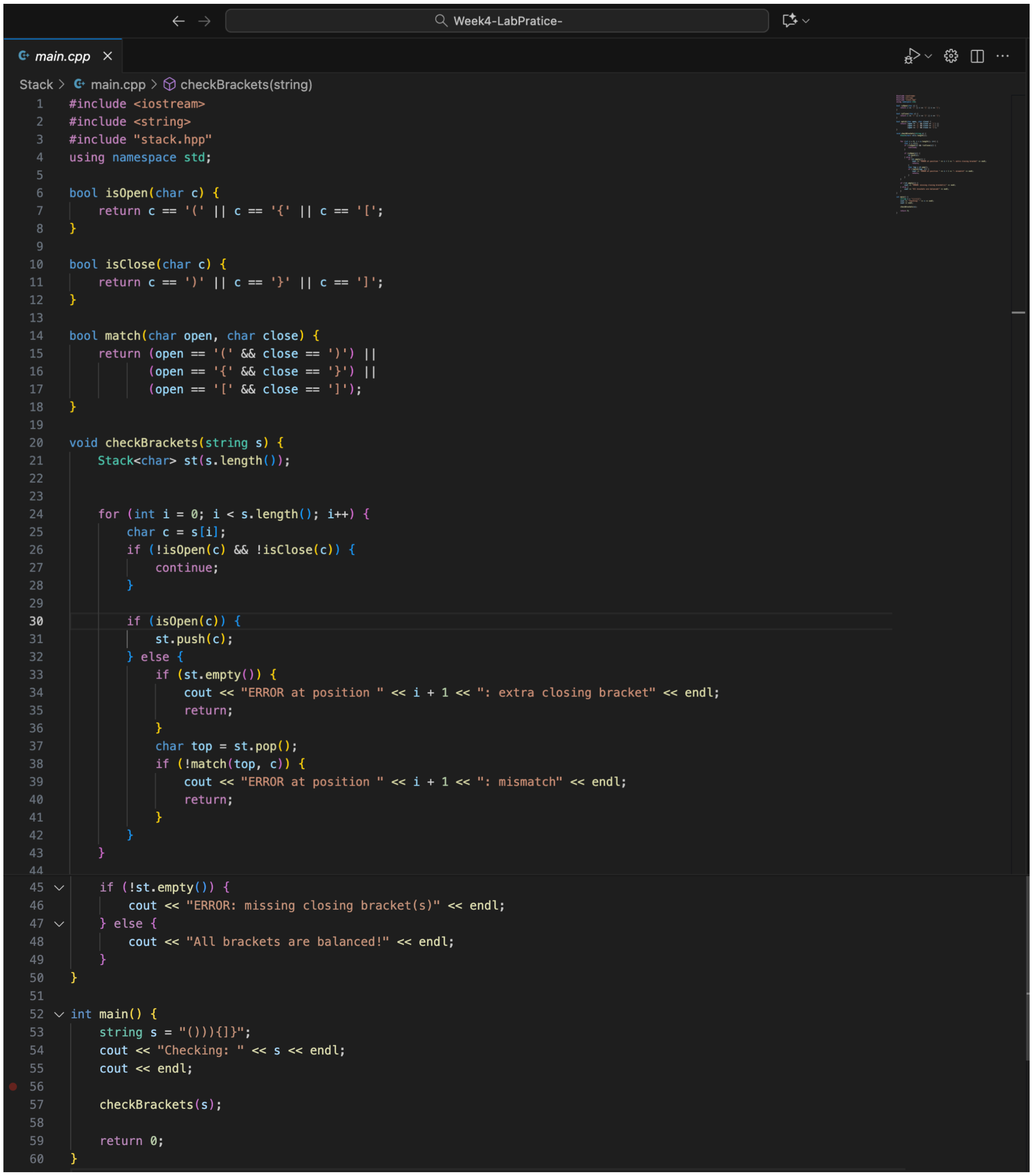
Week4 - Lab Practice : Stacks

Group 6 :

- Chea Kuyhong (IDTB110172)
- Chhuong Sophakvatey (IDTB110187)
- Ean Mana (IDTB110367)
- Ly Sokunnita (IDTB110369)
- Nget Sokunkanha (IDTB110054)

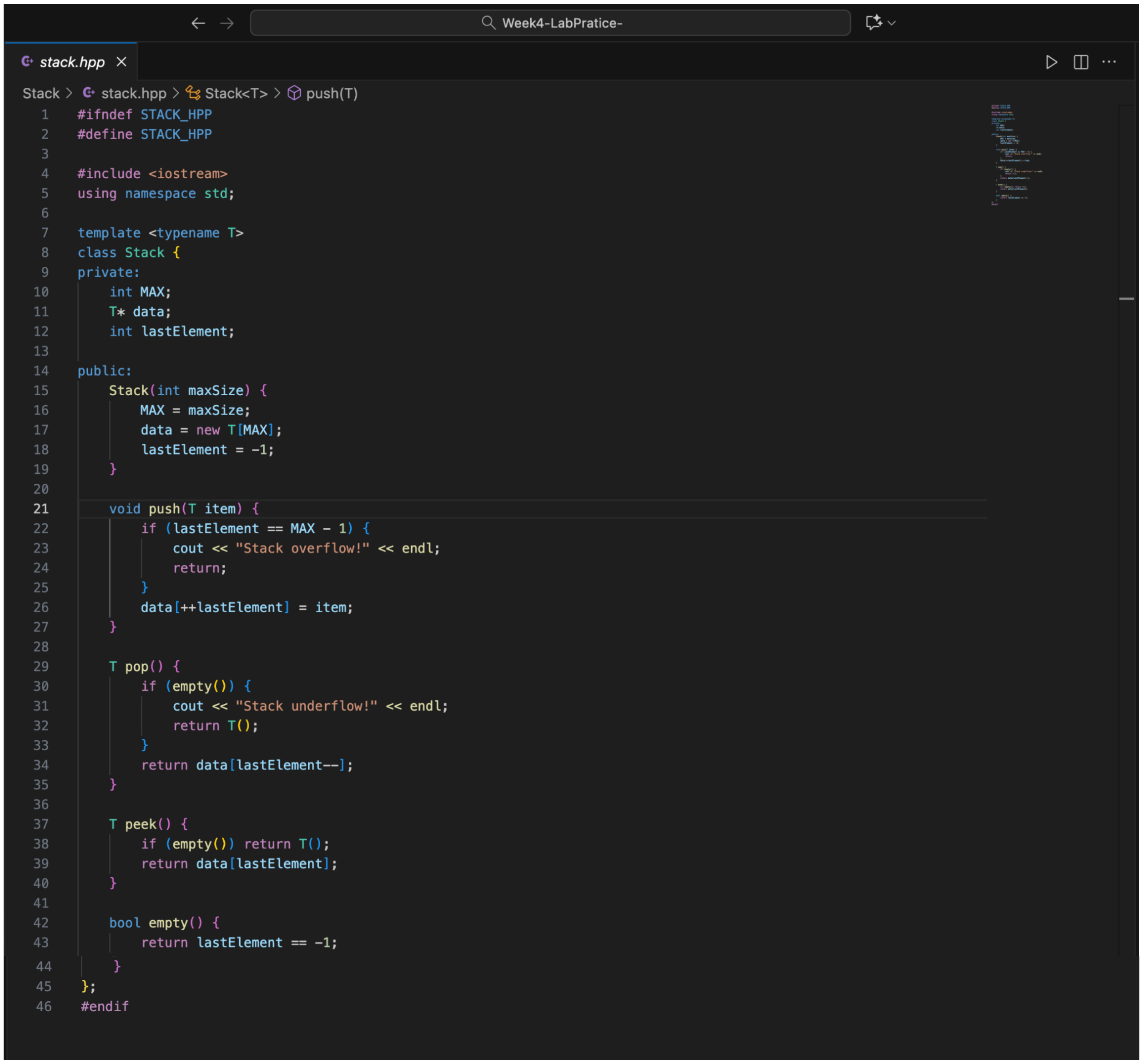
I. Code Solution :

main.hpp



```cpp
#include <iostream>
#include <string>
#include "stack.hpp"
using namespace std;

bool isOpen(char c) {
    return c == '(' || c == '{' || c == '[';
}

bool isClose(char c) {
    return c == ')' || c == '}' || c == ']';
}

bool match(char open, char close) {
    return (open == '(' && close == ')') ||
           (open == '{' && close == '}') ||
           (open == '[' && close == ']');
}

void checkBrackets(string s) {
    Stack<char> st(s.length());


    for (int i = 0; i < s.length(); i++) {
        char c = s[i];
        if (!isOpen(c) && !isClose(c)) {
            continue;
        }

        if (isOpen(c)) {
            st.push(c);
        } else {
            if (st.empty()) {
                cout << "ERROR at position " << i + 1 << ": extra closing bracket" << endl;
                return;
            }
            char top = st.pop();
            if (!match(top, c)) {
                cout << "ERROR at position " << i + 1 << ": mismatch" << endl;
                return;
            }
        }
    }

    if (!st.empty()) {
        cout << "ERROR: missing closing bracket(s)" << endl;
    } else {
        cout << "All brackets are balanced!" << endl;
    }
}

int main() {
    string s = "()))){[]}";
    cout << "Checking: " << s << endl;
    cout << endl;

    checkBrackets(s);

    return 0;
}
```

stack.hpp



```cpp
#ifndef STACK_HPP
#define STACK_HPP

#include <iostream>
using namespace std;

template <typename T>
class Stack {
private:
    int MAX;
    T* data;
    int lastElement;

public:
    Stack(int maxSize) {
        MAX = maxSize;
        data = new T[MAX];
        lastElement = -1;
    }

    void push(T item) {
        if (lastElement == MAX - 1) {
            cout << "Stack overflow!" << endl;
            return;
        }
        data[++lastElement] = item;
    }

    T pop() {
        if (empty()) {
            cout << "Stack underflow!" << endl;
            return T();
        }
        return data[lastElement--];
    }

    T peek() {
        if (empty()) return T();
        return data[lastElement];
    }

    bool empty() {
        return lastElement == -1;
    }
};
#endif
```

II. Explanation :

- **Design choice :**

We used a stack because it is the best data structure for checking balanced brackets. A stack follows the rule of Last-In, First-Out (LIFO), which means the most recent element added is the first one removed, and it matches how brackets work. Every closing bracket must match the most recent opening bracket.

We used an array-based stack since it is easy to build, runs fast, and gives direct access to the top element in constant time. The stack size is set to the same length as the input string, which guarantees enough space to hold all possible open brackets without overflow, so each time the program finds an opening bracket, it pushes it onto the stack, and when it finds a closing bracket, it pops the last open one to check if they match. This simple design makes the program efficient and easy to understand.

- **Edges Cases :**

The program was also designed to handle different edge cases :

- If the input string is empty, the output will be “OK” because there are no brackets to check.
- If there are only closing brackets (like)))), the program immediately reports an extra closing error because there are no matching opens.
- If the input has only opening brackets (like ((((), it reports missing closing brackets at the end since none of them are closed.
- When the brackets are placed incorrectly, for example ([)], the program reports a mismatch at the position where the wrong closing bracket appears.
- It also ignores all other characters that are not brackets, such as letters, numbers, or symbols, so it can handle strings that mix text and brackets.

These checks make the program reliable and able to handle different kinds of input correctly

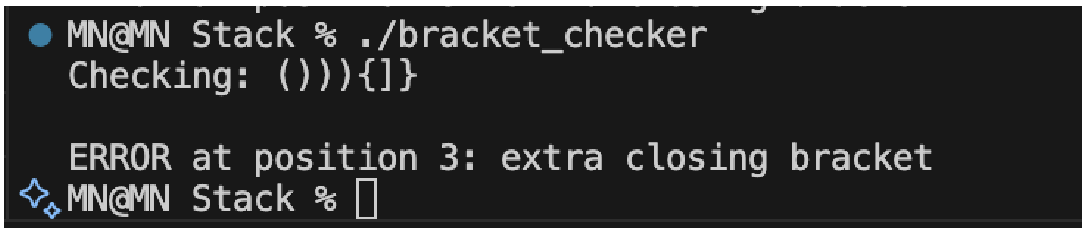
- Why it is enough :

The stack operations used in my code are push, pop, peek, and empty, and these are all that the program needs to solve the problem :

- Push operation stores each opening bracket on the stack so that it can be matched later.
- Pop operation removes and returns the most recent opening bracket when a closing bracket appears, allowing the program to check if they match.
- Peek operation can be used to look at the top element without removing it, although in this program it is optional because pop already provides the needed value.
- Empty operation checks if the stack has no items left, which is important for detecting extra closing brackets or to verify at the end if any brackets remain unclosed.

Together, these operations provide everything needed to detect balanced, mismatched, missing, or extra brackets. The algorithm checks each character once, so it runs in O(n) time and uses O(n) memory, which makes it both simple and efficient for this task.

- Output :



```
MN@MN Stack % ./bracket_checker
Checking: ())){[]}

ERROR at position 3: extra closing bracket
MN@MN Stack % 
```