

Balanced Parenthesis in C++																										
<pre>#include <iostream> #include <stack> using namespace std; bool isBal(string str) { stack<char> s; for (int i = 0; i < str.length(); i++) { if (str[i] == '(' str[i] == '{' str[i] == '[') { s.push(str[i]); } else { if (s.empty()) { return false; } else if ((str[i] == ')' && s.top() == '(') (str[i] == '}' && s.top() == '{') (str[i] == ']' && s.top() == '[')) { s.pop(); } else { return false; } } } return s.empty(); } int main() { cout << boolalpha << isBal("()") << endl; // Example usage return 0; }</pre>	<div>Function Purpose</div> <div>Checks if the string contains balanced brackets:</div> <div><ul style="list-style-type: none">() , {}, and []</div> <div>🔍 Input</div> <div>string str = "()"</div> <div>🔧 Stack Simulation Table</div> <table><tr><th>i</th><th>str[i]</th><th>Stack Before</th><th>Action</th><th>Stack After</th></tr><tr><td>0</td><td>(</td><td>[]</td><td>Push '('</td><td>['(']</td></tr><tr><td>1</td><td>(</td><td>['(']</td><td>Push '('</td><td>['(', '(']</td></tr><tr><td>2</td><td>)</td><td>['(', '(']</td><td>Top '(' matches) → Pop</td><td>['(']</td></tr><tr><td>3</td><td>)</td><td>['(']</td><td>Top '(' matches) → Pop</td><td>[]</td></tr></table> <div>✅ Final Check:</div> <div><ul style="list-style-type: none">Stack is empty → BalancedOutput: true</div> <div>💡 Output:</div> <div>true</div>	i	str[i]	Stack Before	Action	Stack After	0	([]	Push '('	['(']	1	(['(']	Push '('	['(', '(']	2)	['(', '(']	Top '(' matches) → Pop	['(']	3)	['(']	Top '(' matches) → Pop	[]
i	str[i]	Stack Before	Action	Stack After																						
0	([]	Push '('	['(']																						
1	(['(']	Push '('	['(', '(']																						
2)	['(', '(']	Top '(' matches) → Pop	['(']																						
3)	['(']	Top '(' matches) → Pop	[]																						
true																										

Largest area Histogram in C++

```
#include <iostream>
#include <stack>
#include <vector>
using namespace std;

class LargestRectangleInHistogram {
public:
    int largestRectangleArea(vector<int>& heights) {
        stack<int> s;
        int ans = 0;
        for (int i = 0; i <= heights.size(); i++) {
            int temp = (i != heights.size()) ? heights[i] : 0;
            while (!s.empty() && temp < heights[s.top()]) {
                int tbs = s.top();
                s.pop();
                int nsr = i;
                int x1 = nsr - 1;
                int nsl = (s.empty()) ? -1 : s.top();
                int x2 = nsl + 1;
                int area = heights[tbs] * (x1 - x2 + 1);
                ans = max(ans, area);
            }
            s.push(i);
        }
        return ans;
    }
};

int main() {
    vector<int> heights = {2, 1, 5, 6, 2, 3};
    LargestRectangleInHistogram histogram;
    int maxArea =
    histogram.largestRectangleArea(heights);
    cout << "The largest rectangle area is: " <<
    maxArea << endl;
    return 0;
}
```

Step-by-step Table Dry Run

i	temp	Stack (Index)	Action	Computed Area	Max Area
0	2	[]	Push index 0	—	0
1	1	[0]	Pop 0 → height = 2, width = 1 → 2×1=2	2	2
		[]	Push index 1	—	2
2	5	[1]	Push index 2	—	2
3	6	[1, 2]	Push index 3	—	2
4	2	[1, 2, 3]	Pop 3 → height = 6, width = 1 → 6×1=6	6	6
		[1, 2]	Pop 2 → height = 5, width = 2 → 5×2=10	10	10
		[1]	Push index 4	—	10
5	3	[1, 4]	Push index 5	—	10
6	0	[1, 4, 5]	Pop 5 → height = 3, width = 1 → 3×1=3	3	10
		[1, 4]	Pop 4 → height = 2, width = 3 → 2×3=6	6	10
		[1]	Pop 1 → height = 1, width = 6 → 1×6=6	6	10
		[]	Push index 6 (extra 0 at end)	—	10

	<p>✔ Final Output:</p> <p>The largest rectangle area is: 10</p>
The largest rectangle area is: 10	

Max frequency Stack in C++

```
#include <iostream>
#include <unordered_map>
#include <stack>
using namespace std;

class MaxFrequencyStack {
private:
    unordered_map<int, stack<int>>> st;
    unordered_map<int, int> fmap;
    int maxfreq;

public:
    MaxFrequencyStack() {
        maxfreq = 0;
    }

    void push(int val) {
        int f = ++fmap[val];
        st[f].push(val);
        maxfreq = max(maxfreq, f);
    }

    int pop() {
        int val = st[maxfreq].top();
        st[maxfreq].pop();
        if (st[maxfreq].empty()) {
            st.erase(maxfreq);
            maxfreq--;
        }
        fmap[val]--;
        return val;
    }
};

int main() {
    MaxFrequencyStack freqStack;

    freqStack.push(5);
    freqStack.push(7);
    freqStack.push(5);
    freqStack.push(7);
    freqStack.push(4);
    freqStack.push(5);

    cout << freqStack.pop() << endl; // Should print 5
    cout << freqStack.pop() << endl; // Should print 7
    cout << freqStack.pop() << endl; // Should print 5
    cout << freqStack.pop() << endl; // Should print 4

    return 0;
}
```

Dry Run: Input Sequence

```
push(5)
push(7)
push(5)
push(7)
push(4)
push(5)
```

```
pop() → ?
pop() → ?
pop() → ?
pop() → ?
```

■ Dry Run Table (Tracking fmap, st, and maxfreq):

Operation	fmap	st (per freq)	maxfreq	Top Element Popped
push(5)	{5: 1}	{1: [5]}	1	—
push(7)	{5: 1, 7: 1}	{1: [5, 7]}	1	—
push(5)	{5: 2, 7: 1}	{1: [5, 7], 2: [5]}	2	—
push(7)	{5: 2, 7: 2}	{1: [5, 7], 2: [5, 7]}	2	—
push(4)	{5: 2, 7: 2, 4: 1}	{1: [5, 7, 4], 2: [5, 7]}	2	—
push(5)	{5: 3, 7: 2, 4: 1}	{1: [5, 7, 4], 2: [5, 7], 3: [5]}	3	—
pop()	{5: 2, 7: 2, 4: 1}	3 is [5] → pop 5, delete 3	2	5
pop()	{5: 2, 7: 1, 4: 1}	2 is [5, 7] → pop 7	2	7
pop()	{5: 1, 7: 1, 4: 1}	2 is [5] → pop 5, delete 2	1	5
pop()	{5: 1, 7: 1, 4: 0}	1 is [5, 7, 4] → pop 4	1	4

	<div>✓ Output:</div> <div>5 7 5 4</div> <div>💡 Notes:</div>
5 7 5 4	

Min Stack in C++

```
#include <iostream>
#include <stack>
#include <climits>
using namespace std;

class MinStack {
private:
    stack<long long> st;
    long long minVal;

public:
    MinStack() {
        minVal = INT_MAX;
    }

    void push(int val) {
        if (st.empty()) {
            minVal = val;
            st.push(0LL);
        } else {
            long long diff = val - minVal;
            st.push(diff);
            if (val < minVal) {
                minVal = val;
            }
        }
    }

    void pop() {
        long long rem = st.top();
        st.pop();
        if (rem < 0) {
            minVal = minVal - rem;
        }
    }

    int top() {
        long long rem = st.top();
        if (rem < 0) {
            return static_cast<int>(minVal);
        } else {
            return static_cast<int>(minVal + rem);
        }
    }

    int getMin() {
        return static_cast<int>(minVal);
    }
};


int main() {
    MinStack minStack;

    minStack.push(2);
    minStack.push(0);
    minStack.push(3);
    minStack.push(0);

    cout << "Minimum value: " << minStack.getMin()
    << endl; // Should print 0
    minStack.pop();
```

Core Logic Recap

- st stores **differences** between the current value and minVal.
- If the pushed value is **less than** minVal, a **negative diff** is stored. This signals a **new min**.
- When popping, if the top is negative, we **recalculate the previous min** using minVal - rem.

 Test Input:
 minStack.push(2);
 minStack.push(0);
 minStack.push(3);
 minStack.push(0);

pop() → getMin()
 pop() → getMin()
 pop() → getMin()

📋 Dry Run Table:

Operation	Stack (diffs)	minVal	Explanation
push(2)	[0]	2	First element → diff is 0
push(0)	[0, -2]	0	0 < 2 → store diff (-2), update minVal
push(3)	[0, -2, 3]	0	3 > 0 → store diff (3), minVal unchanged
push(0)	[0, -2, 3, 0]	0	0 = minVal → store diff (0), minVal unchanged
pop()	[0, -2, 3]	0	popped 0, not negative → minVal stays
getMin()	—	0	
pop()	[0, -2]	0	popped 3 (diff=3), not negative → minVal stays
getMin()	—	0	
pop()	[0]	2	popped -2 → was a new min at the time → rollback
getMin()	—	2	

```
    cout << "Minimum value: " << minStack.getMin()
<< endl; // Should print 0
    minStack.pop();
    cout << "Minimum value: " << minStack.getMin()
<< endl; // Should print 0
    minStack.pop();
    cout << "Minimum value: " << minStack.getMin()
<< endl; // Should print 2

    return 0;
}
```

✔ **Output:**

Minimum value: 0
Minimum value: 0
Minimum value: 0
Minimum value: 2

Minimum value: 0
Minimum value: 0
Minimum value: 0
Minimum value: 2

Next Greater on the Right in C++

```
#include <iostream>
#include <stack>
using namespace std;

long* nextLargerElement(long* arr, int n)
{
    long* ans = new long[n];
    stack<int> st;
    for(int i = 0; i < n; i++){
        while(!st.empty() && arr[i] > arr[st.top()]){
            int idx = st.top();
            st.pop();
            ans[idx] = arr[i];
        }
        st.push(i);
    }
    while(!st.empty()){
        int idx = st.top();
        st.pop();
        ans[idx] = -1;
    }

    return ans;
}

int main() {
    long arr[] = {4, 8, 5, 2, 25};
    int n = sizeof(arr) / sizeof(arr[0]);

    long* result = nextLargerElement(arr, n);

    cout << "Resulting array:" << endl;
    for (int i = 0; i < n; i++) {
        cout << result[i] << " ";
    }
    cout << endl;

    delete[] result; // Free dynamically allocated memory

    return 0;
}
```

Input:

arr = {4, 8, 5, 2, 25}
n = 5



Iterative Dry Run Table:

i	arr[i]	Stack (indices)	Top Value	Condition Checked	Action Taken	ans Array
0	4	[]	—	—	Push index 0	[-, -, -, -, -]
1	8	[0]	4	8 > 4 → true	Pop 0, set ans[0] = 8, push 1	[8, -, -, -, -]
2	5	[1]	8	5 > 8 → false	Push 2	[8, -, -, -, -]
3	2	[1, 2]	5	2 > 5 → false	Push 3	[8, -, -, -, -]
4	25	[1, 2, 3]	2	25 > 2 → true	Pop 3, ans[3] = 25	[8, -, -, 25, -]
		[1, 2]	5	25 > 5 → true	Pop 2, ans[2] = 25	[8, -, 25, 25, -]
		[1]	8	25 > 8 → true	Pop 1, ans[1] = 25	[8, 25, 25, 25, -]
		[]	—	—	Push 4	[8, 25, 25, 25, -]
—	—	[4]	25	Loop ends	Pop 4, set ans[4] = -1	[8, 25, 25, 25, -1]

✓ Final Output:

Resulting array:
8 25 25 25 -1

Resulting array:
8 25 25 25 -1

Postfix 2 Prefix in C++																																		
<pre>#include <iostream> #include <stack> using namespace std; // Function to convert a postfix expression to a prefix expression. string postToPre(string exp) { stack<string> op; for (int i = 0; i < exp.length(); i++) { char ch = exp[i]; if (ch == '+' ch == '-' ch == '*' ch == '/') { string val2 = op.top(); op.pop(); string val1 = op.top(); op.pop(); string cal = ch + val1 + val2; op.push(cal); } else { op.push(string(1, ch)); } } return op.top(); } int main() { string postfix1 = "ab+c*"; cout << "Postfix: " << postfix1 << " -> Prefix: " << postToPre(postfix1) << endl; // Expected: "+abc" return 0; }</pre>		<p>Input:</p> <p>Postfix Expression = "ab+c*" Expected Prefix = "+abc"</p> <p> Dry Run Table:</p> <table><tr><th>i</th><th>ch</th><th>Stack Before</th><th>Action</th><th>Stack After</th></tr><tr><td>0</td><td>'a'</td><td>[]</td><td>Operand → push "a"</td><td>["a"]</td></tr><tr><td>1</td><td>'b'</td><td>["a"]</td><td>Operand → push "b"</td><td>["a", "b"]</td></tr><tr><td>2</td><td>'+'</td><td>["a", "b"]</td><td>Operator → pop "b", "a" → form +ab, push it</td><td>["+ab"]</td></tr><tr><td>3</td><td>'c'</td><td>["+ab"]</td><td>Operand → push "c"</td><td>["+ab", "c"]</td></tr><tr><td>4</td><td>'*'</td><td>["+ab", "c"]</td><td>Operator → pop "c", "+ab" → form *+abc, push it</td><td>["+*+abc"]</td></tr></table> <p> Final Output:</p> <p>Prefix: *+abc</p>			i	ch	Stack Before	Action	Stack After	0	'a'	[]	Operand → push "a"	["a"]	1	'b'	["a"]	Operand → push "b"	["a", "b"]	2	'+'	["a", "b"]	Operator → pop "b", "a" → form +ab, push it	["+ab"]	3	'c'	["+ab"]	Operand → push "c"	["+ab", "c"]	4	'*'	["+ab", "c"]	Operator → pop "c", "+ab" → form *+abc, push it	["+*+abc"]
i	ch	Stack Before	Action	Stack After																														
0	'a'	[]	Operand → push "a"	["a"]																														
1	'b'	["a"]	Operand → push "b"	["a", "b"]																														
2	'+'	["a", "b"]	Operator → pop "b", "a" → form +ab, push it	["+ab"]																														
3	'c'	["+ab"]	Operand → push "c"	["+ab", "c"]																														
4	'*'	["+ab", "c"]	Operator → pop "c", "+ab" → form *+abc, push it	["+*+abc"]																														
Postfix: ab+c* -> Prefix: *+abc																																		

Prefix to Postfix in C++

```
#include <iostream>
#include <stack>
#include <string>
using namespace std;

// Function to convert a prefix expression to a postfix expression.
string preToPost(string exp) {
    stack<string> op;
    int n = exp.length();
    for (int i = n - 1; i >= 0; i--) {
        char ch = exp[i];
        if (ch == '+' || ch == '-' || ch == '*' || ch == '/') {
            string val1 = op.top();
            op.pop();
            string val2 = op.top();
            op.pop();
            string cal = val1 + val2 + ch;
            op.push(cal);
        } else {
            op.push(string(1, ch));
        }
    }
    return op.top();
}

int main() {
    string prefix1 = "+AB-CDE";
    cout << "Prefix: " << prefix1 << " -> Postfix: " <<
    preToPost(prefix1) << endl; // Expected: "ABC+DE-*"

    string prefix2 = "-A/BC-/DEFG";
    cout << "Prefix: " << prefix2 << " -> Postfix: " <<
    preToPost(prefix2) << endl; // Expected:
    "ABC/-DE/FG-*"

    // Add more test cases as needed

    return 0;
}
```


📌 Dry Run Table:

i (index)	ch	Stack Before	Action	Stack After
7	'E'	[]	Operand → push "E"	["E"]
6	'D'	["E"]	Operand → push "D"	["E", "D"]
5	'C'	["E", "D"]	Operand → push "C"	["E", "D", "C"]
4	'.'	["E", "D", "C"]	Operator → pop "C" & "D" → "CD-"	["E", "CD-"]
3	'B'	["E", "CD-"]	Operand → push "B"	["E", "CD-", "B"]
2	'A'	["E", "CD-", "B"]	Operand → push "A"	["E", "CD-", "B", "A"]
1	'+'	["E", "CD-", "B", "A"]	Operator → pop "A" & "B" → "AB+"	["E", "CD-", "AB+"]
0	'*'	["E", "CD-", "AB+"]	Operator → pop "AB+" & "CD-" → "AB+CD-*"	["AB+CD- *"]

Final Result:

Top of the stack: **"AB+CD-*"**

Prefix: +AB-CDE -> Postfix: AB+CD-*
 Prefix: *-A/BC-/DEFG -> Postfix: ABC/-DE/F.*

Remove adjacent duplicate in C++																															
<pre>#include <iostream> #include <stack> #include <string> using namespace std; string removeAdjacentDuplicates(string s) { stack<char> st; for (char ch : s) { if (!st.empty() && st.top() == ch) { st.pop(); } else { st.push(ch); } } string result = ""; while (!st.empty()) { result = st.top() + result; st.pop(); } return result; } int main() { string s = "abbaca"; cout << removeAdjacentDuplicates(s) << endl; // Output: "ca" return 0; }</pre>		<p>Input:</p> <p>string s = "abbaca";</p> <p> Step-by-Step Stack Trace:</p> <table><tr><th>Step</th><th>Char</th><th>Stack (top to bottom)</th><th>Action</th></tr><tr><td>1</td><td>'a'</td><td>[a]</td><td>Push</td></tr><tr><td>2</td><td>'b'</td><td>[a, b]</td><td>Push</td></tr><tr><td>3</td><td>'b'</td><td>[a]</td><td>'b' == top → Pop</td></tr><tr><td>4</td><td>'a'</td><td>[]</td><td>'a' == top → Pop</td></tr><tr><td>5</td><td>'c'</td><td>[c]</td><td>Push</td></tr><tr><td>6</td><td>'a'</td><td>[c, a]</td><td>Push</td></tr></table> <p>✔ Final Stack (bottom to top): c a</p> <p>So result = "ca".</p>		Step	Char	Stack (top to bottom)	Action	1	'a'	[a]	Push	2	'b'	[a, b]	Push	3	'b'	[a]	'b' == top → Pop	4	'a'	[]	'a' == top → Pop	5	'c'	[c]	Push	6	'a'	[c, a]	Push
Step	Char	Stack (top to bottom)	Action																												
1	'a'	[a]	Push																												
2	'b'	[a, b]	Push																												
3	'b'	[a]	'b' == top → Pop																												
4	'a'	[]	'a' == top → Pop																												
5	'c'	[c]	Push																												
6	'a'	[c, a]	Push																												
ca																															

Smaller no on left in C++

```
#include <iostream>
#include <vector>
#include <stack>

using namespace std;

vector<int> leftSmaller(int n, int a[]) {
    vector<int> ans(n);
    stack<int> st;

    for (int i = n - 1; i >= 0; i--) {
        while (!st.empty() && a[i] < a[st.top()]) {
            int idx = st.top();
            ans[idx] = a[i];
            st.pop();
        }
        st.push(i);
    }

    while (!st.empty()) {
        int idx = st.top();
        ans[idx] = -1;
        st.pop();
    }

    return ans;
}

int main() {
    int arr[] = {4, 8, 5, 2, 25};
    int n = sizeof(arr) / sizeof(arr[0]);

    vector<int> result = leftSmaller(n, arr);

    cout << "Resulting list:" << endl;
    for (int i : result) {
        cout << i << " ";
    }
    cout << endl;

    return 0;
}
```

Input:
arr = {4, 8, 5, 2, 25}

📌 Dry Run Table:

i	arr[i]	Stack (index)	Action	ans (after step)
4	25	[]	Stack empty, push 4	[?, ?, ?, ?, ?]
3	2	[4]	2 < 25 → ans[4] = 2, pop 4; push 3	[?, ?, ?, ?, 2]
2	5	[3]	5 > 2 → push 2	[?, ?, ?, ?, 2]
1	8	[3, 2]	8 > 5 → push 1	[?, ?, ?, ?, 2]
0	4	[3, 2, 1]	4 < 8 → ans[1] = 4, pop 1; 4 < 5 → ans[2] = 4, pop 2; push 0	[?, 4, 4, ?, 2]
		[3, 0]	Final elements → set ans[3] = -1, ans[0] = -1	[-1, 4, 4, -1, 2]

✓ Final Output:

-1 4 4 -1 2

✓ Explanation (Index-wise):

Index	arr[i]	Left Smaller Element
0	4	-1 (nothing to the left)
1	8	4
2	5	4
3	2	-1
4	25	2

Resulting list:
-1 4 4 -1 2

Stock Span in C++

```
#include <iostream>
#include <vector>
#include <stack>

using namespace std;

void stockSpan(vector<int>& arr) {
    stack<int> s;
    s.push(0); // Push index of the first element

    for (int i = 0; i < arr.size(); i++) {
        // Pop elements from stack while the current
        // price is greater than the price of the element at the
        // top of the stack
        while (!s.empty() && arr[s.top()] <= arr[i]) {
            s.pop();
        }

        // Calculate span (i - index at top of stack after
        // popping or i + 1 if stack is empty)
        int span = s.empty() ? (i + 1) : (i - s.top());

        // Print the span
        cout << span << " ";

        // Push the current index onto the stack
        s.push(i);
    }
}

int main() {
    // Test case: array of stock prices
    vector<int> arr = {15, 13, 12, 14, 15};
    stockSpan(arr);
    cout << endl;

    return 0;
}
```

For each day i , the span is: The number of consecutive previous days (including the current day) for which the price was **less than or equal to today's price**.

You're maintaining a stack of **indices**, and for each price:

- You **pop** indices from the stack if the current price is higher than the price at the stack's top.
- The **span** is then $i - s.top()$ or $i + 1$ if the stack is empty.

 Input:

arr = {15, 13, 12, 14, 15}

 **Dry Run Table:**

Day (i)	Price	Stack (indices)	Stack (values)	Span	Explanation
0	15	[]	[]	1	Stack empty → span = 0 + 1
		[0]	[15]		Push index 0
1	13	[0]	[15]	1	13 < 15 → span = 1 - 0
		[0, 1]	[15, 13]		
2	12	[0, 1]	[15, 13]	1	12 < 13 → span = 2 - 1
		[0, 1, 2]	[15, 13, 12]		
3	14	[0, 1, 2] → pop 2, 1	[15]	3	14 > 13 & 12 → span = 3 - 0
		[0, 3]	[15, 14]		
4	15	[0, 3] → pop 3, 0	[]	5	15 >= 14, 15 → stack empty → span = 4 + 1
		[4]	[15]		

✓ **Output: 1 1 1 3 5**

1 1 1 3 5