VE 280 Lab 10

Out: 00:01 am, July 20, 2020; **Due**: 11:59 pm, July 28, 2020.

Specifications

Ex1. Designing a Templated Stack

Related topics: *stack*, *template*, *linked list*, *deep copy*

For `lab10`, you will be implementing a templated list-based stack.

The data structure `Stack` is given in `my_stack.h`:

```
class stackEmpty
// Overview: An exception class.
{

};


template <class T>
struct Node
// Overview: Node.
{
    Node* next;
    T val;
};


template <class T>
class Stack
// Overview: A list based stack.
{

private:
    /* Attributes */
    Node<T>* head;

    /* Utilities */
    void removeAll();
    // EFFECTS: called by destructor/operator=
    //          to remove and destroy all list elements.

    void copyFrom(const Stack &s);
    // MODIFIES: this
    // EFFECTS: called by copy constructor/operator=
    //          to copy elements from a source list l to this list;
    //          if this list is not empty originally,
    //          removes all elements from it before copying.

public:
```

```
39
40      Stack();
41      // constructor
42      Stack(const Stack &s);
43      // copy constructor
44      Stack &operator = (const Stack &s);
45      // assignment operator
46      ~Stack();
47      // destructor
48
49
50      /* Methods */
51      void print();
52      // EFFECTS: print the elements in the stack
53
54      bool isEmpty() const;
55      // EFFECTS: returns true if list is empty, false otherwise.
56
57      size_t size() const;
58      // EFFECTS: returns the size of the stack.
59
60      void push(T val);
61      // MODIFIES: this
62      // EFFECTS: inserts val at the top of the stack.
63
64      void pop();
65      // MODIFIES: this
66      // EFFECTS: removes the top element from a non-empty stack;
67      //          in case of empty stack, throws an instance of emptyList if
    empty.
68
69      T top() const;
70      // EFFECTS: returns the top element from a stack.
71      //          in case of empty stack, throws an instance of emptyList if
    empty.
72
73  };
```

Recall that your `List` in `lab9` contains both the pointer `first` that points to the first node, and the `last` pointer that points to the last node, because this linked list is first-in-first-out (FIFO), *i.e.* support `insertBack` and `removeFront` from both directions.

However, a stack is first-in-last-out (FILO), *i.e.* support `push` and `pop` from only one direction. Thus, it contains only a `head` pointer.

Similarly, dynamically allocation is required, and you must also be careful about the big-3.

Again, `print` is already implemented in `my_stack_impl.h`, which prints elements in the stack in order.
Please do not modify it.

You are left with much freedom in adding helper functions, even attributes to the header, yet **they must be private**. You may find adding an additional attribute maintaining the size of stack useful. Also, you may find Lecture 21 demo useful.

Ex2. Reversing a Stack

Related topics: *stack*

The first-in-last-out (FILO) property of stacks makes it interesting to reverse a stack, and there are many known algorithms to reversing a stack.

You are asked to implement the `reverse` function of the stack data type, as is described in `my_stack.h`:

```
template <class T>
void reverse(Stack<T> &s);
// MODIFIES: s
// EFFECTS: reverse stack s.
//           * for example:
//             [12345] => [54321]
```

Ex3. Appending a Stack

Related topics: *stack*, *overload*

In many circumstances, the operator `+` is interpreted as appending.

Appending to the bottom of stack could be useful, but not at all trivial. Therefore, you will be now overloading the `+` operator to support bottom appending for both a single value or another stack, as is described in `my_stack.h`:

```
/* Operators */
template <class T>
Stack<T> operator +(Stack<T> &s, T val);
// EFFECTS: returns a new stack which is the result of appending stack s by
val.
//          for example:
//             [123] + 4 => [1234]

template <class T>
Stack<T> operator +(Stack<T> &first, Stack<T> &second);
// EFFECTS: returns a new stack which is the result of appending stack first
//          by another stack second.
//          for example:
//             [123] + [45] => [12345]
```

Note that you need to return a new `Stack` as a result of appending, and the ingredient stacks need to stay invariant.

Testing

A `test.cpp` is provided in the start files, which tested your `reverse` and the `+` operator.

The correct output should be

```
12345678
87654321
```

Submission

`my_stack.h` and `my_stack_impl.h` could be found in `lab10_starter_files` on canvas. Please implement the stack methods and another 3 functions in `my_stack_impl.h`. Submit both `my_stack.h` and `my_stack_impl.h` as a `.tar` file via online judgement system.

Please use `valgrind` to check and make sure there is no memory leak.

Challenges

The following challenges are just for fun. They are by no means tested or required by the online judge. Students who have an interest in further study in computer science may have a look and challenge yourself.

1. Swap two stacks.

   Hint: See stack swap() in C++ STL.

2. Implement the cloning method `copyFrom` without extra memory.

   Hint: See Clone a stack without extra space.

3. Rewrite Ex2 and Ex3 with no loop structure (recursive).

   Hint: See Reverse a stack using recursion.

4. Sort a stack, assuming that your type `T` is sortable, *i.e.* `int`.

   Hint: See Sort a stack using a temporary stack.

Created by Martin Ma.

Last update: July 19, 2020