CSCI 2170 **Lecture Notes (13)**

ADT Stack

Operations:

- Create an empty stack
- Destroy a stack
- Determine whether a stack is empty
- Add a new item to the stack
- Remove from the stack the item that was added most recently
- Retrieve from the stack the item that was added most recently

- ***createStack()***
- ***destroyStack()***
- ***isEmpty()*** *// Determines whether a stack is empty.*
- ***push(StackItemType newItem, bool & success****);* *// Adds an item to the top of a stack.*
- ***pop(bool & success);*** *// Removes the top of a stack.*
- ***pop(StackItemType& stackTop, bool & success****);// Retrieves and removes the top of a stack.*
- ***getTop(StackItemType& stackTop, bool & success);*** *// Retrieves the top of a stack.*

Applications of ADT Stack

(1) Read and correct with backspace: reads the input line, for each character read, either enter it into stack S, if it is '←', correct the content of S

```
ReadAndCorrect(Stack S)
{
        success = true;
        S.CreateStack();
        Read newChar;

        while (newChar is not the end of line symbol && success)
        {
                if (newChar is not '←')
                        S.Push(newChar, success);
                Else if  (!S.IsEmpty())
                        S.Pop(success);

                Read newChar;
        }
}
```

(2) write content of stack : directly poping out the content of stack will display the letters in the word in reverse order

```
void DisplayBackward(Stack S)
```

```
{
        while (!S.IsEmpty())
        {
                S.pop(newChar, success);
                Write newChar;
        }
}
```

?? How to write out the content of the stack in the right order?

```
Void DisplayForward (Stack S)
{
        Stack   tmpS;
        While (!S.IsEmpty())
        {
                S.pop(newChar, success);
                tmpS.push(newChar, success);
        }

        DisplayBackward(tmpS);
}
```

(3) Checking for balanced braces

- each time a ‘{‘ is encountered, push it into the stack
- each time a ‘}’ is entered, it is matched to an already encountered ‘{‘, pop stack
- **Balanced :** when reaching the end of the string, all the ‘{‘ has been matched against (stack is empty)
- **NOT balanced** :
  1. when a ‘}’ is entered, there is no existing ‘{‘ to match, OR
  2. when reaching the end of the string, there are still some ‘{‘ not being matched (stack not empty)

```
index = 0;
bool balanced = true, success = false;
Stack braces;
While (balanced && index < strlen(program))
{
        ch = program [index];
        index ++;

        if (ch == ‘{‘)
                braces.Push(ch, success);
        else if (ch == ‘}’)
        {
                braces.Pop(success);
```

```
            if (!success)
                balanced = false;
        }
}
if (balanced && braces.IsEmpty())
        cout << “The braces in this program are balanced.” << endl;
else
        cout << “Syntax error: Braces are NOT balanced.” << endl;
```

```
// ********************************************************
// Header file StackA.h for the ADT stack.
// Array-based implementation.
// ********************************************************
const int MAX_STACK = maximum-size-of-stack;
typedef desired-type-of-stack-item StackItemType;

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

// stack operations:
   bool isEmpty() const;
   // Determines whether a stack is empty.
   // Precondition: None.
   // Postcondition: Returns true if the stack is empty; 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 the insertion is successful, newItem is on the top of the stack.

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

   void pop(StackItemType& stackTop, bool & success);
   // Retrieves and removes the top of a stack.
   // Precondition: None.
   // Postcondition: If the stack is not empty, stackTop contains the item that was added
   // most recently and the item is removed. However, if the stack is empty, deletion is
   // impossible and stackTop is unchanged.

   void getTop(StackItemType& stackTop, bool & success);
   // Retrieves the top of a stack.
   // Precondition: None.
   // Postcondition: If the stack is not empty, stackTop contains the item that was added
   // most recently. However, if the stack is empty, the operation fails and stackTop is
   // unchanged. The stack is unchanged.

private:
   StackItemType items[MAX_STACK]; // array of stack items
   int          top;            // index to top of stack
```

```
}; // end class
// End of header file.

// ********************************************************
// Implementation file StackA.cpp for the ADT stack.
// Array-based implementation.
// ********************************************************
#include "StackA.h" // Stack class specification file

Stack::Stack(): top(-1)
{
} // end default constructor

bool Stack::isEmpty() const
{
  return top < 0;
} // end isEmpty

void Stack::push(StackItemType newItem. bool & success)
{
  success = true;
// if stack has no more room for another item
  if (top >= MAX_STACK-1)
        success = false;
  else
  {  ++top;
    items[top] = newItem;
  } // end if
} // end push

void Stack::pop(bool & success)
{
  success = true;
  if (isEmpty())
   success = false;
  else
    --top; // stack is not empty, pop top
} // end pop

void Stack::pop(StackItemType& stackTop, bool &success)
{
  success = true;
  if (isEmpty())
    success = false;
  else
  {  // stack is not empty, retrieve top
```

```
    stackTop = items[top];
    --top; // pop top
  } // end if
} // end pop

void Stack::getTop(StackItemType& stackTop, bool & success) const
{
  success = true;
  if (isEmpty())
    success = false;
  else
    //  stack is not empty; retrieve top
    stackTop = items[top];
} // end getTop
```

Pointer based implementation of ADT stack

```
// *********************************************************
// Header file StackP.h for the ADT stack.
// Pointer-based implementation.
// *********************************************************
#include "StackException.h"
typedef desired-type-of-stack-item StackItemType;

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

// stack operations:
  bool isEmpty() const;
  void push(StackItemType newItem);
  void pop(bool & success);
  void pop(StackItemType& stackTop, bool & success);
  void getTop(StackItemType& stackTop, bool & success) const;

private:
  struct StackNode // a node on the stack
  {
    StackItemType item; // a data item on the stack
    StackNode *next;  // pointer to next node
  }; // end struct
```

```
   StackNode *topPtr; // pointer to first node in the stack
}; // end Stack class
// End of header file.

// *********************************************************
// Implementation file StackP.cpp for the ADT stack.
// Pointer-based implementation.
// *********************************************************
#include "StackP.h" // header file
#include <cstddef>  // for NULL
#include <cassert>  // for assert

Stack::Stack() : topPtr(NULL)
{
} // end default constructor

Stack::Stack(const Stack& aStack)
{
   if (aStack.topPtr == NULL)
      topPtr = NULL; // original list is empty

   else
   {  // copy first node
      topPtr = new StackNode;
      assert(topPtr != NULL);
      topPtr->item = aStack.topPtr->item;

      // copy rest of list
      StackNode *newPtr = topPtr; // new list pointer
      for (StackNode *origPtr = aStack.topPtr->next;
                origPtr != NULL;
                origPtr = origPtr->next)
      {  newPtr->next = new StackNode;
         assert(newPtr->next != NULL);
         newPtr = newPtr->next;
         newPtr->item = origPtr->item;
      } // end for

   newPtr->next = NULL;
   } // end if
} // end copy constructor

Stack::~Stack()
{
   // pop until stack is empty
   while (!isEmpty())
```

```
    pop();
  // Assertion: topPtr == NULL
} // end destructor

bool Stack::isEmpty() const
{
  return topPtr == NULL;
} // end isEmpty

void Stack::push(StackItemType newItem, bool & success)
{
  success = true;
  // create a new node
  StackNode *newPtr = new StackNode;

  if (newPtr == NULL) // check allocation
      success = false;
  else
  {  // allocation successful; set data portion of new node
    newPtr->item = newItem;
    // insert the new node
    newPtr->next = topPtr;
    topPtr = newPtr;
  } // end if
} // end push

void Stack::pop(bool & success)
{
  success = true;
  if (isEmpty())
    success = false;
  else
  {  // stack is not empty; delete top
    StackNode *temp = topPtr;
    topPtr = topPtr->next;
    // return deleted node to system
    temp->next = NULL; // safeguard
    delete temp;
  } // end if
} // end pop

void Stack::pop(StackItemType& stackTop, bool & success)
{
  success = true;
  if (isEmpty())
      success = false;
```

```
  else
  {  // stack is not empty; retrieve and delete top
     stackTop = topPtr->item;
     StackNode *temp = topPtr;
     topPtr = topPtr->next;

     // return deleted node to system
     temp->next = NULL; // safeguard
     delete temp;
  } // end if
} // end pop

void Stack::getTop(StackItemType& stackTop, bool & success) const
{
  success = true;
  if (isEmpty())
       success = false;
  else
     // stack is not empty; retrieve top
     stackTop = topPtr->item;
} // end getTop
// End of implementation file.
```