Datastructuren opdracht № 1

Lisa Pothoven, David Kleingeld September 21, 2016

Stapel

Questions:

- do we need to keep an eye on integer overflow in arraystack?
- are the comments in the right place?
- wat word bedoelt met constante functies etc is die vorm goed?
- is de output zo goed? we gebruiken std::cout
- kunnen jullie er even naar kijken of het ongeveer oke is?
- nog feedback die meegenomen kan worden voor de volgende opdracht

Main file that calles the other files

```cpp
#include <iostream>
// Visual Studio compiler does not directly import <string>
#include <string>
#include "PointerStack.h"
#include "ArrayStack.h"
#include "StlStack.h"
#include "VectorStack.h"

using namespace std;

template<class T>
void backspace(string input)
{
    T invoerstack;

    char topItem = 0;
    int j = 0;
    // Backspace simulation: When an * is encountered, the previous character will be deleted.
    while (input[j] != 0 && input[j] != ' ') {
        if (input[j] == '*') {
            invoerstack.pop(); // Delete char
            j = j + 1;
        }
        else {
            invoerstack.push(input[j]);
            j = j + 1;
        }
    }

    // Display top item
    invoerstack.top(topItem);
    cout << "Top item: " << topItem << "\n";
    // Pop top item
    invoerstack.pop();
    cout << "Pop" << endl;

    // Display the content of the stack, in the case of STL stack this clears the stack (so, only do this at the end).
    cout << "Stack reads: ";
    invoerstack.read();
    cout << "\n";

    // Check if stack is empty
    invoerstack.clear();
```

```
44      if (invoerstack.empty()) {
45          cout << "Cleanup succeeded.\n" << endl;
46      }
47      else {
48          cout << "Unable to clear ArrayStack.\n" << endl;
49      }
50  }
51
52  int main()
53  {
54      cout << "\nDatastructures" << endl << "Assignment 1: Stacks" << "\n\n"↩
            ;
55      /*
56      // Read a "word", untill the first break.
57      cout << "Give me input to stack: ";
58      string invoer;
59      getline(cin, invoer);
60      */
61      string input = "helll*oww*orld";
62      cout << "Initial input is: " << input << "\n\n";
63
64      // Run the same test for each of the four implementations.
65      ArrayStack<char> arrayStack;
66      cout << "ArrayStack:\n";
67      backspace<ArrayStack<char> >(input);
68      PointerStack<char> pointerStack;
69      cout << "PointerStack:\n";
70      backspace<PointerStack<char> >(input);
71      cout << "STL Stack:\n";
72      StlStack<char> stlstack;
73      backspace<StlStack<char> >(input);
74      cout << "VectorStack:\n";
75      VectorStack<char> vectorStack;
76      backspace<VectorStack<char> >(input);
77      return 0;
78  }
```

```
1  /**
2  * Class name: ArrayStack, classical array stack
3  * @author Lisa Pothoven (s1328263)
4  * @author David Kleingeld (s1432982)
5  * @file arraystack.h
6  * @date 08-09-2016
7  **/
8
9  #ifndef ArrayStack_h
10 #define ArrayStack_h
11
12 #define MAXSIZE 40
13
14 template <class T>
15 class ArrayStack {
16 public:
17     ArrayStack();//TODO shouldnt we pass the MaxSize to the constructor?
18     bool empty();//as an agument
19     void clear();
20     void push(T newItem);
21     bool pop();
22     bool top(T & topItem);
23     void read();
24     void size();
25 private:
26     // Array of certain size
27     T array[MAXSIZE];
28     // Variable to keep track of the top of the array
29     int top_number;
30 };
31
32 /**
33 * @function ArrayStack()
34 * @abstract Constructor: Create object ArrayStack, filled with 0s
35 * @param MaxSize: the size of the array
36 * @return
37 * @pre
38 * @post empty ArrayStack object
39 **/
40 template <class T>
41 ArrayStack<T>::ArrayStack() {
42     for (int m = 0; m < MAXSIZE; m++) {
43         array[m] = 0;
44     }
45 }
```

```
46
47  /**
48  * @function empty()
49  * @abstract Check if object ArrayStack is empty
50  * @param none
51  * @return true (is empty) or false (is not empty)
52  * @pre
53  * @post
54  **/
55  template <class T>
56  bool ArrayStack<T>::empty() {
57      if (array[0] == 0) {
58          return true;
59      }
60      return false;
61  }
62
63
64  /**
65  * @function clear()
66  * @abstract Clear content of array stack, delete array
67  * @param none
68  * @return true (succes) or false (something did not work)
69  * @pre Filled stack of type array
70  * @post Empty stack
71  **/
72  template <class T>
73  void ArrayStack<T>::clear() {
74      while(!empty() ) {
75          pop();
76      }
77  }
78
79  /**
80  * @function push(newItem)
81  * @abstract Add newItem to top of the stack
82  * @param newItem: new item to be added to the stack
83  * @return true (succes) or false (something did not work)
84  * @pre Stack of type array
85  * @post Stack of type array, with new item at top
86  **/
87  template <class T>
88  void ArrayStack<T>::push(T newItem) {
89      if (empty()) {
90          array[0] = newItem;
91      }
92      else {
```

```
93          size(); // Find top_number
94          if (top_number + 1 != MAXSIZE) {
95              array[top_number + 1] = newItem;
96          }
97          else {
98              std::cout << "Too much input\n";
99          }
100     }
101 }
102 
103 /**
104 * @function pop()
105 * @abstract Remove item from top of the stack
106 * @param none
107 * @return true (succes) or false (something did not work)
108 * @pre Stack of type array
109 * @post Stack of type array, with top item removed
110 **/
111 template <class T>
112 bool ArrayStack<T>::pop() {
113     if (!empty())
114     {
115         size();
116         array[top_number] = 0; // Pop
117         top_number = top_number - 1;
118         return true;
119     }
120     else // Array is empty
121     {
122         return false;
123     }
124 }
125 
126 /**
127 * @function top(topItem)
128 * @abstract Return item at the top of the stack, give output to user
129 * @param topItem: the item at the top of the stack
130 * @return true (succes) or false (something did not work)
131 * @pre Stack of type array
132 * @post Stack of type array
133 **/
134 template <class T>//TODO maybe a var that keeps the top element?
135 bool ArrayStack<T>::top(T & topItem) {
136     if (!empty()) {
137         size();
138         topItem = array[top_number];
139         return true;
```

```
140     }
141     else {
142         return false;
143     }
144 }
145 
146 /**
147 * @function size()
148 * @abstract Find size of the array and store the index of the top element ↩
       in a variable
149 * @param
150 * @return index of top item
151 * @pre
152 * @post
153 **/
154 template <class T>
155 void ArrayStack<T>::size() {
156     int i = 0;
157     // Find top item
158     if( !empty() ) {
159         // Array is not empty
160         while (array[i + 1] != 0) {
161             i = i + 1;
162         }
163         top_number = i; // Remember this for later use
164     }
165 }
166 
167 /**
168 * @function output(ostream & out) //TODO ask how output and ostream works ↩
       and if is nesessairy?
169 * @abstract Use ostream to give output to user
170 * @param out: the data to return
171 * @return output
172 * @pre input
173 * @post output
174 **/
175 template <class T>
176 void ArrayStack<T>::read() {
177     for (int k = top_number; k > 0 ; k--) {
178         std::cout << array[k];
179     }
180     std::cout << array[0];
181 }
182 
183 #endif
```

```
1  /**
2  * Class name: PointerStack, stack implemented using a singly connected ↩
       list
3  * @author Lisa Pothoven (s1328263)
4  * @author David Kleingeld (s1432982)
5  * @file PointerStack.h
6  * @date 20-09-2016
7  **/
8
9  #ifndef PointerStack_h
10 #define PointerStack_h
11
12 template <class T>
13 class node {
14 public:
15     node<T>* next;
16     T value;
17 };
18
19
20 template <class T>
21 class PointerStack {
22 public:
23     PointerStack();//constructor, cant be private
24     bool empty();//TODO named empty to conform to default c++ stack names
25     void clear();
26     void push(T newItem);
27     bool pop();
28     bool top(T & topItem);
29     void read();
30 private:
31     node<T>* topElement;//stores the highest element in the stack
32 };
33
34 /**
35 * @function PointerStack()
36 * @abstract Constructor: Create new object node
37 * @param
38 * @return
39 * @pre
40 * @post
41 **/
42 template <class T>
43 PointerStack<T>::PointerStack() {
44     topElement = new node<T>;
```

```
45      topElement->next = NULL;
46  //    topElement->value = -1;//TODO would it be more efficient to fill ↩
        this value
47                                //in the top function?
48  }
49  //NOTE these are equivalent
50  //  a->b
51  //  (*a).b  call member b that foo points to, (read right to left, [*a] = ↩
        [value pointed to])
52
53  /**
54  * @function empty()
55  * @abstract Check if object PointerStack is empty
56  * @param none
57  * @return true (is empty) or false (is not empty)
58  * @pre
59  * @post
60  **/
61  template <class T>
62  bool PointerStack<T>::empty() {
63      if (topElement == NULL) {
64          return true;
65      }
66      return false;
67  }
68
69  /**
70  * @function clear()
71  * @abstract Clear content of the pointer stack, delete every element
72  * @param none
73  * @return
74  * @pre pointer to filled stack
75  * @post Empty stack
76  **/
77  template <class T>
78  void PointerStack<T>::clear() {
79     while (topElement != NULL){pop(); }
80     //delete topElement;//TODO make deconstructor do this?
81  }
82
83  /**
84  * @function push(newItem)
85  * @abstract Add newItem to top of the stack
86  * @param newItem: new item to be added to the stack
87  * @return
88  * @pre stack of n elements
89  * @post stack of n+1 elements
```

```
90  **/
91  template <class T>
92  void PointerStack<T>::push(T newItem) {
93      node<T>* old_top;
94      old_top = topElement;
95
96      topElement = new node<T>;
97      topElement->next = old_top;
98
99      topElement->value = newItem;
100 }
101
102 /**
103 * @function pop()
104 * @abstract Remove item from top of the pointer stack
105 * @param none
106 * @return false if there is no element to pop else true
107 * @pre pointer to first element
108 * @post pointer to second element and first element deleted
109 **/
110 template <class T>
111 bool PointerStack<T>::pop() {
112     node<T>* old_top;
113     old_top = topElement;
114
115     if (!empty() ){
116         topElement = topElement->next;
117         delete old_top;
118         return true;
119         }
120     else{
121         return false;
122     }
123 }
124
125 /**
126 * @function top(topItem)
127 * @abstract Return item at the top of the stack, give output to user
128 * @param topItem: the item at the top of the stack
129 * @return true (succes) or false (if there is no item)
130 * @pre stack of n elements
131 * @post stack of n elements
132 **/
133 template <class T>
134 bool PointerStack<T>::top(T & topItem) {
135
136     if (empty() ) { return false;}
```

```
137     else{
138         topItem = topElement->value;
139         return true;
140     }
141 }
142 
143 /**
144 * @function read()
145 * @abstract Read the data in the pointer stack and give output to user
146 * @param
147 * @return
148 * @pre
149 * @post
150 **/
151 template <class T>
152 void PointerStack<T>::read() {//TODO not in assaignment should we keep?
153     T topItem;
154     node<T>* current = topElement;
155     if (!empty() ){
156         while (current->next != NULL){
157             topItem = current->value;
158             current = current->next;
159             std::cout << topItem;
160         }
161     }
162 }
163 
164 #endif
```

```
1  /**
2  * Class name: VectorStack, stack implemented using the standardliberary ↩
       stack
3  * @author Lisa Pothoven (s1328263)
4  * @author David Kleingeld (s1432982)
5  * @file arraystack.h
6  * @date 20-09-2016
7  **/
8  #include <vector>
9
10 #ifndef VectorStack_h
11 #define VectorStack_h
12
13 template <class T>
14 class VectorStack {
15 public:
16     VectorStack();
17     bool empty();//as an argument
18     void clear();
19     void push(T newItem);
20     bool pop();
21     bool top(T & topItem);
22     void read();
23 private:
24     std::vector<T> vect;
25     // Variable to keep track of the top of the array
26     int top_number;
27 };
28
29 /**
30 * @function VectorStack()
31 * @abstract Constructor: Create object VectorStack
32 * @param
33 * @return VectorStack object
34 * @pre
35 * @post
36 **/
37 template <class T>
38 VectorStack<T>::VectorStack() {
39 }
40
41 /**
42 * @function empty()
43 * @abstract Check if object VectorStack is empty
44 * @param
```

```
45  * @return true (is empty) or false (is not empty)
46  * @pre
47  * @post
48  **/
49  template <class T>
50  bool VectorStack<T>::empty() {
51      return vect.empty();
52  }
53
54  /**
55  * @function clear()
56  * @abstract Clear content of VectorStack
57  * @param
58  * @return true (succes) or false (something did not work)
59  * @pre Stack of n elements
60  * @post Stack of 0 elements
61  **/
62  template <class T>
63  void VectorStack<T>::clear() {
64    vect.clear();
65  }
66
67  /**
68  * @function push(T newItem)
69  * @abstract Add newItem to top of the stack
70  * @param newItem: new item to be added to the stack
71  * @return true (succes) or false (something did not work)
72  * @pre VectorStack of n elements, newItem
73  * @post VectorStack of n+1 elements
74  **/
75  template <class T>
76  void VectorStack<T>::push(T newItem) {
77    vect.push_back(newItem);
78  }
79
80  /**
81  * @function pop()
82  * @abstract Remove item from top of the stack
83  * @param
84  * @return true (succes) or false (something did not work)
85  * @pre VectorStack of n elements
86  * @post VectorStack of n-1 elements
87  **/
88  template <class T>
89  bool VectorStack<T>::pop() {
90      if (!empty())
91      {
```

```
92      vect.pop_back();
93          return true;
94      }
95      else // vect must be empty
96      {
97          return false;
98      }
99  }
100 
101 /**
102 * @function top(topItem)
103 * @abstract Return item at the top of the stack, give output to user
104 * @param topItem: the item at the top of the stack
105 * @return true (succes) or false (something did not work)
106 * @pre VectorStack of n elements
107 * @post VectorStack of n elements
108 **/
109 template <class T>
110 bool VectorStack<T>::top(T & topItem) {
111     if (!empty()) {
112     topItem = vect.back();
113     return true;
114     }
115     else {
116         return false;
117     }
118 }
119 
120 
121 /**
122 * @function read() //TODO ask how output and ostream works and if is ↩
        nesessairy?
123 * @abstract Read the data in VectorStack and give output to user
124 * @param
125 * @return
126 * @pre
127 * @post
128 **/
129 template <class T>
130 void VectorStack<T>::read() {
131   for (unsigned int i=vect.size()-1; i>0; i--)
132     std::cout << vect[i];
133   std::cout << vect[0];//do last element
134 }
135 
136 #endif
```

```
1  /**
2  * Class name: StlStack, wrapper around the standard libery stack
3  * @author Lisa Pothoven (s1328263)
4  * @author David Kleingeld (s1432982)
5  * @file arraystack.h
6  * @date 16-09-2016
7  **/
8
9  #include <iostream>
10 #include <stack>
11
12 #ifndef StlStack_h
13 #define StlStack_h
14
15 template <class T>
16 class StlStack {
17 public:
18     StlStack();
19     bool empty();
20     void clear();
21     void push(T newItem);
22     bool pop();
23     bool top(T & topItem);
24     void read();
25 private:
26     std::stack<T> Stack;
27 };
28
29 /**
30 * @function StlStack()
31 * @abstract Constructor: Create new object node
32 * @param
33 * @return
34 * @pre
35 * @post
36 **/
37 template <class T>
38 StlStack<T>::StlStack() {
39 }
40
41 /**
42 * @function empty()
43 * @abstract Check if container in stack is empty
44 * @param none
45 * @return true (is empty) or false (is not empty)
```

```
46  * @pre
47  * @post
48  **/
49  template <class T>
50  bool StlStack<T>::empty() {
51      if (Stack.empty()) {
52          return true;
53      }
54      else {
55          return false;
56      }
57  }
58
59  /**
60  * @function clear()
61  * @abstract Clear content of the stack, delete every element
62  * @param none
63  * @return
64  * @pre pointer to filled stack
65  * @post Empty stack
66  **/
67  template <class T>
68  void StlStack<T>::clear() {
69      while(!empty() ) {
70          Stack.pop();
71      }
72  }
73
74  /**
75  * @function push(newItem)
76  * @abstract Add newItem to top of the stack
77  * @param newItem: new item to be added to the stack
78  * @return
79  * @pre stack of n elements
80  * @post stack of n+1 elements
81  **/
82  template <class T>
83  void StlStack<T>::push(T newItem) {
84      Stack.push(newItem);
85  }
86
87  /**
88  * @function pop()
89  * @abstract Remove item from top of the stack
90  * @param none
91  * @return false if there is no element to pop else true
92  * @pre pointer to first element
```

```
93  * @post pointer to second element and first element deleted
94  **/
95  template <class T>
96  bool StlStack<T>::pop() {
97      if (empty()) {
98          return false;
99      }
100     Stack.pop();
101     return true;
102 }
103
104 /**
105 * @function top(topItem)
106 * @abstract Return item at the top of the stack, give output to user
107 * @param topItem: the item at the top of the stack
108 * @return true (succes) or false (if there is no item)
109 * @pre stack of n elements
110 * @post stack of n elements
111 **/
112 template <class T>
113 bool StlStack<T>::top(T & topItem) {
114     if (empty()) {
115         return false;
116     }
117     topItem = Stack.top();
118     return true;
119 }
120
121 template <class T>
122 void StlStack<T>::read() {//TODO not in assaignment should we keep?
123     while (!empty()) {
124         std::cout << Stack.top();
125         Stack.pop();
126     }
127 }
128
129 #endif
```