Lab 10: Stacks and Queues

1.1 Information

Topics: Stacks, Queues, Linked Lists

Turn in: All source files (.cpp and .hpp).

Starter files: Download on GitHub or D2L.

```
Lab 10 - Stacks and Queues/
 ├── lab10_main.cpp ...  Contains main()
 ├── lab10_DoublyLinkedList.hpp ...  Doubly Linked List
 │                                   template
 ├── CodeBlocks Project/ ...  Code::Blocks project here
 ├── Queue/
 │   └── Queue.hpp
 └── Stack/
     └── Stack.hpp
```

This program does not have any tests associated with it; you will have to manually test it or write your own tests.

Contents

1.2 Introduction

In this lab, you will create a Stack and a Queue in a few short lines of code. A DoublyLinkedList has already been implemented for you, and the Queue and Stack will each inherit from that class.

Using the functions from DoublyLinkedList, implement Stack and Queue's functions:

- Push
- Pop
- Get (either Top or Front)

main() is already implemented so that the user can select a queue or a stack and, using Polymorphism, it will dynamically create a new queue or stack and run those functions.

The implementation of each function will be one line each.

To explicitly call a function from the child class, you can prefix the function name with the parent class and template marker:

```
DoublyLinkedList<T>::FunctionThingy( blorp );
```

1.3 Queue

Remember that a Queue is a FIRST IN, FIRST OUT structure.

You will need to answer the following:

- When pushing a new item into a queue, where does the new item go? (front? back?)
- When popping an item out of a queue, which item is removed? (front? back?)
- When taking an item from a queue (without removing), which item is accesses? (front? back?)

1.4 Stack

Remember that a Stack is a FIRST IN, LAST OUT (or LAST IN, FIRST OUT) structure.

You will need to answer the following:

- When pushing a new item into a stack, where does the new item go? (front? back?)
- When popping an item out of a stack, which item is removed? (front? back?)
- When taking an item from a stack (without removing), which item is accesses? (front? back?)

1.5 Sample output

1.5.1 Queue

```
--------------------------------
1. Queue or 2. Stack, or anything else to quit.
>> 1

Create queue
Add A, B, C, D
Size: 4
Take: A

Current list:
A    FIRST
B
C
D    LAST

Pop 1
Size: 3
Take: B

Current list:
B    FIRST
C
D    LAST
```

1.5.2 Stack

```
--------------------------------
1. Queue or 2. Stack, or anything else to quit.
>> 2

Create stack
Add A, B, C, D
Size: 4
Take: D

Current list:
A    FIRST
B
C
D    LAST

Pop 1
Size: 3
Take: C

Current list:
A    FIRST
B
C    LAST
```

1.6 Appendix A: Starter code

lab10_main.cpp

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

#include "Queue/Queue.hpp"
#include "Stack/Stack.hpp"

int main()
{
    bool done = false;
    while ( !done )
    {
        List<string>* listObj = nullptr;

        cout << endl << "--------------------------------" << endl;
        cout << "1. Queue or 2. Stack, or anything else to quit." << endl << ">> ";
        int choice;
        cin >> choice;

        // POLYMORPHISM!!
        if ( choice == 1 )
        {
            cout << endl << "Create queue" << endl;
            // Initialize as queue
            listObj = new Queue<string>;
        }
        else if ( choice == 2 )
        {
            cout << endl << "Create stack" << endl;
            // Initialize as stack
            listObj = new Stack<string>;
        }
        else
        {
            break;
        }

        cout << "Add A, B, C, D" << endl;
        listObj->Push( "A" );
        listObj->Push( "B" );
        listObj->Push( "C" );
        listObj->Push( "D" );

        cout << "Size: " << listObj->GetSize() << endl;
        cout << "Take: " << listObj->Take() << endl;

        cout << endl << "Current list:" << endl;
        listObj->Display();

        cout << endl << "Pop 1" << endl;
        listObj->Pop();

        cout << "Size: " << listObj->GetSize() << endl;
        cout << "Take: " << listObj->Take() << endl;

        cout << endl << "Current list:" << endl;
        listObj->Display();

        if ( listObj != nullptr )
        {
            delete listObj; // free up memory
        }
    }

    return 0;
}
```

lab10_DoublyLinkedList

```cpp
#ifndef DOUBLYLINKEDLIST_HPP
#define DOUBLYLINKEDLIST_HPP

#include <stdexcept>
using namespace std;

template <typename T>
class Node
{
    public:
    Node()
    {
        m_ptrNext = nullptr;
        m_ptrPrev = nullptr;
    }

    T m_data;

    Node<T>* m_ptrNext;
    Node<T>* m_ptrPrev;
};

template <typename T>
class DoublyLinkedList
{
    public:
    DoublyLinkedList()
    {
        m_ptrFirst = nullptr;
        m_ptrLast = nullptr;
        m_itemCount = 0;
    }

    virtual ~DoublyLinkedList()
    {
        while ( m_ptrFirst != nullptr )
        {
            PopBack();
        }
    }

    /*
    Pure virtual functions:
    Interfaces for children, to be
    implemented by child classes.
    */
    virtual void Push( T data ) = 0;
    virtual void Pop() = 0;
    virtual T Take() = 0;

    /*
    Inherited public functions
    */

    void Display()
    {
        Node<T>* ptrCurrent = m_ptrFirst;
        while ( ptrCurrent != nullptr )
        {
            cout << ptrCurrent->m_data;

            if ( ptrCurrent == m_ptrFirst )
            {
                cout << "\t FIRST";
            }
            if ( ptrCurrent == m_ptrLast )
            {
                cout << "\t LAST";
            }

            cout << endl;

            ptrCurrent = ptrCurrent->m_ptrNext;
        }
    }

    int GetSize()
    {
        return m_itemCount;
    }

    protected:
    int m_itemCount;
    Node<T>* m_ptrFirst;
    Node<T>* m_ptrLast;

    /*
    Behind-the-scenes inner-workings
    */

    void PushFront( T data )
    {
        Node<T>* newNode = new Node<T>();
        newNode->m_data = data;

        if ( m_ptrFirst = nullptr )
        {
            // Empty list
            m_ptrFirst = newNode;
            m_ptrLast = newNode;
        }
        else
        {
            // Not empty, new node is the new first
            newNode->m_ptrNext = m_ptrFirst;
            m_ptrFirst->m_ptrPrev = newNode;

            // Update pointer
            m_ptrFirst = newNode;
        }

        m_itemCount++;
    }

    void PushBack( T data )
    {
        Node<T>* newNode = new Node<T>();
        newNode->m_data = data;

        if ( m_ptrFirst == nullptr )
        {
            // Empty list
            m_ptrFirst = newNode;
            m_ptrLast = newNode;
        }
        else
        {
            // Not empty, new node is the new last
            m_ptrLast->m_ptrNext = newNode;
            newNode->m_ptrPrev = m_ptrLast;

            // Update pointer
            m_ptrLast = newNode;
        }

        m_itemCount++;
    }

    void Insert( T data, int index )
    {
        if ( index > m_itemCount || index < 0 )
        {
            throw out_of_range( "Invalid index!" );
        }

        Node<T>* newNode = new Node<T>();
        newNode->m_data = data;

        if ( m_ptrFirst = nullptr )
        {
            // Empty list
            m_ptrFirst = newNode;
            m_ptrLast = newNode;
        }
        else
        {
            // Traverse list to find position
            int counter = 0;
            Node<T>* ptrCurrent = m_ptrFirst;

            while ( counter != index )
            {
                counter++;
            }

            // Add item in list
            newNode->m_ptrPrev = ptrCurrent->m_ptrPrev;
            newNode->m_ptrNext = ptrCurrent;

            ptrCurrent->m_ptrPrev->m_ptrNext = newNode;
            ptrCurrent->m_ptrPrev = newNode;
        }

        m_itemCount++;
    }


    void PopFront()
    {
        if ( m_ptrFirst == nullptr )
        {
            return;
        }

        if ( m_ptrFirst == m_ptrLast )
        {
            delete m_ptrLast;
            m_ptrFirst = nullptr;
            m_ptrLast = nullptr;
            m_itemCount--;
            return;
        }

        // Keep track of 2nd item
        Node<T>* ptrSecond = m_ptrFirst->m_ptrNext;

        // Update the 2nd element's prev pointer
        ptrSecond->m_ptrPrev = nullptr;

        // Clear out the data at m_ptrFirst
        delete m_ptrFirst;

        // Update first pointer
        m_ptrFirst = ptrSecond;

        m_itemCount--;
    }

    void PopBack()
    {
        if ( m_ptrLast == nullptr )
        {
            return;
        }

        if ( m_ptrFirst == m_ptrLast )
        {
            delete m_ptrLast;
            m_ptrFirst = nullptr;
            m_ptrLast = nullptr;
            m_itemCount--;
            return;
        }

        // Keep track of 2nd-to-last item
        Node<T>* ptrPenultimate = m_ptrLast->m_ptrPrev;

        // Update 2nd-to-last item’s next ptr
        ptrPenultimate->m_ptrNext = nullptr;

        // Clear out data at last element
        delete m_ptrLast;

        // Update last pointer
        m_ptrLast = ptrPenultimate;

        m_itemCount--;
    }

    void Remove( int index )
    {
        if ( index > m_itemCount || index < 0 )
        {
            throw out_of_range( "Invalid index!" );
        }

        // Locate item
        Node<T>* ptrCurrent = m_ptrFirst;

        int counter = 0;
        while ( counter != index )
        {
            counter++;
        }

        Node<T>* ptrPrev = ptrCurrent->m_ptrPrev;
        Node<T>* ptrNext = ptrCurrent->m_ptrNext;

        // Update previous item's pointer
        ptrPrev->m_ptrNext = ptrNext;

        // Update next item's pointer
        ptrNext->m_ptrPrev = ptrPrev;

        // Free this item
        delete ptrCurrent;

        m_itemCount--;
    }

    T& GetFront()
    {
        if ( m_ptrFirst == nullptr )
        {
            throw out_of_range( "First pointer is nullptr" );
        }

        return m_ptrFirst->m_data;
    }

    T& GetBack()
    {
        if ( m_ptrLast == nullptr )
        {
            throw out_of_range( "Last pointer is nullptr" );
        }

        return m_ptrLast->m_data;
    }

    T& Get( int index )
    {
        if ( index > m_itemCount || index < 0 )
        {
            throw out_of_range( "Invalid index!" );
        }

        // Locate item
        Node<T>* ptrCurrent = m_ptrFirst;

        int counter = 0;
        while ( counter != index )
        {
            ptrCurrent = ptrCurrent->m_ptrNext;
            counter++;
        }

        return ptrCurrent->m_data;
    }
};

template <typename T>
using List = DoublyLinkedList<T>;

#endif
```

Queue/Queue.hpp

```
#ifndef QUEUE_HPP
#define QUEUE_HPP

#include "../DoublyLinkedList.hpp"

template <typename T>
class Queue : public DoublyLinkedList<T>
{
    public:
    Queue()
        : DoublyLinkedList<T>()
    {
    }

    virtual ~Queue()
    {
    }

    virtual void Push( T data )
    {
        DoublyLinkedList<T>::PushBack( data );
    }

    virtual void Pop()
    {
        DoublyLinkedList<T>::PopFront();
    }

    virtual T Take()
    {
        return DoublyLinkedList<T>::GetFront();
    }
};

#endif
```

Stack/Stack.hpp

```
#ifndef STACK_HPP
#define STACK_HPP

#include "../DoublyLinkedList.hpp"

template <typename T>
class Stack : public DoublyLinkedList<T>
{
    public:
    Stack()
        : DoublyLinkedList<T>()
    {
    }

    virtual ~Stack()
    {
    }

    virtual void Push( T data )
    {
        DoublyLinkedList<T>::PushBack( data );
    }

    virtual void Pop()
    {
        DoublyLinkedList<T>::PopBack();
    }

    virtual T Take()
    {
        return DoublyLinkedList<T>::GetBack();
    }
};

#endif
```