CS242: Object-Oriented Design and Programming

Program Assignment 5
Due Thursday, Feb 26^{th}, 1998

Robust and Efficient Unbounded Stack

Most implementations of your previous unbounded Stack implementation of the ADT Stack had the following limitations:

- *Non-robust* – many implementations did not handle memory failure correctly, *e.g.* with respect to `push` operations.
- *Inefficient* – many implementations contained excessive calls to global `operator new` and `delete`.

Therefore, your next assignment will be to write a more robust and efficient unbounded stack using dynamic memory and a linked list. This will also give you more practice using dynamic memory management in C++.

Note that this change will mostly affect your stack representation, rather than the stack interface. Your task is to reimplement the methods that operate upon objects of class `Stack`:

```
/* -*- C++ -*- */
#include <stdlib.h>

// Forward declaration (use the "Cheshire Cat" approach
// for information hiding).
template <class T> class Node;

template <class T>
class Stack
  // = TITLE
  //     Implement a generic LIFO abstract data type.
  //
  // = DESCRIPTION
  //     This implementation of a Stack uses a linked list.
{
public:
  typedef T TYPE;
  // C++ trait.

  // = Initialization, assignment, and termination methods.
  Stack (size_t size_hint);
  // Initialize a new stack so that it is empty.  The
  // <size_hint> can be used to preinitialize the <Node>
  // freelist if you so desire.

  Stack (const Stack<T> &s);
  // The copy constructor (performs initialization).

  void operator= (const Stack<T> &s);
  // Assignment operator (performs assignment).
```

```
 ~Stack (void);
  // Perform actions needed when stack goes out of scope.

  // = Classic Stack operations.

  void push (const T &new_item);
  // Place a new item on top of the stack. Does not check if the
  // stack is full.

  void pop (T &item);
  // Remove and return the top stack item. Does not check if stack
  // is full.

  void top (T &item) const;
  // Return top stack item without removing it. Does not check if
  // stack is empty.

  // = Check boundary conditions for Stack operations.

  int is_empty (void) const;
  // Returns 1 if the stack is empty, otherwise returns 0.

  int is_full (void) const;
  // Returns 1 if the stack is full, otherwise returns 0.

  int operator == (const Stack<T> &s) const;
  // Checks for Stack equality.

  int operator != (const Stack<T> &s) const;
  // Checks for Stack inequality.

   static void delete_free_list (void);
   // Returns all dynamic memory on the free list to the free store.

private:
  void delete_all_nodes (void);
  // Delete all the nodes in the stack.

  void copy_all_nodes (const Stack<T> &s);
  // Copy all nodes from <s> to <this>.

  Node<T> *head_;
  // Head of the linked list of <Nodes>.
};
```

Note that `push`, `pop`, and `top` do *not* explicitly check whether the stack is empty or full. Therefore, you *must* call `is_empty` or `is_full` before adding, removing, or viewing a stack element.

The following `Node` class defines operations on a node in the linked list. Since your solution uses the "Cheshire Cat" approach to information hiding, you can actually put this implementation into your Stack.C file. Therefore, clients of this class won't have access to the implementation at all!

```
template <class T>
class Node
{
  // = TITLE
  //     Implement a generic <Node>.
```

```
  //
  // = DESCRIPTION
  //     <Node>s can be chained together via their <next_> field.
  //     This class also implements an efficient memory cache
  //     using class-specific overloaded operator new and delete.
public:
  // = Constructors.
  Node (const T &val, Node<T> *next = 0);
  // Create and initialize a node to a certain value, assigning
  // <next> to <next_>.

  Node (void);
  // Just create the node, no initializing.

  Node (const Node<T> &from);
  // The copy constructor (performs initialization).

  // = Class-specific freelist management methods.
  void *operator new (size_t n_bytes);
  void  operator delete (void *ptr);

  static void free_all_nodes (void);
  // Returns all dynamic memory on the free list to the free store.

  // = Accessors.
  Node<T> *next (void) const;
  // Get the pointer to the next node.

  void next (Node<T> *new_next);
  // Set the pointer to the next node.

  const T &value (void) const;
  // Get the current value at this node.

  void value (const T &a);
  // Set the current value of this node.

private:
  T value_;
  // Value at this node.

  Node<T> *next_;
  // Pointer to the next node.

  static Node<T> *free_list_;
  // Head of the free list of Nodes used to speed up allocation.
};
```

Points to Ponder

The following are some points to consider in your implementation:

- *Node of last resort* – You decide where to put the node of last resort. For instance, it can go in each instance of the `Stack` class or as a `static` data member in the `Node` class. Whatever you decide to do, make sure to document and justify your choice.

- *Memory allocation within the class-specific new operator* – You need to decide how to allocate memory within the class-specific `operator new`. I recommend you use `new char[n_bytes]` and then cast the result to a `Node<T>`.
- *You must NOT overload the global operator new* – Failure to follow this rule will result in death by public ridicule!
- *Exception handling* – For the purposes of this assignment, please don't use exceptions. In particular, if your C++ compiler throws an exception when `new` fails make sure you either (1) disable this or (2) use the `nothrow` argument to the operator placement new.

For this assignment, please use the same test driver as before, and just modify the Stack.h and Stack.C implementation to use dynamically allocated memory + the `Node` cache freelist, a linked list, and the "node of last resort" technique we discussed in class.