CS 215/91: Intermediate Software Design

Programming Assignment 3
Part 1 due Wednesday, Feb 15th, 2006
Part 1 due Wednesday, Feb 22th, 2006

A stack is an *Abstract Data Type* (ADT) that implements a priority queue with "last-in, first-out" (LIFO) behavior. Common operations on a stack include *push*, *pop*, *top*, and *is_empty*. This part of your programming assignment focuses upon building and using the following two different implementations of stacks:

1. *AStack* – the first one will use an array whose size is given an initial value at creation time, but which can grow over time. Implementing this program should be trivial now that you've implemented the `Array` class.
2. *LStack* – the second one will use using dynamic memory organized as a linked list. Note that this change only affects the stack representation, but does not affect the stack method interface.

Part 1: Array Stack (AStack)

For this assignment you will write is a stack using the class `Array` you implemented for your first assignment. Here's the class declaration for this `AStack`:

```
/**
 * @class AStack
 *
 * @brief Implement a generic LIFO abstract data type using a resizeable array.
 */

template <typename T>
class AStack
{
public:
  typedef T TYPE;
  // C++ trait.

  enum { INITIAL_STACK_SIZE = 80 };

  // Initialize a new stack so that it is empty.
  AStack (size_t size = INITIAL_STACK_SIZE);

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

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

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

  // Place a new item on top of the stack.  Throws the
  // <Stack::overflow> exception if the stack is full.
  void push (const T &new_item) throw (Stack::overflow);
  // Remove and return the top stack item.  Throws the
  // <Stack::underflow> exception if the stack is empty.
  void pop (T &item) throw (Stack::underflow);

  // Return top stack item without removing it.  Throws the
  // <Stack::underflow> exception if the stack is empty.
  void top (T &item) const throw (Stack::underflow);

  // Returns true if the stack is empty, otherwise returns false.
  bool is_empty (void) const;

  // Checks for stack equality.
  bool operator == (const AStack<T> &s) const;

  // Checks for stack inequality.
  bool operator != (const AStack<T> &s) const;

private:
  // Keeps track of the top of the stack.
  size_t top_;

  // Array that stores the elements in the stack.
  Array<T> stack_;
};
```

The `pop()` and `top()` methods explicitly check whether the stack is empty using the `is_empty()` method. The `push()` method uses the `Array::set()` method, which grows the array as necessary.

Part 2: List Stack (LStack)

In this variant, your task is to implement the methods that operate upon objects of class `LStack`, which uses a linked list representation:

```
// Forward declaration (use the "Cheshire Cat" approach to information
// hiding).
template <typename T>
class Stack_Node;

/**
 * @class LStack
 *
 * @brief Implement a generic LIFO abstract data type using a linked list.
 */

template <typename T>
class LStack
{
public:
  typedef T TYPE;

  // = Initialization, assignment, and termination methods.

  // Initialize a new stack so that it is empty.
  LStack (size_t size_hint = 0);
  // The copy constructor (performs initialization).
  LStack (const LStack<T> &s);

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

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

  // = Classic Stack operations.

  // Place a new item on top of the stack.  Throws the
  // <Stack::overflow> exception if the stack is full.
  void push (const T &new_item) throw (Stack::overflow);

  // Remove and return the top stack item.  Throws the
  // <Stack::underflow> exception if the stack is empty.
  void pop (T &item) throw (Stack::underflow);

  // Return top stack item without removing it.  Throws the
  // <Stack::underflow> exception if the stack is empty.
  void top (T &item) const throw (Stack::underflow);

  // = Check boundary conditions for Stack operations.

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

  // Checks for stack equality.
  bool operator == (const LStack<T> &s) const;

  // Checks for stack inequality.
  bool operator != (const LStack<T> &s) const;

  // Returns all dynamic memory on the LStack<T> free list to the global free store.
  static void delete_free_list (void);

private:

  // Swap the head of the linked list. Useful in implementing
  // exception-safe assignment operator.
  void swap (LStack<T>& s);

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

  // Copy all nodes from <s> to <this>.  Assumes that <this> has no
  // nodes in it.
  void copy_all_nodes (const LStack<T> &s);

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

The `pop()` and `top()` methods explicitly check whether the stack is empty using the `is_empty()` method. The `push()` method allocates a new `Node` object and connects it at the head of a linked list. Since your solution uses the “Cheshire Cat” approach to information hiding, you can actually put the implementation of class `Node` into the `Stack.cppp` file so clients of the `LStack` class won’t have access to the implementation at all!

```
template <typename T>
class Stack_Node
{
friend class Stack::LStack<T>;
public:
  // = Initialization methods

  // Create a stack node with value i and assign it’s next to point to next
  Stack_Node (const T &i,
              Stack_Node<T> *next = 0);

  // Default constructor.
  Stack_Node (void);

  // Copy constructor.
  Stack_Node (const Stack_Node<T> &n);

  // = Class-specific freelist management methods.

  // Class specific new which amortizes the cost of dynamic memory
  // allocation by maintaining a free list.
  void *operator new (size_t bytes);

  // Class specific operator delete which adds the deleted node to the
  // free list for use in future allocation operations.
  void operator delete (void *ptr);

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

private:
  // Pointer to next element in the list of Stack_Nodes.
  Stack_Node<T> *next_;

  // Current value of the item in this node.
  T item_;
};
```

Please note that your solution should meet at least the basic exception safety guarantees.

Extra Capabilities for Grad Students

If you signed up for this course as a CS 291 please do the following additional things:

- Implement a free list inside of `Stack_Node` that recycles allocated nodes and optimizes the creation of new nodes.
- Make sure that your solution has the *strong* exception safety guarantees¿

You can get the “shells” for the program from `www.cs.wustl.edu/~schmidt/cs215/assignment3` in the Makefile, the test drivers, and `AStack.h` and `LStack.h` files are written for you. You’ll need to edit the `AStack.cpp`, `LStack.cpp`, `AStack.inl`, and `LStack.inl` files to add the methods that implement the Stack ADT. You’ll also need to copy over your Array files from assignment two.