CMPT 225 D2
Fall 2020
T.Shermer

Assignment 2
Stacks (with Templates and Exception)

Due Oct 9 at 23:59

You are to write a generic (templated) C++ Stack class, along with some other code mainly for testing the Stack implementation.

The principal class will be called **Stack,** and it should be templated on the type of the stack. (E.g. Stack<int> or Stack<string>.)

Stack has the following member functions (T is the template parameter).

type	name	
T&	pop()	// returns and removes the top of the stack. // (Or throws an EmptyStackException if the stack is empty)
void	push(T&)	// pushes the argument onto the stack.
bool	empty()	// returns true if the stack is empty.
int	size()	// returns the number of elements in the stack.
int	capacity()	// returns the current capacity of the stack.

It should also have a constructor and a destructor. As a constructor argument, it should take a capacity, which is the size of the array that it makes to hold the stack elements. This argument should default to 4. If there is a push() operation and the stack is at capacity (i.e. it's full), then make a new array that has twice the size of the old capacity and copy the elements from the old array to the new array. Delete the old array.

EmptyStackException is a new class that you should create. It can be created in the same file as the Stack class.

To exercise your stack functionality, create a class called ArrayUtils, and give it a single member function, reverse(), which is **static**. Its arguments should be an array and its size. It should be templated so as to be able to take any array. Reverse should reverse the array that it is passed by (1) making a stack of that type, (2) pushing each element of the array on the stack, and (3) popping each element of the array from the stack, placing it in the correct place in the array. It should return its array argument.

Your main routine should call several test functions. These are: testStackUnderflow(), testStackGrowth(), testReverseIntegers(), testReverseStrings(), and testReverseEmployees().

testStackUnderflow() will check to see if your exception is thrown when a stack underflows. In it, you should use a try-catch statement that catches EmptyStackException. Inside the try block, create a Stack, push an element onto it, pop it twice, and then print "did not catch exception". The second pop should cause an exception; if this is properly done, then the line that prints "did not catch exception" should not execute. Inside the catch block, print "caught EmptyStackException".

testStackGrowth() will check to see that your stack grows as appropriate. In it, create a stack with capacity 4. Push items onto it, printing them as you push. After 5 items, print the capacity of the stack. Keep pushing and printing items until you get 9 items, and print the capacity of the stack again. Then pop all of the elements, printing them as you do so.

The remainder of the test functions are relatively the same and are for testing out that you did the templating correctly. In testReverseIntegers(), make an array of 12 integers, print it, call reverse (in ArrayUtils) on it, and print it again. In testReverseStrings(), make an array of 12 strings, print it, call reverse on it, and print it again. In testReverseEmployees(), make an array of 12 Employees, print it, call reverse on it, and print it again.

You'll have to make a class Employee for the last test. This class should have two constructor arguments, which are a name and an employee ID number. The class should be immutable with getters for the name and ID number. It should also have a function toString() which returns a string that is the name followed by a space followed by the ID number. Use this function in printing an Employee in testReverseEmployees().

Print a blank line between each pair of tests in main.

All classes should be in separate files (.cpp and/or .h) named with the class name. (The exception class is an exception to this.)

You will be judged on correctness of your code and on code style, so don't forget to keep your code clean as you develop it! (Or at the very least, clean it up before submission. We don't want to see untidy code.)

note: Tests that print results, like the ones above, are in general discouraged in software engineering. Why? Because we do so many tests of our classes and functions that a human quickly becomes fatigued if they have to look at all the test output. The tests here are nowhere near the amount of tests that one would in practice write to ensure the correct operation of Stack. Tests are now normally made to be silent (no printing) if they succeed, and only print something if they fail. If we were to do this for, say, testReverseIntegers(), we'd have to test that the array we got back from reverse() contained the reverse elements from what we sent in to the function. (We'd check to make sure that each element of the resulting array was correct.) Do **not** do this for this assignment. We will expect your code to print results as described above.

more fun (not required)

If you finish the above and want to have more fun (but no extra credit), implement a templated StackDuo class. This class contains two stacks A and B of the templated type. It has operations *push**,*** which pushes an item onto stack A; *transfer*, which pops an item from stack A and puts it on stack B; *pop*, which pops and returns an item from stack B; *size*, which is the sum of the number of elements in the two stacks; and *sizeA* and *size**B***, which return the number of elements in stack A or stack B respectively. Figure out their argument and return types.

Now add a function to main which tests the StackDuo. You may have to do several tests, but try to make the results easy to interpret. You may want to test its exception-throwing behaviour as well.

Do not turn in your StackDuo code. We will not look at it. You can ensure that it is correct by testing it.

(advanced!) As a theoretical exercise, consider a sequence of n pushes, n transfers, and n pops, with these operations intermingled. Suppose the pushes are of consecutive integers 1, 2, 3, …, n. Record the number sequence that you get from the pops. Can you get every permutation of 1, 2, 3, …, n out of the StackDuo by some sequence of operations? If not, which permutations can you get?

For instance, suppose n is 4 and we want the permutation 1423.

We could:

```
push(1)
transfer()
pop()                   // 1 is popped.
push(2)
transfer()                      // leave 2 on stack B
push(3)                         // leave 3 on stack A
push(4)
transfer()
pop()                   // 4 is popped.
pop()                   // 2 is popped.
transfer()                      // 3 is moved from A to B
pop()                   // 3 is popped.
```