```
#ifndef CSCI180_LINKED_STACK_H
#define CSCI180_LINKED_STACK_H

#include <stdexcept>

namespace csci180 {

  /** A stack implementation based upon use of a singly-linked list.
   *  Elements are inserted and removed according to the last-in
   *  first-out principle.
   *
   *  This implementation is based  on that given pages 180-181
   *  of our text, but it has been adjusted to suit my tastes.
   */
  template <typename Object>
  class LinkedStack {

  protected:
    struct Node {                               // a node in the stack
      Object element;                           // element
      Node*  next;                              // next pointer
      Node(const Object& e = Object(), Node* n = NULL)
        : element(e), next(n) { }               // constructor
    };

  private:
    Node* tp;                                   // pointer to stack top
    int sz;                                     // number of items in stack

  public:
    /** Standard constructor creates an empty stack. */
    LinkedStack() : tp(NULL), sz(0) { }

    /** Returns the number of objects in the stack.
     *  @return number of elements
     */
    int size() const {
      return sz;
    }

    /** Determines if the stack is currently empty.
     *  @return true if empty, false otherwise.
     */
    bool empty() const {
      return sz == 0;
    }

    /** Returns a const reference to the top object in the stack.
     * @return reference to top element
     */
    const Object& top() const {
      if (empty())
        throw std::runtime_error("Access to empty stack");
      return tp->element;
    }

    /** Returns a live reference to the top object in the stack.
     * @return reference to top element
     */
    Object& top() {
      if (empty())
        throw std::runtime_error("Access to empty stack");
      return tp->element;
    }
    /** Inserts an object at the top of the stack.
     *  @param the new element
     */
    void push(const Object& elem) {
      tp = new Node(elem, tp);                 // new node points to old top
      sz++;
    }

    /** Removes the top object from the stack. */
    void pop()  {
      if (empty())
        throw std::runtime_error("Access to empty stack");
      Node* old = tp;                         // node to remove
      tp = tp->next;
      sz--;
      delete old;
    }

  protected:                                    // protected utilities
    void removeAll() {                          // remove entire stack contents
      while (!empty()) pop();
    }

    void copyFrom(const LinkedStack& other) {   // copy from other
      tp = NULL;
      Node* model = other.tp;                   // model is current node in other
      Node* prev   = NULL;
      while (model != NULL) {
        Node* v = new Node(model->element, NULL);  // make copy of model
        if (tp == NULL)
          tp = v;                               // if first node
        else
          prev->next = v;                       // else link after prev
        prev  = v;
        model = model->next;
      }
      sz = other.sz;
    }

  public:
    /** Copy constructor */
    LinkedStack(const LinkedStack& other) {
      copyFrom(other);
    }

    /** Destructor */
    ~LinkedStack() {
      removeAll();
    }

    /** Assignment operator */
    LinkedStack& operator=(const LinkedStack& other) {
      if (this != &other) {                     // avoid self copy (x = x)
        removeAll();                            // remove old contents
        copyFrom(other);                        // copy new contents
      }
      return *this;
    }

  };  // end of LinkedStack class

} // end of csci180 namespace
#endif
```