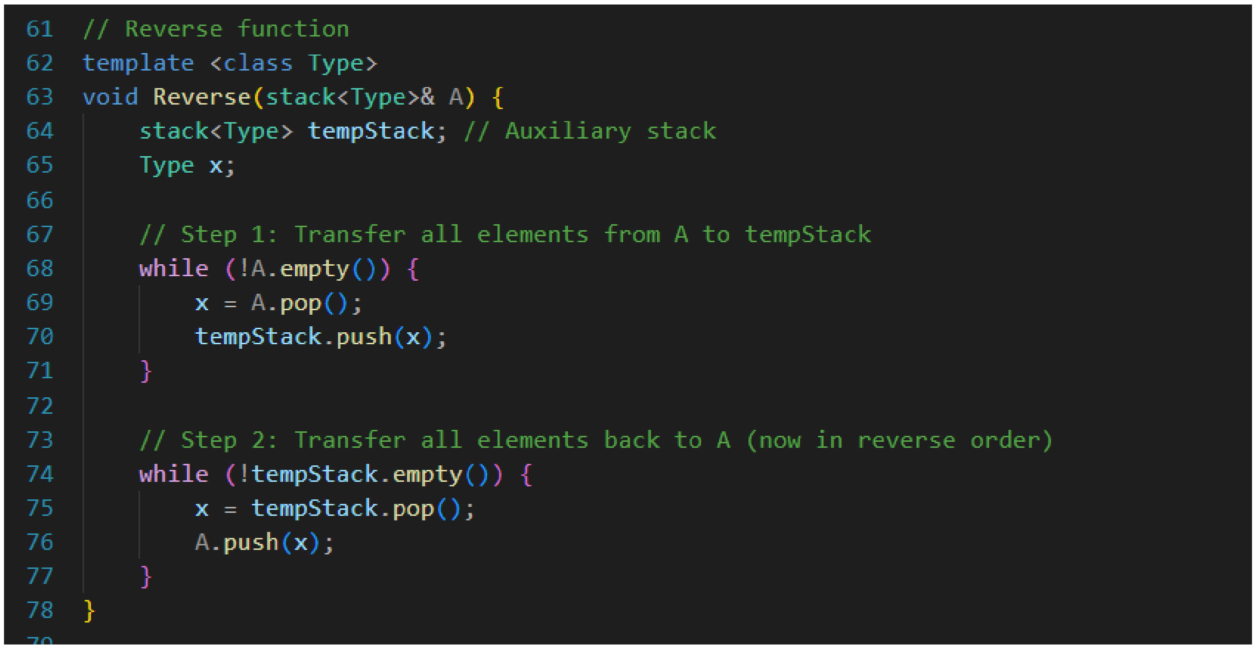
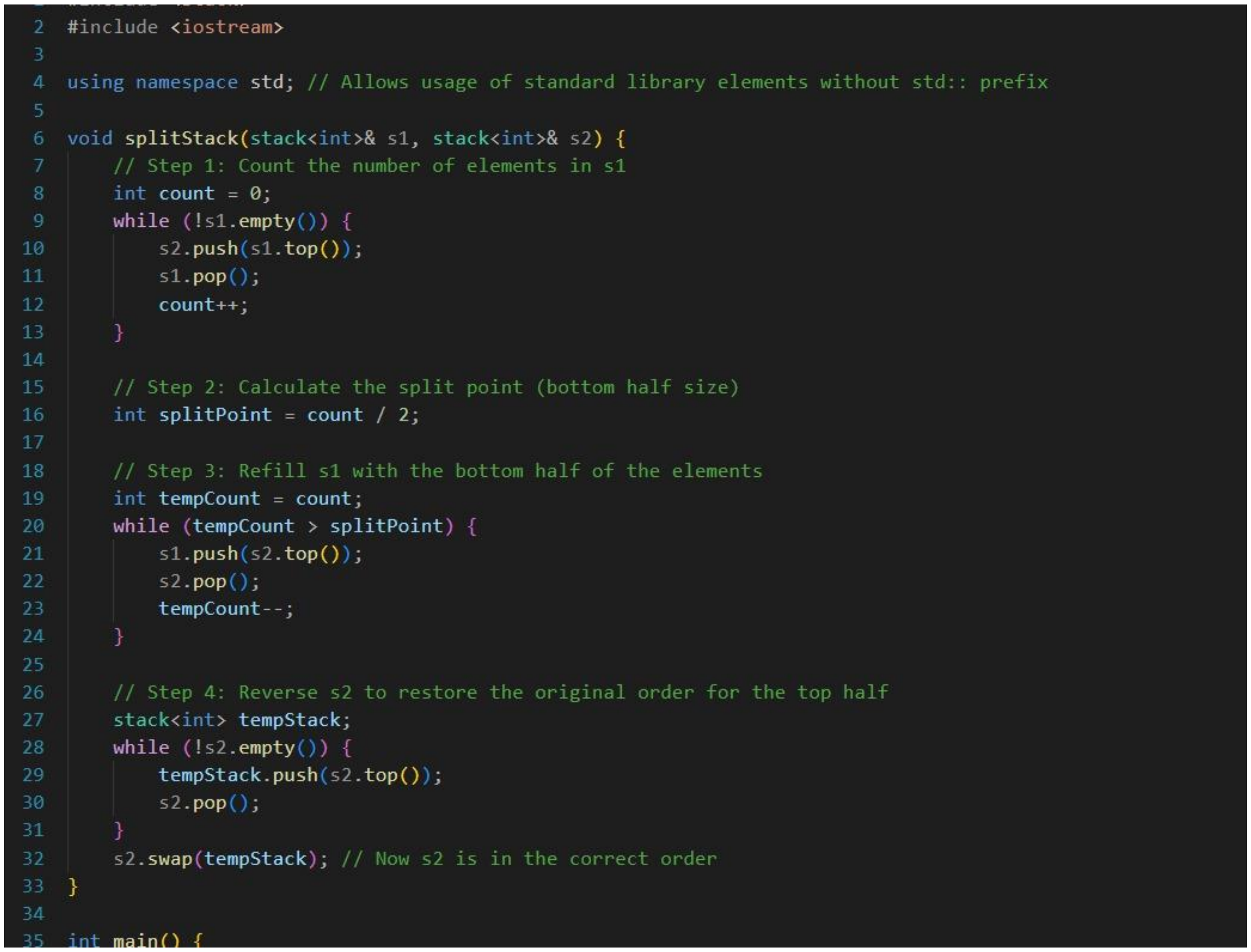
Reverse Stack



```
// Reverse function
template <class Type>
void Reverse(stack<Type>& A) {
    stack<Type> tempStack; // Auxiliary stack
    Type x;

    // Step 1: Transfer all elements from A to tempStack
    while (!A.empty()) {
        x = A.pop();
        tempStack.push(x);
    }

    // Step 2: Transfer all elements back to A (now in reverse order)
    while (!tempStack.empty()) {
        x = tempStack.pop();
        A.push(x);
    }
}
```

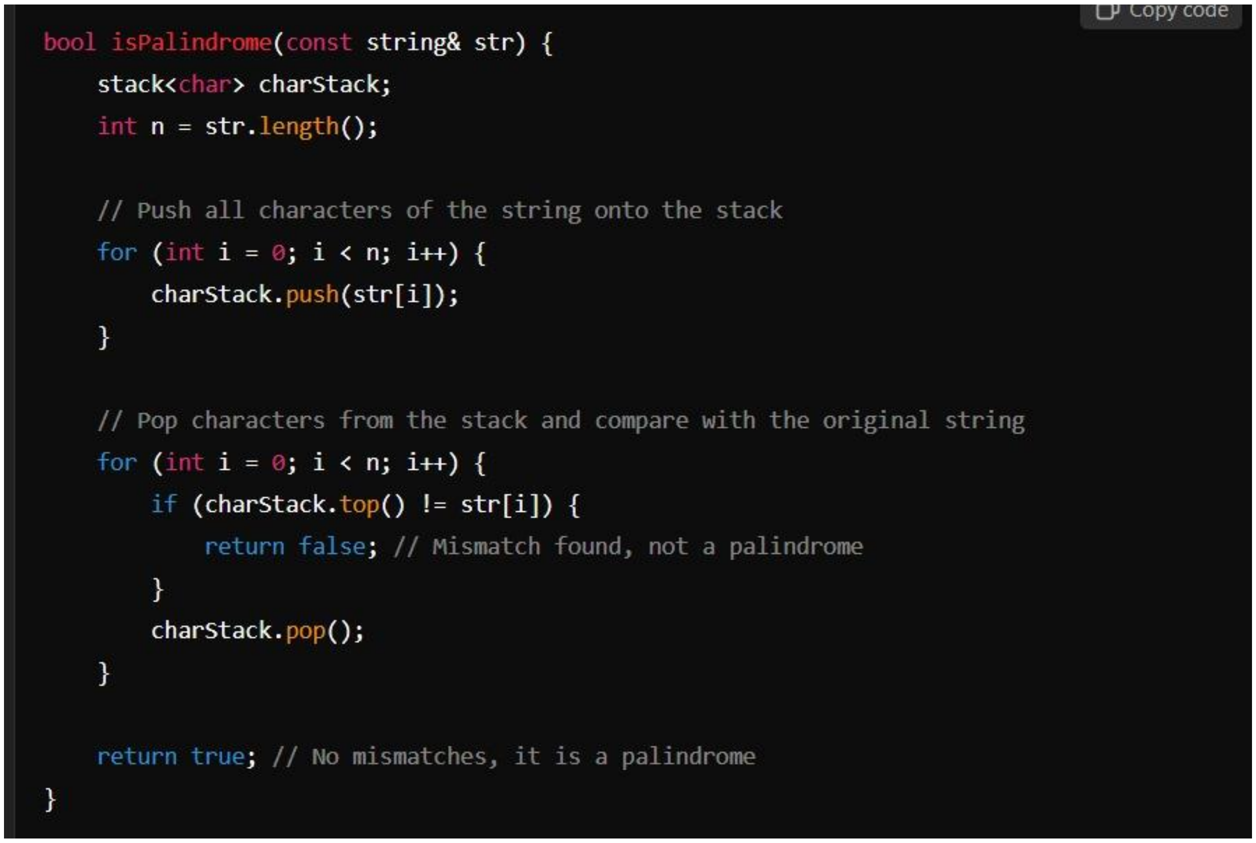
```cpp
#include"stack.h"

template <class Type>
bool stack<Type>::empty()
{
        if (top == NULL)
                return true;
        else
                return false;
}

template <class Type>
stack<Type>::stack()
{
        size = 0;
        top = NULL;
}

template <class Type>
void stack<Type>::push(Type element)
{
        node<Type>* newNode = new node<Type>();
        newNode->setdata(element);
      → newNode->setnext(top);
        top = newNode;
        size++;
}

template <class Type>
Type stack<Type>::pop()
{
        if (!empty())
        {
                node<Type>* temp = top;
              → Type element = temp->getdata();
                top = top->getnext();
                size--;
                delete temp;
                return element;
        }
        else
        {
                return 0;
        }
}

```

Split Stack

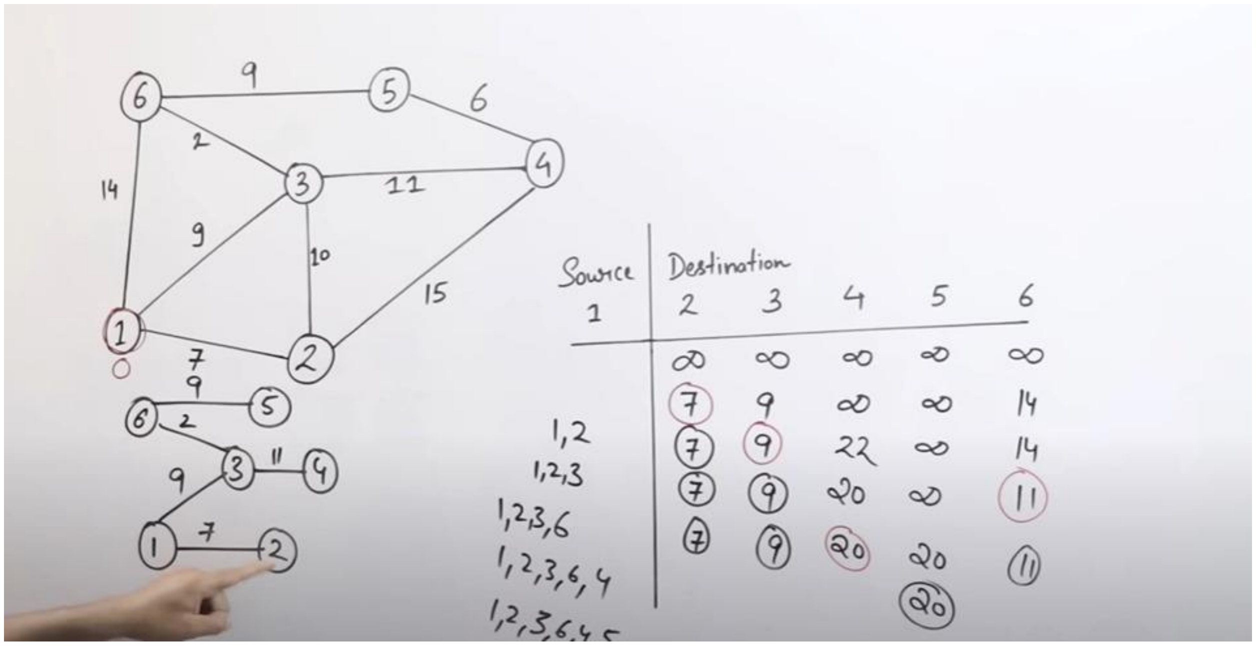


```cpp
#include <iostream>

using namespace std; // Allows usage of standard library elements without std:: prefix

void splitStack(stack<int>& s1, stack<int>& s2) {
    // Step 1: Count the number of elements in s1
    int count = 0;
    while (!s1.empty()) {
        s2.push(s1.top());
        s1.pop();
        count++;
    }

    // Step 2: Calculate the split point (bottom half size)
    int splitPoint = count / 2;

    // Step 3: Refill s1 with the bottom half of the elements
    int tempCount = count;
    while (tempCount > splitPoint) {
        s1.push(s2.top());
        s2.pop();
        tempCount--;
    }

    // Step 4: Reverse s2 to restore the original order for the top half
    stack<int> tempStack;
    while (!s2.empty()) {
        tempStack.push(s2.top());
        s2.pop();
    }
    s2.swap(tempStack); // Now s2 is in the correct order
}

int main() {
```

Palindrome with stack



```
bool isPalindrome(const string& str) {
    stack<char> charStack;
    int n = str.length();

    // Push all characters of the string onto the stack
    for (int i = 0; i < n; i++) {
        charStack.push(str[i]);
    }

    // Pop characters from the stack and compare with the original string
    for (int i = 0; i < n; i++) {
        if (charStack.top() != str[i]) {
            return false; // Mismatch found, not a palindrome
        }
        charStack.pop();
    }

    return true; // No mismatches, it is a palindrome
}
```



| Source | Destination | | | | |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |
| | ∞ | ∞ | ∞ | ∞ | ∞ |
| | 7 | 9 | ∞ | ∞ | 14 |
| 1,2 | 7 | 9 | 22 | ∞ | 14 |
| 1,2,3 | 7 | 9 | 20 | ∞ | 11 |
| 1,2,3,6 | 7 | 9 | 20 | 20 | 11 |
| 1,2,3,6,4 | | | | 20 | |
| 1,2,3,6,4,5 | | | | | |