Storage for a stack can be allocated and reclaimed as individual elements or storage can be allocated for a group of elements and reclaimed as a group. In this note we will allocate storage for a group of elements. An array is used to store the elements of a stack.

Since this course specifically addresses the management of storage for objects, we allocate and reclaim storage for the stack dynamically.

To manage the stack, three variables are required.

1. *size*: Variable *size* retains the maximum number of elements available for the stack.
2. *tos*: Variable *tos* records the index of the element on top of the stack.
3. *S*: Variable *S* points to an array allocated to store elements on the stack.

We organize the foregoing variables as **private** members of **class** *Stack*. A pictorial representation of **class** *Stack* is given in Figure 1.

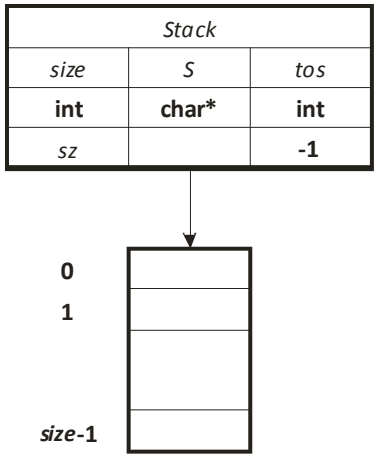

Figure 1. Structure of **class** *Stack*

Successive characters are pushed on the stack moving downward from element zero (0). Member *tos* records the index of the element on top of the stack. Member *tos* is incremented *before* an element is stored on the stack and decremented *after* an element is removed from the stack. Finding an initial value for member *tos* is determined by noting that element zero (0) must store the value stored on the stack first. The maximum number of elements available for the stack can be parameterized at the time the stack is declared. The value of variable *sz*, containing the maximum number of available elements, is assigned to member size when the stack is initialized.

```
struct StackException {
    StackException(char* m)
    {   cout << endl << "I am the Stack and I am " << m << "." << endl;
    }
};
class Stack {
    int size;                   //Number of available elements
    int tos;                    //Index of the element on top of the stack
    char* S;                    //Points to storage for the stack
public:
    Stack(int sz=100);          //Constructor
    ~Stack();                   //Destructor
    bool IsFull(void);          //Is the Stack full?
    bool IsEmpty(void);         //Is the Stack empty?
    void Push(char v);          //Put v on top of the stack
    char Pop(void);             //Return the character on top of the stack
};
```

Figure 2. class *Stack.*

Member functions:

1. ***Stack*(int *sz*)** Member function *Stack* is the constructor. The construction is called when an object of type *Stack* is declared.
   1.1. Initialize member *size* to the value stored in parameter *sz*.
   1.2. Initialize member *tos* to **–1**. Negative one is chosen as a value that when incremented will yield zero, the index of the first available element in array *S*.
   1.3. Allocate storage referenced by member *S*.

2. **~*Stack*()** Member function ~*Stack* is the destructor. The destruction is called when control for an object of type *Stack* goes out of the scope where the object was declared.
   2.1. Reclaim storage for the stack if, indeed, it was allocated.

3. **bool *IsFull*(void)** Member function *IsFull* determines if the stack is full. Element index values i are: $0 \le i \le size - 1$. The stack is full when $tos \ge size - 1$.
   3.1. Return **true** when member $tos \ge size - 1$; otherwise return **false**.

4. **bool *IsEmpty*(void)** Member function *IsEmpty* determines if the stack is empty. Element index values i are: $0 \le i \le size - 1$. The stack is empty when $tos < 0$.
   4.1. Return **true** when member $tos < 0$; otherwise return **false**.

5. **void *Push*(char *v*)** Member function *Push* places the value of parameter *v* on top of the stack.
   5.1. Throw exception *StackException* if the stack is full.
   5.2. At least one element is available to store the value of parameter *v*; otherwise control would have passed to the calling function.
   5.3. Increment member *tos* to the index of the next available element.

5.4. Store the value of parameter *v* in array *S* using *tos* as the index of the next available element.

6. **char** *Pop***(void)** Member function *Pop* returns the value on top of the stack.
   6.1. Throw exception *StackException* if the stack is empty.
   6.2. At least one element can be removed from the stack.
   6.3. Obtain a copy of the element on top of the stack.
   6.4. Decrement member *tos* to the index of the previous element stored on the stack.
   6.5. Return the copy of the element on top of the stack.

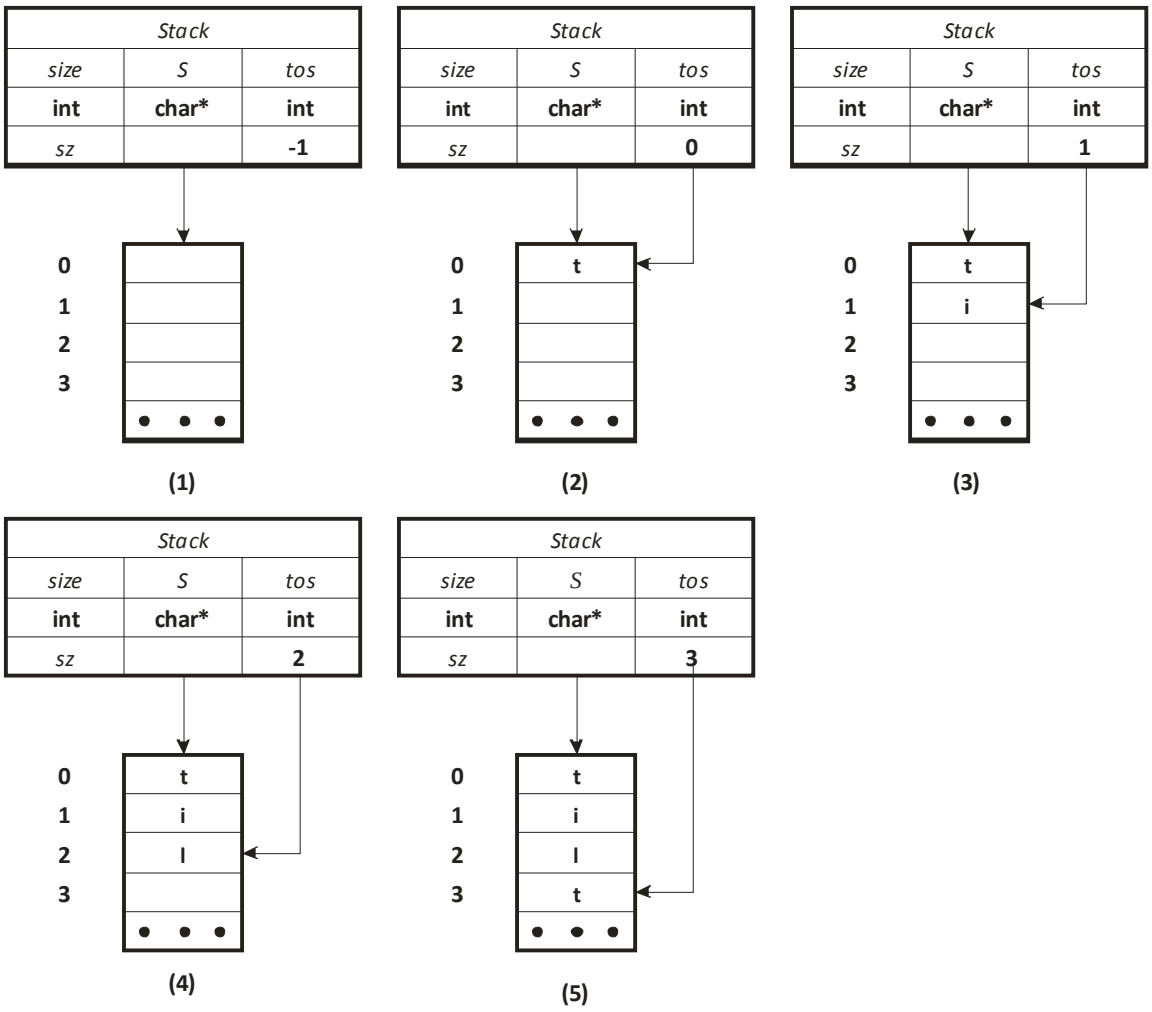

Figure 3. Successive snap shots of *Stack S* showing the characters of *string* "tilt"

Diagrams in Figure 3 show *Stack S* as the characters of *string* "tilt" are stored on it.

Exercises:

1. Modify file **Stack.h** given in this note so that it stores values of type **double**. Write a program that will evaluate a postfix string. For example the postfix string "3 4 5 * +" evaluates to 60.

2. A queue is a first-in-first-out (FIFO) data structure. Design a dequeue (pronounced “deck”) that implements both a stack and a queue.