CMPT 115: Principles of Computer Science

Array-Based Stack ADT

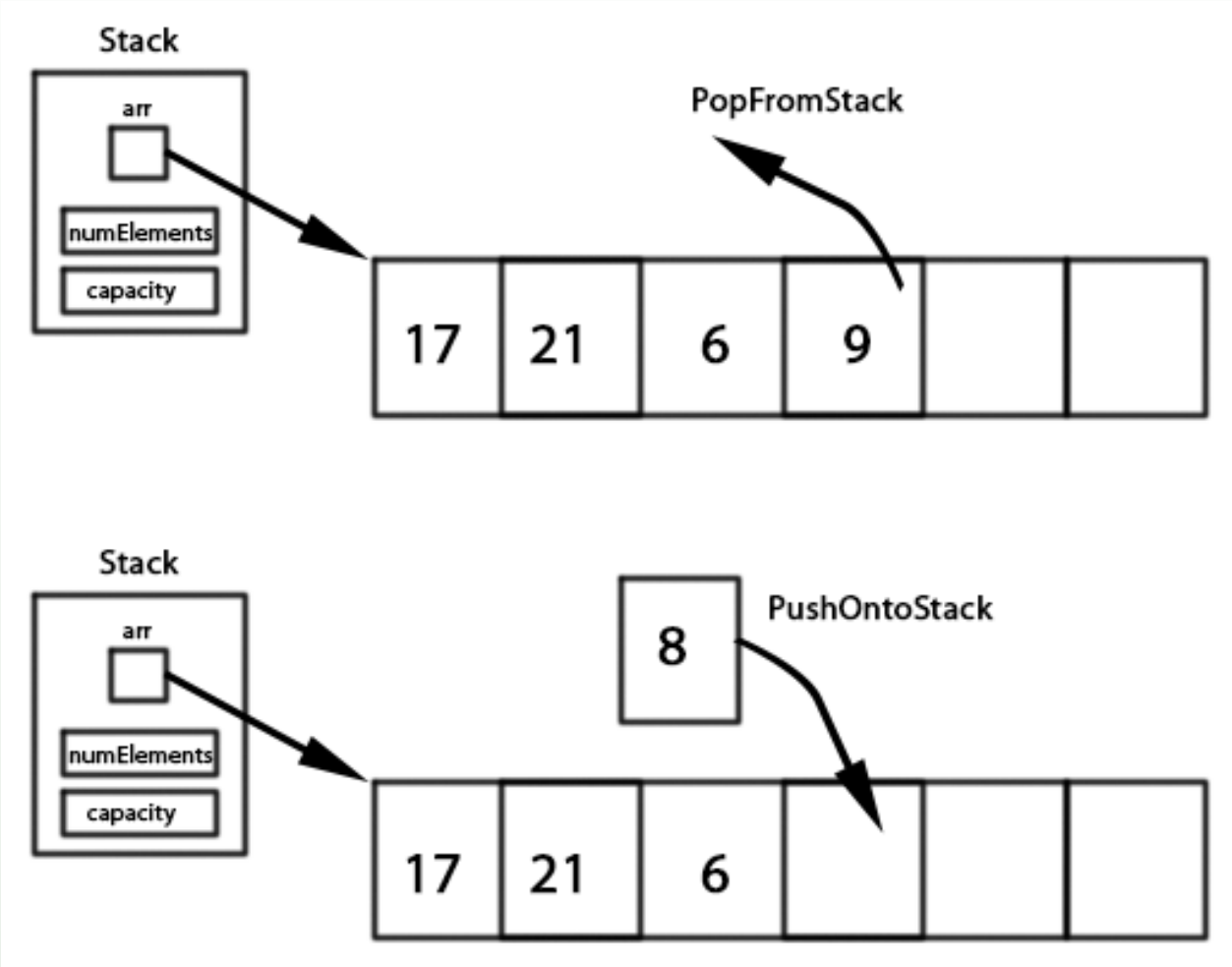
Department of Computer Science
University of Saskatchewan

March 7-11, 2016

Laboratory 7 Overview

1. To implement an array-based stack ADT in C++.
2. Exercises (to hand in with Assignment 7)
   - File: `Stack.cpp` containing the 8 operations.

Array-based Stack



To use an array as a stack, we have to add and remove elements only from the end or TAIL of the array.

Question: Why do we add new elements on the tail of the array? Why not the HEAD of the array?

Lab objective: build and test an array-based stack

In this lab you'll write a program that does the following:

1. Create a new stack.
2. Repeatedly:
   - ask the user for an integer,
   - put the integer on the stack

   Keep repeating until the user types -999, or the stack capacity is reached.
3. Display the following:
   - the last integer the user entered,
   - the number of integers entered,
   - all the integers entered, but in the reverse of the order they were entered.
4. Destroy the stack.

Program Structure

Your program will consist of four files:

- `Stack.h` is the header file that contains
  - Function prototypes
  - The Stack record type definition ("struct")
  - A typedef for Element (for this lab: `typedef int Element;`).
- `Stack.cc` will contain Stack ADT operation implementations.
- `testStack.cc` will be the driver program that will test the functions we create in the Stack ADT.
- Note: `testStack.cc` can ONLY use the stack ADT operations

Iterative Development (1)

Start with the following files:

1. `Stack.h` contains function headers from a design document (as comments). You can find `Stack.h` on the Moodle page.
2. `Stack.cpp` contains the operators defined as "do-nothing" functions. You can find `Stack.cpp` on the Moodle page.
3. Create a `testStack.cc` file with a main() that displays something (anything) but does not actual work yet. It should `#include "Stack.h"` and `#include <iostream>`, etc. Also your name, etc.
4. Make sure those all compile correctly using
   `g++ -Wall -pedantic testStack.cc Stack.cc`

Iterative Development (2)

Then continue by implementing each of the Stack ADT operations one at a time. Always compile your program and test what you've written before moving on to the next one!

1. Using the pseudocode on slide 9, implement CreateStack in `Stack.cc`. You are replacing the do-nothing stub with a real function. Compile. Fix your code if it doesn't compile.
2. Modify `testStack.cc` to create a stack (but nothing else). Compile and run!
3. Repeat the process for every Stack ADT operation: implement the operation in `Stack.cc`, add some test code to `testStack.cc`, compile and run. Debug if necessary.

It might help if you have all three files open on your desktop at the same time, so you can see them all.

Development time

You should aim for 10 minutes per operation:

- Create function prototype in `Stack.h` (starting from the given function header)
- Implement function in `Stack.cc` (starting from given pseudocode)
- Write a few lines of code to test the new function in `testStack.h`
- You might need a few more minutes if you need to debug your operation!
- You may increase the time to complete this work geometrically if you try to write all the operations all at once. Do them one at a time, and test each one before going on.

CreateStack

ACTIVITY: In `Stack.cc`, implement the CreateStack function.

Algorithm *CreateStack*(cap)
`Creates an array-based stack in dynamic memory`
Pre: cap is an integer constant indicating the size of the array to be created
Post: A stack will be created in dynamic memory
Return: A pointer to the newly created stack

```
RefToStack newStack <- allocate new Stack
newStack->numElements <- 0
newStack->arr <- allocate new Element [cap]

if(newStack->arr == NULL)
 newStack->capacity <- 0
else
 newStack->capacity <- cap
endif

return newStack
```

DestroyStack

ACTIVITY: In `Stack.cc`, implement the DestroyStack function.

Algorithm *DestroyStack*(stack)
`Deletes the stack from memory`
Pre: stack is a reference to a stack in memory
Post: A stack will be removed from dynamic memory
Return: none

deallocate stack->arr
deallocate stack

ReadFromStack

ACTIVITY: In `Stack.cc`, implement the ReadFromStack function.

Algorithm *ReadFromStack*(stack, el)
`Reads the element from the top of the stack and stores it in el`
Pre: stack is a reference to a stack in memory, el is a reference to an Element
Post: el will be updated to contain the appropriate element found in the stack
Return: true if an element was retrieved, false if it was not

```
if(stack->numElements > 0)
 *el <- stack->arr[stack->numElements-1]
 return true
else
 return false
endif
```

PrintStack

ACTIVITY: In `Stack.cc`, implement the PrintStack function.

Algorithm *PrintStack*(stack)
`Prints the entire stack`
Pre: stack is a reference to a stack in memory
Post: prints out the stack to console
Return: none

```
if(stack->numElements != 0)
 for i from stack->numElements -1 to 0
  print stack->arr[i]
 endfor
else
 print "stack is empty"
endif
```

Exercise

1. Implement the following four functions:
   - StackIsEmpty: checks to see if the stack is empty
   - StackCount: returns the number of elements in the stack
   - PushOntoStack: see the next slide for algorithm header
   - PopFromStack: see the slide after that for the algorithm header
2. Modify your testStack.cc file to test those functions
3. Compile all those files together using the following

```
g++ -o stack -Wall -pendantic testStack.cc Stack.cc
```

4. Only submit the file `Stack.cc`.

PushOntoStack

Algorithm *PushOntoStack*(stack, el)
`Pushes a copy of the element el onto the stack`
Pre: stack is a reference to a stack in memory
el is an Element (not a reference to an element)
Post: stack is updated to contain a new element
Return: true if el was added successfully, false if the stack was full

Notes: First check to see if the stack is full,
then copy the element to the end of the stack,
and increment the number of elements

PopFromStack

Algorithm *PopFromStack*(stack, el)
`Pushes a copy of the element el onto the stack`
Pre: stack is a reference to a stack in memory, el is a reference to an Element
Post: stack is updated to contain one less element.
element contains the value that was removed
Return: true if el was removed successfully, false if the stack was empty

Notes: First check to see if the stack is empty,
then copy the element from the end of the stack,
and decrement the number of elements