```cpp
/**
 * @file -  ArrayStack.cpp
 *
 * Implements StackADT<T> using array as the internal data structure for the stack elements.
 *
 * @author - Humayun Kabir, Instructor, CSCI 161, VIU
 * @version - 0.0.1
 * @date - April 25, 2021
 *
 */

#include <iostream>
#include "StackADT.h"

using namespace std;

#define DEFAULT_CAPACITY 100

template <typename T>
class ArrayStack: public StackADT<T> {
private:
  int top;
  int capacity;
  T* elements;

public:

  /**
   * Default constructor
   */
  ArrayStack(): StackADT<T>::StackADT(), top(-1), capacity(DEFAULT_CAPACITY), elements(new T[capacity]) {
    cout<<"ArrayStack::default constructor......."<<endl;
  }

  /**
   * Regular constructor
   */
  ArrayStack(int capacity): StackADT<T>::StackADT(), top(-1), capacity(capacity), elements(new T[capacity]) {
    cout<<"ArrayStack::regular constructor......."<<endl;
  }

  /**
   * Copy constructor
   */
  ArrayStack(const ArrayStack& copy): StackADT<T>::StackADT(), top(copy.top), capacity(copy.capacity), elements(new T[capacity]) {
      for (int i=0; i<=top; i++) {
          elements[i] = copy.elements[i];
      }
      cout<<"ArrayStack::copy constructor......."<<endl;
  }

  /**
   * Move constructor
   */
  ArrayStack(ArrayStack&& temp): StackADT<T>::StackADT(), top(temp.top), capacity(temp.capacity), elements(temp.elements) {
      temp.elements = NULL;
      cout<<"ArrayStack::move constructor......"<<endl;
  }

  /**
   * Destructor
   */
  ~ArrayStack() {
      if (elements != NULL) {
```

```cpp
                delete [] elements;
            }
        cout<<"ArrayStack::destructor......"<<endl;
    }


    /**
     * Copy assignment operator
     */
    ArrayStack& operator = (const ArrayStack& copy) {
        if( this == &copy ) {
            return *this;
        }
        top = copy.top;
        capacity = copy.capacity;
        delete [] elements;
        elements = new T[capacity];
        for ( int i=0; i<=top; i++ ) {
            elements[i] = copy.elements[i];
        }
        cout<<"ArrayStack::copy assignment......."<<endl;
        return *this;
    }


    /**
     * Move assignment operator
     */
    ArrayStack& operator = (ArrayStack&& temp) {
          if( this == &temp) {
              return *this;
          }
          top = temp.top;
          capacity = temp.capacity;
          delete [] elements;
          elements = temp.elements;
          temp.elements = NULL;
          cout<<"ArrayStack::move assignment......"<<endl;
          return *this;
    }

    /**
     * @brief - Gives the capacity of the internal array that is being used to hold the
     elements of the stack.
     *
     * This function will be available to use only with ArrayStack<T> object or reference
     but not with a
     * StackADT<T> reference.
     *
     * @return - capcity of the internal array
     */
    int getCapacity() {
        return capacity;
    }

    /**
     * @brief - Pushes the element on the top of the stack and advances the top
     *
     * This function will be available to use with ArrayStack<T> object and both
     ArrayStack<T> and StackADT<T>
     * references.
     *
     * @param - element, the element to be pushed onto the stack.
     */
    void push(T element) override {
        if(top < (capacity-1)){
          elements[++top] = element;
        }
        else {
```

```
        throw "Stack is Full!";
      }
    }

     /**
     * @brief - Pops the top element of the stack and reverts the top
     *
     * This function will be available to use with ArrayStack<T> object and both
     ArrayStack<T> and StackADT<T>
     * references.
     *
     * @return - top element
     */

    T pop() override {
      if(top > -1) {

        return elements[top--];
      }
      else {
        throw "Stack is Empty!";
      }
    }


     /**
     * @brief - Gives the top element of the stack and does not revert the top
     *
     * This function will be available to use with ArrayStack<T> object and both
     ArrayStack<T> and StackADT<T>
     * references.
     *
     * @return - top element
     */
    T peek() override {
      if(top > -1) {
        return elements[top];
      }
      else {
        throw "Stack is Empty";
      }
    }

     /**
     * @brief - Gives the current size or the number of elements that has been pushed inot
     the stack but not popped yet.
     *
     * This function will be available to use only with ArrayStack<T> object or reference
     but not with a
     * StackADT<T> reference.
     *
     * @return - current size of the stack
     */

    int getSize() override {
      return top+1;
    }


     /**
     * @brief - Returns true if the stack has no element to pop, false otherwise.
     *
     * This function will be available to use only with ArrayStack<T> object or reference
     but not with a
     * StackADT<T> reference.
     *
     * @return - true if the stack is empty.
     */
    bool isEmpty() override {
        return top == -1;
    }

};

```