```
/* HEADER FILE - LIST IMPLEMENTATION OF A STACK */

#include "underflowoverflow.h"

template <class Object>
class Stack
{
  public:

    // Constructor
    Stack( );

    // Push x on the stack
    void push( const Object & x );

    // Pop top item from the stack - error if empty
    void pop( ) throw(Underflow);

    // Access top item on the stack - error if empty
    const Object & top( ) const throw(Underflow);

    // Remove and return top item - error if empty
    Object topAndPop( ) throw(Underflow);
```

```
    // Test to see whether stack is empty
    bool empty( ) const;

    // Test to see whether stack is full - never true for list
    bool full( ) const;

    // Make the stack empty, discarding all items on it
    void makeEmpty( );

    // Methods needed for list - called automatically by compiler
    // Not normally called by user code

    Stack( const Stack & rhs );
    ~Stack( );
    const Stack & operator=( const Stack & rhs );

  private:
    class ListNode;
    ListNode *topOfStack;
};
```

```
/* IMPLEMENTATION FILE - LIST IMPLEMENTATION OF A STACK */

#include "stackl.h"

#ifndef NULL
#define NULL 0
#endif

template <class Object>
class Stack<Object>::ListNode
{
    Object element;
    ListNode *next;
    ListNode(const Object & theElement, ListNode * n = NULL)
    : element( theElement ), next( n ) { }
};
```

```
template <class Object>
Stack<Object>::Stack( )
{
    topOfStack = NULL;
}
```

```
template <class Object>
void Stack<Object>::push( const Object & x )
{
    topOfStack = new ListNode( x, topOfStack );
}
```

```
template <class Object>
void Stack<Object>::pop( ) throw(Underflow)
{
    if( empty( ) )
        throw new Underflow( );

    ListNode *oldTop = topOfStack;
    topOfStack = topOfStack->next;
    delete oldTop;
}
```

```
template <class Object>
const Object & Stack<Object>::top( ) const throw(Underflow)
{
    if( empty( ) )
        throw Underflow( );
    return topOfStack->element;
}
```

```
template <class Object>
Object Stack<Object>::topAndPop( ) throw(Underflow)
{
    Object topItem = top( );
    pop( );
    return topItem;
}
```

```
template <class Object>
bool Stack<Object>::empty( ) const
{
    return topOfStack == NULL;
}
```

```
template <class Object>
bool Stack<Object>::full( ) const
{
    return false;
}
```

```
template <class Object>
void Stack<Object>::makeEmpty( )
{
    // Existing nodes must be recycled
    while( !empty( ) )
        pop( );
}
```

```
template <class Object>
Stack<Object>::Stack( const Stack<Object> & rhs )
{
    topOfStack = NULL;
    *this = rhs;    // A deep copy will be done by code for
                    // operator =
}
```

```
template <class Object>
Stack<Object>::~Stack( )
{
    makeEmpty( );   // This will discard all existing nodes
}
```

```
template <class Object>
const Stack<Object> & Stack<Object>::
operator=( const Stack<Object> & rhs )
{
    if( this != &rhs )
    {
        // Assignment requires discarding all existing nodes, and
        // then making a deep copy

        makeEmpty( );
        if( rhs.empty( ) )
            return *this;

        ListNode *rptr = rhs.topOfStack;
        ListNode *ptr  = new ListNode( rptr->element );
        topOfStack = ptr;

        for( rptr = rptr->next; rptr != NULL; rptr = rptr->next )
            ptr = ptr->next = new ListNode( rptr->element );
    }
    return *this;
}
```