

```

1  /**
2  * @file - ArrayStack.cpp
3  *
4  * Implements StackADT<T> using array as the internal data structure for the stack
  elements.
5  *
6  * @author - Humayun Kabir, Instructor, CSCI 161, VIU
7  * @version - 0.0.1
8  * @date - April 25, 2021
9  *
10 */
11
12 #include <iostream>
13 #include "StackADT.h"
14
15 using namespace std;
16
17 #define DEFAULT_CAPACITY 100
18
19 template <typename T>
20 class ArrayStack: public StackADT<T> {
21 private:
22     int top;
23     int capacity;
24     T* elements;
25
26 public:
27
28     /**
29     * Default constructor
30     */
31     ArrayStack(): StackADT<T>::StackADT(), top(-1), capacity(DEFAULT_CAPACITY), elements(
new T[capacity]) {
32         cout<<"ArrayStack::default constructor....."<<endl;
33     }
34
35     /**
36     * Regular constructor
37     */
38     ArrayStack(int capacity): StackADT<T>::StackADT(), top(-1), capacity(capacity),
elements(new T[capacity]) {
39         cout<<"ArrayStack::regular constructor....."<<endl;
40     }
41
42     /**
43     * Copy constructor
44     */
45     ArrayStack(const ArrayStack& copy): StackADT<T>::StackADT(), top(copy.top), capacity(
copy.capacity), elements(new T[capacity]) {
46         for (int i=0; i<=top; i++) {
47             elements[i] = copy.elements[i];
48         }
49         cout<<"ArrayStack::copy constructor....."<<endl;
50     }
51
52     /**
53     * Move constructor
54     */
55     ArrayStack(ArrayStack&& temp): StackADT<T>::StackADT(), top(temp.top), capacity(temp.
capacity), elements(temp.elements) {
56         temp.elements = NULL;
57         cout<<"ArrayStack::move constructor....."<<endl;
58     }
59
60     /**
61     * Destructor
62     */
63     ~ArrayStack() {
64         if (elements != NULL) {

```

```

65         delete [] elements;
66     }
67     cout<<"ArrayStack::~destructor....."<<endl;
68 }
69
70
71 /**
72  * Copy assignment operator
73  */
74 ArrayStack& operator = (const ArrayStack& copy) {
75     if( this == &copy ) {
76         return *this;
77     }
78     top = copy.top;
79     capacity = copy.capacity;
80     delete [] elements;
81     elements = new T[capacity];
82     for ( int i=0; i<=top; i++ ) {
83         elements[i] = copy.elements[i];
84     }
85     cout<<"ArrayStack::copy assignment....."<<endl;
86     return *this;
87 }
88
89
90 /**
91  * Move assignment operator
92  */
93 ArrayStack& operator = (ArrayStack&& temp) {
94     if( this == &temp) {
95         return *this;
96     }
97     top = temp.top;
98     capacity = temp.capacity;
99     delete [] elements;
100    elements = temp.elements;
101    temp.elements = NULL;
102    cout<<"ArrayStack::move assignment....."<<endl;
103    return *this;
104 }
105
106 /**
107  * @brief - Gives the capacity of the internal array that is being used to hold the
108  * elements of the stack.
109  *
110  * This function will be available to use only with ArrayStack<T> object or reference
111  * but not with a
112  * StackADT<T> reference.
113  *
114  * @return - capacity of the internal array
115  */
116 int getCapacity() {
117     return capacity;
118 }
119
120 /**
121  * @brief - Pushes the element on the top of the stack and advances the top
122  *
123  * This function will be available to use with ArrayStack<T> object and both
124  * ArrayStack<T> and StackADT<T>
125  * references.
126  *
127  * @param - element, the element to be pushed onto the stack.
128  */
129 void push(T element) override {
130     if(top < (capacity-1)){
131         elements[++top] = element;
132     }
133     else {

```

```

131     throw "Stack is Full!";
132 }
133 }
134
135 /**
136  * @brief - Pops the top element of the stack and reverts the top
137  *
138  * This function will be available to use with ArrayStack<T> object and both
139  * ArrayStack<T> and StackADT<T>
140  * references.
141  *
142  * @return - top element
143  */
144 T pop() override {
145     if(top > -1) {
146         return elements[top--];
147     }
148     else {
149         throw "Stack is Empty!";
150     }
151 }
152 }
153
154
155 /**
156  * @brief - Gives the top element of the stack and does not revert the top
157  *
158  * This function will be available to use with ArrayStack<T> object and both
159  * ArrayStack<T> and StackADT<T>
160  * references.
161  *
162  * @return - top element
163  */
164 T peek() override {
165     if(top > -1) {
166         return elements[top];
167     }
168     else {
169         throw "Stack is Empty";
170     }
171 }
172
173 /**
174  * @brief - Gives the current size or the number of elements that has been pushed inot
175  * the stack but not popped yet.
176  *
177  * This function will be available to use only with ArrayStack<T> object or reference
178  * but not with a
179  * StackADT<T> reference.
180  *
181  * @return - current size of the stack
182  */
183 int getSize() override {
184     return top+1;
185 }
186
187 /**
188  * @brief - Returns true if the stack has no element to pop, false otherwise.
189  *
190  * This function will be available to use only with ArrayStack<T> object or reference
191  * but not with a
192  * StackADT<T> reference.
193  *
194  * @return - true if the stack is empty.
195  */
196 bool isEmpty() override {

```

```
195     return top == -1;
196 }
197
198 };
199
```