Celebrity in C++

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

void findCelebrity(int arr[][4], int n) {
    stack<int> st;
    for (int i = 0; i < n; i++) {
        st.push(i);
    }

    while (st.size() > 1) {
        int i = st.top();
        st.pop();
        int j = st.top();
        st.pop();

        if (arr[i][j] == 1) {
            st.push(j);
        } else {
            st.push(i);
        }
    }

    int potential = st.top();
    bool isCelebrity = true;
    for (int i = 0; i < n; i++) {
        if (i != potential) {
            if (arr[i][potential] == 0 || arr[potential]
[i] == 1) {
                isCelebrity = false;
                break;
            }
        }
    }

    if (isCelebrity) {
        cout << potential << endl;
    } else {
        cout << "none" << endl;
    }
}

int main() {
    // Hardcoded input
    int n = 4;
    int arr[4][4] = {
        {0, 0, 0, 0},
        {1, 0, 1, 1},
        {1, 1, 0, 1},
        {1, 1, 1, 0}
    };

    // Finding the celebrity
    findCelebrity(arr, n);

    return 0;
}
```

Each cell arr[i][j] tells us whether person i knows person j.

```
int arr[4][4] = {
    {0, 0, 0, 0},  // Person 0 knows nobody
    {1, 0, 1, 1},  // Person 1 knows 0, 2, 3
    {1, 1, 0, 1},  // Person 2 knows 0, 1, 3
    {1, 1, 1, 0}   // Person 3 knows 0, 1, 2
};
```

📦 Stack-Based Elimination Table

Step	Stack Before	i (pop1)	j (pop2)	arr[i][j]	Action Taken	Stack After
1	[0, 1, 2, 3]	3	2	1	3 knows 2 → eliminate 3	[0, 1, 2]
2	[0, 1, 2]	2	1	1	2 knows 1 → eliminate 2	[0, 1]
3	[0, 1]	1	0	1	1 knows 0 → eliminate 1	[0]

Now stack.top() gives us **potential celebrity = 0**

🔍 Verification Table

Check if person 0 is a **celebrity**:

i	arr[i][0] (i knows 0)	arr[0][i] (0 knows i)	Condition Satisfied?
0	—	—	Skip self
1	1	0	✅ Person 1 knows 0, 0 knows no one
2	1	0	✅ Person 2 knows 0
3	1	0	✅ Person 3 knows 0

✔ All conditions met — 0 is a celebrity

✅ Final Output:
0

0

Merge overlapping Interval in C++

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

// Structure to represent a pair of start and end times
struct Pair {
    int st;
    int et;

    Pair(int s, int e) {
        st = s;
        et = e;
    }
};

// Comparator function to sort pairs based on start time
bool comparePairs(const Pair& a, const Pair& b) {
    return a.st < b.st;
}

// Function to merge overlapping intervals and print in increasing order of start time
void mergeOverlappingIntervals(vector<Pair>& intervals) {
    // Sort intervals based on start time
    sort(intervals.begin(), intervals.end(), comparePairs);

    stack<Pair> st;
    st.push(intervals[0]);

    for (int i = 1; i < intervals.size(); i++) {
        Pair top = st.top();

        // If current interval overlaps with the top of the stack, merge them
        if (intervals[i].st <= top.et) {
            top.et = max(top.et, intervals[i].et);
            st.pop();
            st.push(top);
        } else {
            st.push(intervals[i]);
        }
    }

    // Output the merged intervals in sorted order
    stack<Pair> result;
    while (!st.empty()) {
        result.push(st.top());
        st.pop();
    }

    while (!result.empty()) {
        Pair p = result.top();
        cout << p.st << " " << p.et << endl;
        result.pop();
    }
}
```

Input Intervals (Unsorted)

{22, 28}
{1, 8}
{25, 27}
{14, 19}
{27, 30}
{5, 12}

🔁 Step 1: Sort Intervals by Start Time

After sorting using comparePairs, the list becomes:

Index	Start	End
0	1	8
1	5	12
2	14	19
3	22	28
4	25	27
5	27	30

🔁 Step 2: Merge Overlapping Intervals using Stack

i	Current Interval	Top of Stack	Action	Stack Content
0	{1, 8}	-	Push first interval	[{1, 8}]
1	{5, 12}	{1, 8}	Overlaps, merge to {1, 12}	[{1, 12}]
2	{14, 19}	{1, 12}	No overlap, push	[{1, 12}, {14, 19}]
3	{22, 28}	{14, 19}	No overlap, push	[{1, 12}, {14, 19}, {22, 28}]
4	{25, 27}	{22, 28}	Overlaps, merge to {22, 28}	[{1, 12}, {14, 19}, {22, 28}] (no change)
5	{27, 30}	{22, 28}	Overlaps, merge to {22, 30}	[{1, 12}, {14, 19}, {22, 30}]

```
int main() {
    // Hardcoded input
    vector<Pair> intervals = {
        {22, 28},
        {1, 8},
        {25, 27},
        {14, 19},
        {27, 30},
        {5, 12}
    };

    // Calling the function to merge overlapping
intervals
    mergeOverlappingIntervals(intervals);

    return 0;
}
```

📜 **Final Stack (top to bottom):**

{22, 30}
{14, 19}
{1, 12}

🔚 **Step 3: Print Intervals in Sorted Order**

We reverse the stack to maintain start-time order:

1 12
14 19
22 30

✅ **Output:**

1 12
14 19
22 30

1 12
14 19
22 30

Sliding Window max in C++

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

vector<int> slidingWindowMaximum(vector<int>&
arr, int k) {
   int n = arr.size();
   vector<int> result;
   stack<int> st;
   vector<int> nge(n);

   st.push(n-1);
   nge[n-1] = n;

   for (int i = n-2; i >= 0; i--) {
      while (!st.empty() && arr[i] >= arr[st.top()]) {
         st.pop();
      }

      if (st.empty()) {
         nge[i] = n;
      } else {
         nge[i] = st.top();
      }

      st.push(i);
   }

   for (int i = 0; i <= n-k; i++) {
      int j = i;
      while (nge[j] < i+k) {
         j = nge[j];
      }

      result.push_back(arr[j]);
   }

   return result;
}

int main() {
   // Hardcoded input
   vector<int> arr = {1, 3, -1, -3, 5, 3, 6, 7};
   int k = 3;

   vector<int> result = slidingWindowMaximum(arr,
k);

   // Output the result
   for (int num : result) {
      cout << num << " ";
   }
   cout << endl;

   return 0;
}
```

Input:
arr = {1, 3, -1, -3, 5, 3, 6, 7}
k = 3
n = 8

Step 1: Compute Next Greater Element Index Array (nge[])

We initialize an array nge[n], where:

- nge[i] = index of the next greater element to the right of arr[i]
- If no such index, set nge[i] = n

NGE Construction Table

We build from **right to left** using a stack:

i	arr[i]	Stack (Top to Bottom)	nge[i]
7	7	[7]	8
6	6	[7, 6]	7
5	3	[7, 6, 5]	6
4	5	[7, 6, 4]	6
3	-3	[7, 6, 4, 3]	4
2	-1	[7, 6, 4, 2]	4
1	3	[7, 6, 4, 1]	4
0	1	[7, 6, 4, 1, 0]	1

➞ Final nge[] = {1, 4, 4, 4, 6, 6, 7, 8}

Step 2: Compute Max in Each Sliding Window

For each window starting at i, you walk forward using nge[] until nge[j] >= i + k.

Sliding Window Loop (i = 0 to n - k)

i	Window	j Traversal (via NGE)	Max Value
0	[1 3 -1]	0 → 1	3
1	[3 -1 -3]	1 → 4 (exits, 4 ≥ 4)	3
2	[-1 -3 5]	2 → 4	5
3	[-3 5 3]	3 → 4	5
4	[5 3 6]	4 → 6	6
5	[3 6 7]	5 → 6 → 7	7

	✓ Output: 3 3 5 5 6 7
3 3 5 5 6 7	

Two Stacks in C++

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

class TwoStack {
private:
    vector<int> data;
    int tos1;  // Top of stack 1
    int tos2;  // Top of stack 2
public:
    TwoStack(int cap) {
        // Constructor to initialize the two stacks
        data.resize(cap);  // Resize the vector to given capacity
        tos1 = -1;  // Initialize top of stack 1 to -1
        tos2 = cap;  // Initialize top of stack 2 to cap (end of array)
    }

    int size1() {
        // Returns the size of stack 1
        return tos1 + 1;
    }

    int size2() {
        // Returns the size of stack 2
        return data.size() - tos2;
    }

    void push1(int val) {
        // Pushes an element onto stack 1
        if (tos2 == tos1 + 1) {
            cout << "Stack overflow\n";
        } else {
            tos1++;
            data[tos1] = val;
        }
    }

    void push2(int val) {
        // Pushes an element onto stack 2
        if (tos2 == tos1 + 1) {
            cout << "Stack overflow\n";
        } else {
            tos2--;
            data[tos2] = val;
        }
    }

    int pop1() {
        // Pops an element from stack 1
        if (size1() == 0) {
            cout << "Stack underflow\n";
            return -1;
        } else {
            int val = data[tos1];
            tos1--;
            return val;
        }
    }
```

Let's break down a **tabular dry run** of your TwoStack implementation with:

Capacity = 5
Operations = push1(10), push1(20), push2(30), push2(40), push2(50), push1(60)

🧠 Initial State

- data: [_, _, _, _, _]
- tos1 = -1, tos2 = 5
- Stack 1 grows → from index 0
- Stack 2 grows ← from index 4

🧪 Operation by Operation

Operation	tos1	tos2	data (0 to 4)	Note
push1(10)	0	5	[10, _, _, _, _]	10 pushed to stack 1
push1(20)	1	5	[10, 20, _, _, _]	20 pushed to stack 1
push2(30)	1	4	[10, 20, _, _, 30]	30 pushed to stack 2
push2(40)	1	3	[10, 20, _, 40, 30]	40 pushed to stack 2
push2(50)	1	2	[10, 20, 50, 40, 30]	50 pushed to stack 2
push1(60)	-	-	Overflow!	tos2 == tos1 + 1 (2 == 2)

🧪 Stack Status

- Stack1 (left): [10, 20] → tos1 = 1
- Stack2 (right): [50, 40, 30] → tos2 = 2

📜 Output Operations

```
cout << "top1: " << st.top1() << "\n"; // 20
cout << "pop1: " << st.pop1() << "\n"; // 20
cout << "top1: " << st.top1() << "\n"; // 10
cout << "pop1: " << st.pop1() << "\n"; // 10
cout << "top2: " << st.top2() << "\n"; // 50
cout << "pop2: " << st.pop2() << "\n"; // 50
cout << "top2: " << st.top2() << "\n"; // 40
cout << "pop2: " << st.pop2() << "\n"; // 40
cout << "top1: " << st.top1() << "\n"; // Underflow (-1)
cout << "pop1: " << st.pop1() << "\n"; // Underflow (-1)
cout << "top2: " << st.top2() << "\n"; // 30
cout << "pop2: " << st.pop2() << "\n"; // 30
```

```
    int pop2() {
        // Pops an element from stack 2
        if (size2() == 0) {
            cout << "Stack underflow\n";
            return -1;
        } else {
            int val = data[tos2];
            tos2++;
            return val;
        }
    }

    int top1() {
        // Returns the top element of stack 1
        if (size1() == 0) {
            cout << "Stack underflow\n";
            return -1;
        } else {
            return data[tos1];
        }
    }

    int top2() {
        // Returns the top element of stack 2
        if (size2() == 0) {
            cout << "Stack underflow\n";
            return -1;
        } else {
            return data[tos2];
        }
    }
};

int main() {
    // Hardcoded example
    int capacity = 5;
    TwoStack st(capacity);

    // Perform operations
    st.push1(10);
    st.push1(20);
    st.push2(30);
    st.push2(40);
    st.push2(50);
    st.push1(60);

    cout << "top1: " << st.top1() << "\n";
    cout << "pop1: " << st.pop1() << "\n";
    cout << "top1: " << st.top1() << "\n";
    cout << "pop1: " << st.pop1() << "\n";
    cout << "top2: " << st.top2() << "\n";
    cout << "pop2: " << st.pop2() << "\n";
    cout << "top2: " << st.top2() << "\n";
    cout << "pop2: " << st.pop2() << "\n";
    cout << "top1: " << st.top1() << "\n";
    cout << "pop1: " << st.pop1() << "\n";
    cout << "top2: " << st.top2() << "\n";
    cout << "pop2: " << st.pop2() << "\n";

    return 0;
}
```

✅ Final Stack States

- Stack1: empty
- Stack2: empty
- tos1 = -1, tos2 = 5

Stack overflow

```
top1: 20
pop1: 20
top1: 10
pop1: 10
top2: 50
pop2: 50
top2: 40
pop2: 40
Stack underflow
top1: -1
Stack underflow
pop1: -1
top2: 30
pop2: 30
```