```
// Public Interface Header file for the ADT stack.
typedef desired-type-of-stack-item stackItemType;

class stackClass
{
public:
   stackClass();                        // default constructor
   stackClass(const stackClass &);      // copy constructor
   ~stackClass();                       // destructor

   bool StackIsEmpty() const;
   bool StackIsFull() const;
   // Determines whether a stack is empty (or full).
   // Precondition: None.
   // Postcondition: Returns true if the stack is empty (or full);
   //    otherwise returns false.

   void Push(stackItemType NewItem, bool& Success);
   // Adds an item to the top of a stack.
   // Precondition: NewItem is the item to be added.
   // Postcondition: If insertion was successful, NewItem
   //    is on the top of the stack and Success is true;
   //    otherwise Success is false.

   void Pop(bool& Success);
   // Removes the top of a stack.
   // Precondition: None.
   // Postcondition: If the stack was not empty, the item
   //    that was added most recently is removed and Success
   //    is true. However, if the stack was empty, deletion is
   //    impossible and Success is false.

   void Pop(stackItemType& StackTop, bool& Success);
   // Retrieves and removes the top of a stack.
   // Precondition: None.
   // Postcondition: If the stack was not empty, StackTop
   //    contains the item that was added most recently, the
   //    item is removed, and Success is true. However, if the
   //    stack was empty, deletion is impossible, StackTop is
   //    unchanged, and Success is false.

   void GetStackTop(stackItemType& StackTop, bool& Success) const;
   // Retrieves the top of a stack.
   // Precondition: None.
   // Postcondition: If the stack was not empty, StackTop
   //    contains the item that was added most recently and
   //    Success is true. However, if the stack was empty, the
   //    operation fails, StackTop is unchanged, and Success
   //    is false. The stack is unchanged.
private:
```

```
// Sample "stack" application

#include <iostream.h>
#include <assert.h>

// Assume stackItemType is 'char'

#include "Stack.h"    // Some version of the stack header
                      // e.g, StackL.h, StackA.h, StackP.h

int main()
{
   stackItemType AnItem;
   stackClass    S;
   bool          Success;

   // Read in all the chars on one input line
   cin.get(AnItem);                 // read first item
   while(AnItem != '\n')
   {
      S.Push(AnItem, Success);      // push it onto stack
      assert(Success);              // Optional

      cin.get(AnItem);              // read next item
   }

   cout << endl;

   while( ! S.StackIsEmpty() )
   {
      S.Pop(AnItem, Success);       // Get LIFO item
      assert(Success);              // Optional

     cout << AnItem;           // Write out next item
   }

   cout << endl;
   cout << "End of Sample Stack Application" << endl;

   return(0);
}
```

```
// *********************************************************
// Header file StackL.h for the ADT stack.
// ADT list implementation.
// *********************************************************

#include "ListP.h"      // list operations

typedef listItemType stackItemType;

class stackClass
{
public:
// constructors and destructor:
   stackClass();                        // default constructor
   stackClass(const stackClass& S);     // copy constructor
   ~stackClass();                       // destructor

// stack operations:
   bool StackIsEmpty() const;
   void Push(stackItemType NewItem, bool& Success);
   void Pop(bool& Success);
   void Pop(stackItemType& StackTop, bool& Success);
   void GetStackTop(stackItemType& StackTop, bool& Success) const;

private:

   // list of stack items
   listClass L;

};
```

```
// *********************************************************
// Implementation file StackL.cpp for the ADT stack.
// ADT list implementation.
// *********************************************************

#include "StackL.h"    // header file

stackClass::stackClass()
{
}  // end default constructor

stackClass::stackClass(const stackClass& S): L(S.L)
{
}  // end copy constructor

stackClass::~stackClass()
{
}  // end destructor

bool stackClass::StackIsEmpty() const
{
   return bool(L.ListLength() == 0);
}

void stackClass::Push(stackItemType NewItem, bool& Success)
{
   L.ListInsert( 1, NewItem, Success);
}

void stackClass::Pop(bool& Success)
{
   L.ListDelete( 1, Success);
}

void stackClass::Pop(stackItemType& StackTop, bool& Success)
{
   L.ListRetrieve( 1, StackTop, Success);
   L.ListDelete( 1, Success);
}

void stackClass::GetStackTop(stackItemType& StackTop,
                             bool& Success) const
{
   L.ListRetrieve( 1, StackTop, Success);
}
```

```
// ********************************************************
// Header file StackA.h for the ADT stack.
// Static-allocation Array-based implementation.
// ********************************************************

const int MAX_STACK = maximum-size-of-stack;

typedef desired-type-of-stack-item stackItemType;

class stackClass
{
public:
   stackClass();  // default constructor

   // copy constructor supplied by the compiler
   // destructor supplied by the compiler

   bool StackIsEmpty() const;
   void Push(stackItemType NewItem, bool& Success);
   void Pop(bool& Success);
   void Pop(stackItemType& StackTop, bool& Success);
   void GetStackTop(stackItemType& StackTop, bool& Success) const;

private:
   // array of stack items
   stackItemType Items[MAX_STACK];

   // index to top of stack
   int           Top;
};
```

```
// ********************************************************
// Implementation file StackA.cpp for the ADT stack.
// Static-Allocation Array-based implementation.
// ********************************************************
#include "StackA.h"  // header file

stackClass::stackClass(): Top(-1)
{  }

bool stackClass::StackIsEmpty() const
{
    return bool(Top < 0);
}

void stackClass::Push(stackItemType NewItem, bool& Success)
{
   Success = bool(Top < MAX_STACK - 1);
   if (Success)                      // Stack is not full
   {
       ++Top;                        // push top
       Items[Top] = NewItem;         // store top
   }
}

void stackClass::Pop(bool& Success)
{
   Success = bool(!StackIsEmpty());
   if (Success)                      // Stack is not empty
       --Top;                        // pop top
}

void stackClass::Pop(stackItemType& StackTop, bool& Success)
{
   Success = bool(!StackIsEmpty());
   if (Success)                      // Stack is not empty
   {
       StackTop = Items[Top];        // retrieve top
       --Top;                        // pop top
   }
}

void stackClass::GetStackTop(stackItemType& StackTop,
                             bool& Success) const
{
   Success = bool(!StackIsEmpty());
   if (Success)                      // Stack is not empty
       StackTop = Items[Top];        // retrieve top
}
```

```
// *********************************************************
// Header file StackAD.h for the ADT stack.
// Dynamic-allocation Array-based implementation.
// *********************************************************

const int DEFAULT_MAX_STACK = maximum-size-of-stack;

typedef desired-type-of-stack-item stackItemType;

class stackClass
{
public:
   // default constructor
   stackClass(const int MaxStk = DEFAULT_MAX_STACK);

   stackClass(const stackClass& S);  // copy constructor
   ~stackClass();                    // destructor

   bool StackIsEmpty() const;
   void Push(stackItemType NewItem, bool& Success);
   void Pop(bool& Success);
   void Pop(stackItemType& StackTop, bool& Success);
   void GetStackTop(stackItemType& StackTop, bool& Success) const;

private:
   // array of stack items
   stackItemType *Items;

   // Maximum size of stack: Set in constructor
   int            MAX_STACK;

   // index to top of stack
   int            Top;
};
```

```
// ********************************************************
// Implementation file StackAD.cpp for the ADT stack.
// Dynamic-Allocation Array-based implementation.
// ********************************************************
#include <stddef.h>  // for NULL
#include <assert.h>  // for assert

#include "StackAD.h"  // header file

stackClass::stackClass(const int MaxStk)
   : Top(-1), MAX_STACK(MaxStk)
{
   Items = new stackItemType[MAX_STACK];
   assert(Items != NULL);
}

stackClass::~stackClass()
{
   delete [ ] Items;
}

stackClass::stackClass(const stackClass& S)
   : Top(S.Top), MAX_STACK(S.MAX_STACK)
{
   // Allocate a new copy of the Items array
   Items = new stackItemType[MAX_STACK];
   assert(Items != NULL);

   // Copy the Items array contents
   for(int i=0; i <= Top; ++i)
      Items[i] = S.Items[i];
}


// Rest of the stackClass methods can be the same
// as for the Static array implementation
//  (Stack can still get full, etc.)
```

```
// StackAD.h  PUSH Method
// Instead of returning Success = FALSE if the
// current stack is full (e.g. Top >= MAX_STACK - 1)
//
// it is EASY to modify the "PUSH" method to
// automatically 'grow' the stack, if necessary
```

```
void stackClass::Push(stackItemType NewItem, bool& Success)
{
   Success = bool(Top < MAX_STACK - 1);

   if ( ! Success)              // Existing Stack is full
   {
      // Make stack twice as big
      int NewStackSize = 2 * MAX_STACK;

      // Allocate new dynamic array
      stackItemType *p = new stackItemType[NewStackSize];

      Success = bool(p != NULL);

      if(Success)           // New allocation succeeded
      {
         // Copy contents of old array into new array
         for(int i=0; i <= Top; ++i)
            p[i] = Items[i];

         // Deallocate the original array
         delete [ ] Items;

         // Update this stackClass object bookkeeping
         Items = p;                      // Update array ptr
         MAX_STACK = NewStackSize;       // Update Max
      }
   }

   if (Success)                  // Stack is not full
   {
      ++Top;                     // push top
      Items[Top] = NewItem;      // store top
   }
}
```

```
// ********************************************************
// Header file StackP.h for the ADT stack.
// Node-Linked Pointer-based implementation.
// ********************************************************

typedef desired-type-of-stack-item stackItemType;

struct stackNode;  // defined in implementation file

typedef stackNode* ptrNode;  // pointer to node

class stackClass
{
public:
   stackClass();                        // default constructor
   stackClass(const stackClass& S);     // copy constructor
   ~stackClass();                       // destructor

   bool StackIsEmpty() const;
   void Push(stackItemType NewItem, bool& Success);
   void Pop(bool& Success);
   void Pop(stackItemType& StackTop, bool& Success);
   void GetStackTop(stackItemType& StackTop, bool& Success) const;

private:

   // points to top of stack
   ptrNode   TopPtr;

};
```

```
// ********************************************************
// Implementation file StackP.cpp for the ADT stack.
// Node-Linked Pointer-based implementation.
// ********************************************************
#include <stddef.h>  // for NULL
#include <assert.h>  // for assert

#include "StackP.h"  // header file

struct stackNode
{
   stackItemType Item;
   ptrNode       Next;
};

stackClass::stackClass() : TopPtr(NULL)    // constructor
{
}

stackClass::~stackClass()                  // destructor
{
   bool Success;

   // pop until stack is empty (Success is false)
   Pop(Success);

   while (Success)
      Pop(Success);
}

bool stackClass::StackIsEmpty() const
{
   return bool(TopPtr == NULL);
}

void stackClass::GetStackTop(stackItemType& StackTop,
                             bool& Success) const
{
   Success = bool( ! StackIsEmpty());

   if (Success)                            // Stack is not empty
      StackTop = TopPtr->Item;             // retrieve top
}
```

```
 void stackClass::Push(stackItemType NewItem, bool& Success)
{
    // create a new node
    ptrNode  NewPtr = new stackNode;

    Success = bool(NewPtr != NULL);   // check allocation
    if (Success)
    {
        // set data portion of new node
        NewPtr->Item = NewItem;

        // insert the new node
        NewPtr->Next = TopPtr;
        TopPtr = NewPtr;
    }
}

void stackClass::Pop(bool& Success)
{
    Success = bool(!StackIsEmpty());
    if (Success)                              // Stack is not empty
{
        ptrNode  Temp = TopPtr;               // delete top
        TopPtr = TopPtr->Next;

        // return deleted node to system
        Temp->Next = NULL;  // safeguard
        delete Temp;
    }
}

void stackClass::Pop(stackItemType& StackTop, bool& Success)
{
    Success = bool(!StackIsEmpty());
    if (Success)                              // Stack is not empty
    {
        StackTop = TopPtr->Item;              // retrieve top

        ptrNode  Temp = TopPtr;               // delete top
        TopPtr = TopPtr->Next;

        // return deleted node to system
        Temp->Next = NULL;  // safeguard
        delete Temp;
    }
}
```

```
stackClass::stackClass(const stackClass& S)  // copy constructor
{
   if (S.TopPtr == NULL)
      TopPtr = NULL;  // original list is empty

   else
   {
      // copy first node
      TopPtr = new stackNode;
      assert(TopPtr != NULL);

      TopPtr->Item = S.TopPtr->Item;

      // copy rest of list
      ptrNode  NewPtr = TopPtr;     // new list pointer

      for (ptrNode OrigPtr = S.TopPtr->Next;
                   OrigPtr != NULL;
                   OrigPtr = OrigPtr->Next)
      {
         NewPtr->Next = new stackNode;
         assert(NewPtr->Next != NULL);

         NewPtr = NewPtr->Next;
         NewPtr->Item = OrigPtr->Item;
      }

      NewPtr->Next = NULL;
   }
}
```