```
 1: #ifndef CSCI180_ARRAY_STACK_H
 2: #define CSCI180_ARRAY_STACK_H
 3:
 4: #include <stdexcept>
 5:
 6: namespace csci180 {
 7:
 8:   /** A stack implementation based upon use of a fixed-sized array.
 9:    *  Elements are inserted and removed according to the last-in
10:    *  first-out principle.
11:    *
12:    *  This implementation is based  on that given on pages 163-164
13:    *  of our text, but it has been adjusted to suit my tastes.
14:    */
15:   template <typename Object>
16:   class ArrayStack {
17:
18:   private:
19:     int       capacity;                  // actual length of underlying array
20:     Object*   S;                         // pointer to the underlying array
21:     int       t;                         // index of the top of the stack
22:
23:   public:
24:
25:     /** Standard constructor creates an empty stack with given capacity. */
26:     ArrayStack(int cap = 1000) :
27:       capacity(cap), S(new Object[capacity]), t(-1)  { }
28:
29:     /** Returns the number of objects in the stack.
30:      *  @return number of elements
31:      */
32:     int size() const {
33:       return t+1;
34:     }
35:
36:     /** Determines if the stack is currently empty.
37:      *  @return true if empty, false otherwise.
38:      */
39:     bool empty() const {
40:       return t < 0;
41:     }
42:
43:     /** Returns a const reference to the top object in the stack.
44:      * @return reference to top element
45:      */
46:     const Object& top() const {
47:       if (empty())
48:         throw std::runtime_error("Access to empty stack");
49:       return S[t];
50:     }
51:
52:     /** Returns a live reference to the top object in the stack.
53:      * @return reference to top element
54:      */
55:     Object& top() {
56:       if (empty())
57:         throw std::runtime_error("Access to empty stack");
58:       return S[t];
59:     }
```

```
60:     /** Inserts an object at the top of the stack.
61:      *  @param the new element
62:      */
63:     void push(const Object& elem) {
64:       if (size() == capacity)
65:         throw std::runtime_error("Stack overflow");
66:       S[++t] = elem;
67:     }
68:
69:     /** Removes the top object from the stack. */
70:     void pop()  {
71:       if (empty())
72:         throw std::runtime_error("Access to empty stack");
73:       t--;
74:     }
75:
76:     // Housekeeping functions
77:
78:     /** Copy constructor */
79:     ArrayStack(const ArrayStack& other) :
80:       capacity(other.capacity), S(new Object[capacity]), t(other.t)
81:     {
82:       for (int i=0; i <= t; i++)
83:         S[i] = other.S[i];
84:     }
85:
86:     /** Destructor */
87:     ~ArrayStack() {
88:       delete[] S;
89:     }
90:
91:     /** Assignment operator */
92:     ArrayStack& operator=(const ArrayStack& other) {
93:       if (this != &other) {             // avoid self copy (x = x)
94:         capacity = other.capacity;
95:         t = other.t;
96:         delete [] S;                    // delete old contents
97:         S = new Object[capacity];
98:         for (int i=0; i <= t; i++)
99:           S[i] = other.S[i];
100:      }
101:      return *this;
102:    }
103:
104:  };  // end of ArrayStack class
105:
106: } // end of csci180 namespace
107: #endif
```