CS242: Object-Oriented Design and Programming

Program Assignment 6
Due Thursday, March 19th, 1998

This programming assignment focuses upon the use of inheritance and dynamic binding to build and test both *bounded* and *unbounded* implementations of the ADT *stack*. Both versions will inherit from the following *abstract base class*:

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

  // = Pure virtual methods.

  virtual ~Stack (void) = 0;
  // Destructor.

  virtual void push (const T &item) = 0;
  // Place a new item on top of the stack.

  virtual void pop (T &item) = 0;
  // Remove and return the top stack item.

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

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

  // = Template Methods
  void operator= (const Stack<T> &s);
  // Assignment operator (performs assignment).  This should be
  // implemented in Stack.C and *not* overridden by derived classes.

  void top (T &item) const;
  // Return top stack item without removing it.  This should be
  // implemented in Stack.C and *not* overridden by derived classes.

  int operator== (const Stack<T> &s) const;
  // Checks for Stack equality.  This should be
  // implemented in Stack.C and *not* overridden by derived classes.

  int operator!= (const Stack<T> &s) const;
  // Checks for Stack inequality.  This should be
  // implemented in Stack.C and *not* overridden by derived classes.

  virtual size_t size (void) const = 0;
  // Return the number of elements currently in the stack.

  virtual Iterator<T> *iterator (void) const = 0;
  // Return the (dynamically bound) iterator for this Stack.

protected:
  virtual void copy_elements (const Stack<T> &s);
  // Copy all the elements from <s> to <this>
  // using the Iterator pattern.

  virtual void delete_elements (void);
  // Delete all elements of <this>.
};
```

Note that most of the methods in this class are *pure virtual functions*, which means they must be provided by a subclass. In addition, there is no `private:` section in this class since there are no implementation details in the `Stack` abstract base class!

However, several methods (such as `operator=` and `top` can be implemented in the base class using the *Template Method* pattern. Here's how `operator=` can be implemented, for example:

```
// Assignment operator (performs assignment).
template <class T> void
Stack<T>::operator= (const Stack<T> &s)
{
  // Virtual calls.
  this->delete_elements ();
  this->copy_elements (s);
}
```

Naturally, there are other ways to implement this method, as well.

Given this abstract class, you will implement two versions that derive from `Stack`:

1. *Bounded* – The first version will use an array whose bounds are fixed at creation time. Implementing this program should be trivial since you can implement the bounded `Stack` class using your `Array` class.
2. *Unbounded* – The second version will use a linked list, which is "unbounded" (at least in principle...) and uses dynamic memory. Implementing this program will require you to implement an unbounded `Stack` class using a linked list of `Nodes`. It is not necessary to use the "free list" version of your unbounded stack, though you are certainly welcome to do so.

Part 1 – Bounded Stack

The first implementation you will write is a `Bounded_Stack` subclass of `Stack`:

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

template <class T>
class Bounded_Stack : public Stack<T>
  // = TITLE
  //     Implement a generic LIFO abstract data type.
  //
  // = DESCRIPTION
  //     This implementation of a Stack uses a bounded array.
{
```

```
public:
  typedef T TYPE;
  // C++ trait.

  // = Initialization, assignment, and termination methods.

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

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

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

  // = Classic Stack operations.

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

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

  // = Check boundary conditions for Stack operations.

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

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

  virtual size_t size (void) const;
  // Return the number of elements currently in the stack.

  virtual Iterator<T> *iterator (void) const;
  // Return the (dynamically bound) iterator for
  // this Bounded Stack.

protected:
  virtual void copy_elements (const Stack<T> &s);
  // Copy all the elements from <s> to <this>.

  virtual void delete_elements (void);
  // Delete all elements of <this>.

private:
  // You fill in here...
};
```

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

Part 2 – Unbounded Stack

A limitation of the bounded implementation of the Stack ADT is that a `Bounded_Stack` cannot grow beyond their initial size. Therefore, your second implementation you will write is an "unbounded" stack using dynamic memory. Note that this change only affects the stack representation, but does not affect the stack interface.

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

// Forward declaration.
template <class T> class Node;

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

  // = Initialization, assignment, and termination methods.

  Unbounded_Stack (size_t size_hint);
  // Initialize a new stack so that it is empty.

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

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

  // ... Same as for the Bounded_Stack

private:
  Node<T> *top_;
  // A pointer to the top of the stack.  Note
  // The use of the ‘‘Cheshire Cat’’ technique for
  // information hiding.
};
```

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).

  // = 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.
};
```

Iterators

The key to making your subclass implementations of `Stack` work effectively is to define an `Iterator` base class:

```
template <class T>
class Iterator
// = TITLE
//   Allows clients to access elements in
//   a container sequentially.
{
public:
  virtual void first (void) = 0;
  // Resets the iterator to the beginning.

  virtual void next (void) = 0;
  // Advance the iterator to the next item.

  virtual int is_done (void) const = 0;
  // True if we've seen all the items, else false.
  virtual T current_element (void) const = 0;
  // Return a const reference to the current item.

  virtual void set_current_element (const T &item) = 0;
  // Set the value of the current item.

  virtual ˜Iterator (void) = 0;
  // Virtual destructor ensures correct deletions.
};
```

Iterator is a pattern that allows clients to access elements in an aggregate collection sequentially without revealing the collection’s implementation details. To apply this pattern to the current assignment, you’ll need to subclass the base class `Iterator` to define classes that iterate over `Bounded_Stack` and `Unboundded_Stack` instances transparently.

```
template <class T>
class Bounded_Stack_Iterator : public Iterator<T>
{
  // ...
};

template <class T>
class Unbounded_Stack_Iterator : public Iterator<T>
{
  // ...
};
```

Naturally, the factory methods for the `iterator` methods on `Bounded_Stack` and `Unboundded_Stack` will return the appropriate concrete `Iterator`.

Test Driver Code

The following code implements a test driver to test your stack implementation:

```
/* -*- C++ -*- */
// Uses a stack to reverse a name.

#include <iostream.h>
#include <assert.h>
#include "Bounded_Stack.h"
#include "Unbounded_Stack.h"

const int MAX_NAME_LEN = 80;

// Factory that creates an appropriate Stack<char>
// subclass.

Stack<char> *
make_stack (int bounded)
{
  if (bounded)
    return new Bounded_Stack<char> (MAX_NAME_LEN);
  else
    return new Unbounded_Stack<char> (MAX_NAME_LEN);
}

int
main (int argc)
{
  char name[MAX_NAME_LEN];

  Stack<char> *s1 = make_stack (1);
  Stack<char> *s2 = make_stack (0);

  cout << "Please enter your name..: ";
  cin.getline (name, MAX_NAME_LEN);
  int readin = cin.gcount () - 1;

  for (int i = 0; i < readin && !s1->is_full (); i++)
    s1->push (name[i]);

  cout << "doing the assignment...\n";
  *s2 = *s1;
  assert (*s2 == *s1);
  cout << "done\n";

  cout << "your name backwards is..: ";

  while (!s1->is_empty ())
    {
      Stack<char>::TYPE c;
      s1->pop (c);
      cout << c;
    }

  cout << "\nand again...:\n";

  while (!s2->is_empty ())
    {
      Stack<char>::TYPE c;
      s2->pop (c);
      cout << c;
    }

  cout << endl;
  assert (s1->is_empty ());
  return 0;
}
```

Getting Started

You can get the “shells” and Makefile for part one of the program from your account on cec. These files are stored in `/project/adaptive/cs242/assignment-6/`. Here’s a script that shows you how to set everything up and get these files:

```
% cd ˜/cs242
% mkdir assignment-6
% cd assignment-6
% cp -r /project/adaptive/cs242/assignment-6/* .
% ls
stack-test.C
Stack.h
Bounded_Stack.h
Iterator.h
Node.h
Unbounded_Stack.h
```

You'll need to create the other files you need, *e.g.*, the Makefile and the `Array` class, etc. from your earlier assignments.