CS242: Object-Oriented Design and Programming

Program Assignment 2
Part 1 (Bounded Stack) Due Thursday, Feb 1st, 1996
Part 2 (Unbounded Stack) Due Tuesday, Feb 5th, 1996

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*, *is_empty* and *is_full*. This part of your programming assignment focuses upon building and using *bounded* and *unbounded* implementations of stacks.

You will implement and profile two versions of the ADT `Stack`:

1. *Bounded* – the first one will use an array whose bounds are fixed at creation time. Implementing this program should be trivial now that you've implemented the `Array` class.
2. *Unbounded* – the second one will use a linked list, which is "unbounded" (at least in principle...) and uses dynamic memory. This will be much more challenging to write correctly...

Part 1 – Bounded Stack

The first implementation you will write is a "bounded" stack. Your task is to implement the methods that operate upon objects of class `Stack`. Feel free to reuse the class `Array` you implemented for your first assignment. Here's the class declaration for `Stack`:

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

template <class T>
class Stack
  // = 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.

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

  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.

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 Stack implementation of the ADT Stack is that stacks 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.

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 "Stack.h"
```

```
int
main (void)
{
  const int MAX_NAME_LEN = 80;
  char name[MAX_NAME_LEN];

  Stack<char> s1 (MAX_NAME_LEN);

  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]);

  // Test the copy constructor.
  Stack<char> s2 (s1);
  assert (s1 == s2);

  // Test the assignment operator
  s1 = s2;
  assert (s1 == s2);

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

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

  cout << endl;
  assert (s1.is_empty ());
  assert (!s2.is_empty ());
  assert (s1 != s2);
  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-3/Stack/`. Here’s a script that shows you how to set everything up and get these files:

```
% cd ~/cs242
% mkdir assignment-3
% cd assignment-3
% cp -r /project/adaptive/cs242/assignment-3/Stack/* .
% ls
Makefile
stack-test.C
Stack.h
Stack.C
% make
```

The `Makefile`, `stack-test.C` and `Stack.h` files are written for you. All you need to do is edit the `Stack.C` files to add the methods that implement the bounded stack.

I'll put the shells for part 2 out shortly.