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:

```
1 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

```
#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;
        }

```

```
38          cout << "Add A, B, C, D" << endl;
39          listObj->Push( "A" );
40          listObj->Push( "B" );
41          listObj->Push( "C" );
42          listObj->Push( "D" );
43
44          cout << "Size: " << listObj->GetSize() << endl;
45          cout << "Take: " << listObj->Take() << endl;
46
47          cout << endl << "Current list:" << endl;
48          listObj->Display();
49
50          cout << endl << "Pop 1" << endl;
51          listObj->Pop();
52
53          cout << "Size: " << listObj->GetSize() << endl;
54          cout << "Take: " << listObj->Take() << endl;
55
56          cout << endl << "Current list:" << endl;
57          listObj->Display();
58
59          if ( listObj != nullptr )
60          {
61              delete listObj; // free up memory
62          }
63      }
64
65      return 0;
66  }
```

lab10_DoublyLinkedList

```
#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();
        }
    }

```

```
42      /*
43      Pure virtual functions:
44      Interfaces for children, to be
45      implemented by child classes.
46      */
47      virtual void Push( T data ) = 0;
48      virtual void Pop() = 0;
49      virtual T Take() = 0;
50
51      /*
52      Inherited public functions
53      */
54
55      void Display()
56      {
57          Node<T>* ptrCurrent = m_ptrFirst;
58          while ( ptrCurrent != nullptr )
59          {
60              cout << ptrCurrent->m_data;
61
62              if ( ptrCurrent == m_ptrFirst )
63              {
64                  cout << "\t FIRST";
65              }
66              if ( ptrCurrent == m_ptrLast )
67              {
68                  cout << "\t LAST";
69              }
70
71              cout << endl;
72
73              ptrCurrent = ptrCurrent->m_ptrNext;
74          }
75      }
76
77      int GetSize()
78      {
79          return m_itemCount;
80      }
81
82      protected:
83      int m_itemCount;
84      Node<T>* m_ptrFirst;
```

```
85      Node<T>* m_ptrLast;
86
87      /*
88      Behind-the-scenes inner-workings
89      */
90
91      void PushFront( T data )
92      {
93          Node<T>* newNode = new Node<T>();
94          newNode->m_data = data;
95
96          if ( m_ptrFirst = nullptr )
97          {
98              // Empty list
99              m_ptrFirst = newNode;
100             m_ptrLast = newNode;
101         }
102         else
103         {
104             // Not empty, new node is the new first
105             newNode->m_ptrNext = m_ptrFirst;
106             m_ptrFirst->m_ptrPrev = newNode;
107
108             // Update pointer
109             m_ptrFirst = newNode;
110         }
111
112         m_itemCount++;
113     }
114
115     void PushBack( T data )
116     {
117         Node<T>* newNode = new Node<T>();
118         newNode->m_data = data;
119
120         if ( m_ptrFirst == nullptr )
121         {
122             // Empty list
123             m_ptrFirst = newNode;
124             m_ptrLast = newNode;
125         }
126         else
127         {
```

```
128             // Not empty, new node is the new last
129             m_ptrLast->m_ptrNext = newNode;
130             newNode->m_ptrPrev = m_ptrLast;
131 
132             // Update pointer
133             m_ptrLast = newNode;
134         }
135 
136         m_itemCount++;
137     }
138 
139     void Insert( T data, int index )
140     {
141         if ( index > m_itemCount || index < 0 )
142         {
143             throw out_of_range( "Invalid index!" );
144         }
145 
146         Node<T>* newNode = new Node<T>();
147         newNode->m_data = data;
148 
149         if ( m_ptrFirst = nullptr )
150         {
151             // Empty list
152             m_ptrFirst = newNode;
153             m_ptrLast = newNode;
154         }
155         else
156         {
157             // Traverse list to find position
158             int counter = 0;
159             Node<T>* ptrCurrent = m_ptrFirst;
160 
161             while ( counter != index )
162             {
163                 counter++;
164             }
165 
166             // Add item in list
167             newNode->m_ptrPrev = ptrCurrent->m_ptrPrev;
168             newNode->m_ptrNext = ptrCurrent;
169 
170             ptrCurrent->m_ptrPrev->m_ptrNext = newNode;
```

```
171             ptrCurrent->m_ptrPrev = newNode;
172         }
173 
174         m_itemCount++;
175     }
176 
177 
178     void PopFront()
179     {
180         if ( m_ptrFirst == nullptr )
181         {
182             return;
183         }
184 
185         if ( m_ptrFirst == m_ptrLast )
186         {
187             delete m_ptrLast;
188             m_ptrFirst = nullptr;
189             m_ptrLast = nullptr;
190             m_itemCount--;
191             return;
192         }
193 
194         // Keep track of 2nd item
195         Node<T>* ptrSecond = m_ptrFirst->m_ptrNext;
196 
197         // Update the 2nd element's prev pointer
198         ptrSecond->m_ptrPrev = nullptr;
199 
200         // Clear out the data at m_ptrFirst
201         delete m_ptrFirst;
202 
203         // Update first pointer
204         m_ptrFirst = ptrSecond;
205 
206         m_itemCount--;
207     }
208 
209     void PopBack()
210     {
211         if ( m_ptrLast == nullptr )
212         {
213             return;
```

```
214         }
215
216         if ( m_ptrFirst == m_ptrLast )
217         {
218             delete m_ptrLast;
219             m_ptrFirst = nullptr;
220             m_ptrLast = nullptr;
221             m_itemCount--;
222             return;
223         }
224
225         // Keep track of 2nd-to-last item
226         Node<T>* ptrPenultimate = m_ptrLast->m_ptrPrev;
227
228         // Update 2nd-to-last item's next ptr
229         ptrPenultimate->m_ptrNext = nullptr;
230
231         // Clear out data at last element
232         delete m_ptrLast;
233
234         // Update last pointer
235         m_ptrLast = ptrPenultimate;
236
237         m_itemCount--;
238     }
239
240     void Remove( int index )
241     {
242         if ( index > m_itemCount || index < 0 )
243         {
244             throw out_of_range( "Invalid index!" );
245         }
246
247         // Locate item
248         Node<T>* ptrCurrent = m_ptrFirst;
249
250         int counter = 0;
251         while ( counter != index )
252         {
253             counter++;
254         }
255
256         Node<T>* ptrPrev = ptrCurrent->m_ptrPrev;
```

```
257         Node<T>* ptrNext = ptrCurrent->m_ptrNext;
258
259         // Update previous item's pointer
260         ptrPrev->m_ptrNext = ptrNext;
261
262         // Update next item's pointer
263         ptrNext->m_ptrPrev = ptrPrev;
264
265         // Free this item
266         delete ptrCurrent;
267
268         m_itemCount--;
269     }
270
271     T& GetFront()
272     {
273         if ( m_ptrFirst == nullptr )
274         {
275             throw out_of_range( "First pointer is
    nullptr" );
276         }
277
278         return m_ptrFirst->m_data;
279     }
280
281     T& GetBack()
282     {
283         if ( m_ptrLast == nullptr )
284         {
285             throw out_of_range( "Last pointer is nullptr
    " );
286         }
287
288         return m_ptrLast->m_data;
289     }
290
291     T& Get( int index )
292     {
293         if ( index > m_itemCount || index < 0 )
294         {
295             throw out_of_range( "Invalid index!" );
296         }
297
```

```
298         // Locate item
299         Node<T>* ptrCurrent = m_ptrFirst;
300 
301         int counter = 0;
302         while ( counter != index )
303         {
304             ptrCurrent = ptrCurrent->m_ptrNext;
305             counter++;
306         }
307 
308         return ptrCurrent->m_data;
309     }
310 };
311 
312 template <typename T>
313 using List = DoublyLinkedList<T>;
314 
315 #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
```