ArrayStack.h

```
 1: #ifndef ARRAYSTACK_H
 2: #define ARRAYSTACK_H
 3:
 4: #include <string>
 5: using namespace std;
 6:
 7: //-----------------------------------------------------------
 8: // For illustration purposes, this will be a stack of strings
 9: //-----------------------------------------------------------
10:
11: class ArrayStack {
12: private:
13:     enum { DEF_CAPACITY = 10 };                    // default stack capacity
14:
15:     string* S;                                     // array of stack elements
16:     int capacity;                                  // stack capacity
17:     int t;                                         // index of top of the stack
18:
19: public:
20:     ArrayStack(int cap=DEF_CAPACITY);              // constructor from capacity
21:     int size() const;                              // number of items in stack
22:     bool empty() const;                            // is the stack empty?
23:     const string& top() const;                     // get the top element
24:     void push(const string& e);                    // push element onto stack
25:     void pop();                                    // pop the stack
26:
27:     // Housekeeping functions
28:     ArrayStack(const ArrayStack& other);           // copy constructor
29:     ArrayStack& operator=(const ArrayStack& other); // assignment operator
30:     ~ArrayStack();                                 // destructor
31: };
32:
33: #endif
```

ArrayStack.cpp

```
 1: #include <iostream>
 2: #include <stdexcept>
 3: #include "ArrayStack.h"
 4: using namespace std;
 5:
 6: ArrayStack::ArrayStack(int cap)
 7:     : S(new string[cap]), capacity(cap), t(-1) {}
 8:
 9: int ArrayStack::size() const {
10:     return (t + 1);
11: }
12:
13: bool ArrayStack::empty() const {
14:     return (t < 0);
15: }
16:
17: const string& ArrayStack::top() const {
18:     if (empty()) throw runtime_error("Stack is Empty");
19:     return S[t];
20: }
21:
22: void ArrayStack::push(const string& e) {
23:     if (size() == capacity) throw runtime_error("Stack is Full");
24:     S[++t] = e;                // note well use of preincrement
25: }
26:
27: void ArrayStack::pop() {
28:     if (empty()) throw runtime_error("Stack is Emtpy");
29:     --t;
30: }
31:
32: // ------------------ housekeeping functions follow --------------------
33:
34: // Copy Constructor
35: ArrayStack::ArrayStack(const ArrayStack& other)
36:     : S(new string[other.capacity]), capacity(other.capacity), t(other.t)
37: {
38:     for (int j=0; j <= t; j++)
39:         S[j] = other.S[j];
40: }
41:
42: // Destructor
43: ArrayStack::~ArrayStack() {
44:     delete[] S;
45: }
46:
47: // Assignment Operator
48: ArrayStack& ArrayStack::operator=(const ArrayStack& other) {
49:     if (this != &other) {
50:         capacity = other.capacity;
51:         t = other.t;
52:         delete[] S;
53:         S = new string[capacity];
54:         for (int j=0; j <= t; j++)
55:             S[j] = other.S[j];
56:     }
57:     return *this;
58: }
```