**Problem 1 follows…**  
**Description:** Problem 1 is designed to demonstrate my knowledge of stacks. It consists of functions for initializing, deleting, placing at front/end, swapping and removing data from the double linked list. There are also get front/index and length functions. AllocationCount will tell how many things are currently dynamically allocated.

**Answers to Questions:**

1. An abstract data type allows the user to manipulate a data type without any knowledge of how the data type is being represented. It means that the specifics of the user data is separate from the implementation of the abstract data type. For example, a stack is an abstract data type. It can manipulate different data types using an array or linked list. Because the operations are hidden from the user to deal with data types, all the user knows is what operations can be performed and what those operations are supposed to do.
2. A stack is a linear data structure that has items placed and removed inside it, such that the most recently added item will be the next to be removed. This is called a last in-first out (LIFO) data structure. A FIFO means that the first item (not necessarily most recent) placed inside gets removed first. Both examples can add or remove data, but the difference is in which data gets removed when a pop is called.
3. Anything which is stowed on top of objects of the same type, like a stack of plates, spoons, t-shirts, or chairs. I will use the example of plates. The abilities of the stack are to be able to initialize a stack. If you own a cabinet, initializing the stack is like clearing out space for as many plates that you need to have lying on top of one another. Also, the stack could help you identify if it is empty. If you open your cabinet and see no plates stacked on top of one another then the stack is empty. If you open your cabinet and see a single plate or more, then the stack is not empty. This helps identify whether you could take a plate off the stack, i.e. call a pop. You could also get the number of items in the stack to give you an idea of how many pops you can call before the stack is empty. A pop requires, in the case of a LIFO stack, to remove the most recently placed plate and remove it from the stack. Imagine taking the first plate off the top of the rest. The last plate stacked on top is the first plate someone reaches for to take it out of the stack for use at dinner time. A stack also lets you check the top item in the stack without removing it, called a peek. If you wanted to see if the plate on top inside your cabinet is your favorite plate or not, you might call a peek. A stack can also tell you if it is full or not, to identify whether another push could occur. A push will update the top with another item. So when you push a new plate, another plate will be added to the top of the stack of plates inside your cabinet.
4. Arrays are set to a fixed size which means the push must be able to determine if there is enough room in the stack for another item. If there is, place the item in the stack and update the "top" to signal it is the most recently added. If there is not enough room, the size of the array would have to be changed. This cannot be done in C, so it means creating a new array and allocating memory for it, copying the data to the new array and deleting the old memory.

A linked list does not have the same fixed-size constraint that arrays have. If there is a need to push, new items can be added to a linked list if allocation/deallocation of link nodes is done. This is because the top/bottom are not indices but a connection of nodes.

1. Since arrays are fixed size, when a pop occurs and an item is removed, the array will still have the same length. This requires the array to be resized so that there are no "holes" after removing an element. Arrays cannot be resized in C so a new array would have to be created and allocated memory for, with data copied to the new array and the old memory deleted.

A linked list can pop an element. Since the top/bottom are connections of nodes instead of indices if allocation/deallocation is done once every pop is done.

**Problem 2 follows…**  
**Description:** Problem 2 is designed to demonstrate my knowledge of stack implementation. It consists of functions and definitions for the double linked list. It also contains stack functions for initializing, checking if the stack is empty, push/pop, peek and deleting the stack. AllocationCount will tell how many things are currently dynamically allocated.

**Program Output:**

On startup, #allocations is 0

After initStack called, #allocations is 2

push called, data is 1000, #allocations is 3

push called, data is 2000, #allocations is 4

push called, data is 3000, #allocations is 5

push called, data is 4000, #allocations is 6

peek called, data is 4000, #allocations is 6

pop called, data is 4000, #allocations is 5

peek called, data is 3000, #allocations is 5

pop called, data is 3000, #allocations is 4

peek called, data is 2000, #allocations is 4

pop called, data is 2000, #allocations is 3

peek called, data is 1000, #allocations is 3

pop called, data is 1000, #allocations is 2

Before deleteStack called, #allocations is 2

After deleteStack called, #allocations is 0

**Answers to Questions:**

* 1. initStack first created a stack structure and initialized its data contents. It also dynamically allocated memory for it, so AllocationCount increased to 1. initStack then created a double linked list and initialized its data contents. Doing so, it dynamically allocates memory for it. This caused AllocationCount to increase to 2. So, the AllocationCount reflects dynamic memory allocation of the stack and double linked list.
  2. The calls to push checked that the stack was not equal to NULL. Since both a stack and double linked list existed after initStack, data then gets added into the double linked list at the front. Then the stack is set to "not empty." If you were to check if the stack were empty, it would return false. The first-time push is called while the double linked list is empty, the data is assigned to both the head and the tail. After which, data is simply added to the double linked list as the new head. This happens four times because UserData has four elements. AllocationCount increased by 1 each time a push is called because the double linked list must create a new node to place it in the list. This updates the linkage from the new node to the next and previous nodes if there are any. Initializing a new node will allocate memory for it, which increases AllocationCount by 1.
  3. The stack contains a boolean called empty to return whether the stack is empty (true) or not empty (false). The while loop that does a peek and pop reads the value of the Boolean called empty to know when the entire stack has been processed. Empty is originally set to true when initStack is called. Empty is set to false whenever a push is called because a data node is added to the linked list. Pop sets empty to true or false based on the condition that the linked list contains 1 data set. Peek checks whether the stack is not NULL and not empty to return a valid front node. Since empty will be false at the beginning of the loop and true at the end of the loop’s condition after pop has deleted all nodes in the linked list, the while loop can start and complete fired logic properly.
  4. The order of the data values peeked and popped make sense because the stack is of last in-first out (LIFA) type. This means the most recently added item will be the next to be removed. First, push is called adding 1000 as the head and tail. Then another push is called adding 2000 as the next head (or front) so that the stack is {2000, 1000}. Then two more pushes are called to add 3000 then 4000 making the stack {4000, 3000, 2000, 1000}. The sequence of peeks and pops starts with the most recently added, 4000. First peek will display 4000 because it is the front or head of the stack. Then it gets deleted because pop is called to remove the front node. This is repeated as many times as needed until the stack is empty. After the first peek and pop, the stack will be {3000, 2000, 1000}. Peek will display 3000 because it is the new front or head of the stack. Then it gets deleted because pop is called to remove the front node. After which the stack will be {2000, 1000}. This occurs two more times until the stack is empty. AllocationCount changed as data was popped but not when peeked because peek will only return a valid front node if there is one. Pop is the function responsible for deleting a node which requires the release of dynamically allocated memory for that node. Once the node is deleted, memory is freed and AllocationCount decreases by 1.
  5. It makes sense that AllocationCount was 2 after the stack data was popped. Pops were called until the stack was empty, decreasing AllocationCount by 1 each time. During which, nodes get deleted hence the decrease of AllocationCount. Once empty, the stack and linked list still exist in memory. Therefore, AllocationCount is still 2 even though the stack data has been popped. They have been initialized and will not be freed until the stack is deleted.
  6. deleteStack checks that the stack is not equal to NULL. Then it gets the stack's linked list and deletes it also by checking that the linked list is not equal to NULL. It does this by getting the front node and it deletes it, removing dynamically allocated memory for each node until there are no longer any nodes in the linked list. Each time, AllocationCount decreases by 1 until there are no nodes and AllocationCount equals 2. Then the information structure (linked list) is deleted, freeing its memory and AllocationCount decreases by 1 to equal 1. It returns NULL to indicate that there is no longer a linked list. After the stack's linked list is deleted, the stack is freed and AllocationCount decreases by 1 to equal 0. It also returns NULL to indicate there is no longer a stack.
  7. Stack.h contains C function declarations and definitions so that a stack can be manipulated. Specifically, it includes a typedef struct with two items such as a pointer to an underlying linked list and a boolean. It contains functions to initialize, check if the stack is empty, push, pop, peek and delete a stack. It is included in Stack.c and StackTester.c because they require the callable functions and checks to maintain a stack. Without including the .h file, the functions return type and parameters will be assumed instead of referring to the hard-coded ones and the program could throw warnings, run poorly or crash.

**Problem 3 follows…**  
**Description:** Problem 3 is designed to demonstrate my knowledge of stack implementation by reading from a file and placing the information in a stack without the use of an array. It consists of functions and definitions for the double linked list. It also contains stack functions for initializing, checking if the stack is empty, push/pop, peek and deleting the stack. AllocationCount will tell how many things are currently dynamically allocated.

**Program Code (UserData.h):**

#ifndef USERDATA\_H\_INCLUDED

#define USERDATA\_H\_INCLUDED

// The UserData struct defines what each node in the LL

// contains.

//

// User data in each node contains an integer and char array

typedef struct {

int taskNumber;

char taskName[80];

} UserData, \*UserDataPtr;

#endif // USERDATA\_H\_INCLUDED

**Program Code(StackTester.c):**

// StackTester will demonstrate the init, push, peek, pop and delete

// for a stack.

// - It builds a stack that holds UserData from a file being read

// - it peeks at the stack, returning UserData

// - It pops UserData from the stack, returning UserData

// - it uses empty() to determine if the stack holds any data that

// can be popped or peeked

// - when done, it deletes the stack

// For demonstration purposes, it shows the number of allocations for

// everything it does.

// printf support

#include <stdio.h>

// we use strcpy from string.h

#include <string.h>

// we use assert from assert.h

#include <assert.h>

// stack callable routines

#include "Stack.h"

// UserData definition for making and getting stack data

#include "UserData.h"

// local functions

// PrintStackItem is a local function that we can call to print out a message (msg) and

// a UserData item. So we can see how many things are allocated as we proceed,

// it will also print out the number of things allocated

static void PrintStackItem (char msg[], UserData D);

// PrintAllocations is a local function that will print out a message (msg) and the

// current global AllocationCount

static void PrintAllocations (char msg[]);

int main(int argc, const char \* argv[]) {

// initialize variables for reading data from a file

char fileName[] = "StackData.txt";

int count;

int i = 0;

int taskNumberTemp;

char taskNameTemp[80];

// Show the allocation count when we start

PrintAllocations ("On startup");

// create a stack ans see the effect of on the number of allocations

Stack S = initStack();

PrintAllocations ("After initStack called");

// push the data on the stack, showing the data and allocations

// attempt to open the file

FILE \* inputFile = fopen(fileName, "r");

// exit if the file did not open

assert(inputFile != NULL);

// the file opened, so proceed and process its contents

// read records from the file, printing out each record

do {

//try to read in a record

count = fscanf (inputFile, "%d%s", &taskNumberTemp, taskNameTemp);

//push to and print stack if enough records are read

if (count == 2) {

//create data to fill in with real values

UserData D;

//set that data's taskNumber and taskName, strcpy is used to write to an array

D.taskNumber = taskNumberTemp;

strcpy(D.taskName, taskNameTemp);

//push now set data to the stack

push (S, D);

//print the results of the push

PrintStackItem("push called, data is", D);

}

//if 2 records are not read, exit the loop

if (count != 2) {

break;

}

//otherwise increment a counter value and continue the do while loop

i++;

} while (count == 2);

//stop reading the file

fclose (inputFile);

// pop and print the stack content

// peek at the data before popping it so we can see what peek yields

while (!empty(S))

{

PrintStackItem ("peek called, data is", peek(S));

PrintStackItem ("pop called, data is", pop(S));

}

// delete the stack and see the effect on the allocations

PrintAllocations ("Before deleteStack called");

S = deleteStack(S);

PrintAllocations ("After deleteStack called");

return 0;

}

/\*

PrintStackData prints out the received message and the data in UserData

\*/

void PrintStackItem (char msg[], UserData D)

{

//now the integer to be printed is UserData's taskNumber

//and UserData's taskName

printf ("%s %d %s, #allocations is %d\n", msg, D.taskNumber, D.taskName, AllocationCount);

}

/\*

PrintAllocations prints out the received message and the current allocation count

The allocation count is global AllocationCount

\*/

void PrintAllocations (char msg[])

{

printf ("%s, #allocations is %d\n", msg, AllocationCount);

return;

}

**Data File:**

1 task1

2 task2A

7 task3A

9 task2B

3 task4A

2 task4B

6 task3B

**Output File:**

On startup, #allocations is 0

After initStack called, #allocations is 2

push called, data is 1 task1, #allocations is 3

push called, data is 2 task2A, #allocations is 4

push called, data is 7 task3A, #allocations is 5

push called, data is 9 task2B, #allocations is 6

push called, data is 3 task4A, #allocations is 7

push called, data is 2 task4B, #allocations is 8

push called, data is 6 task3B, #allocations is 9

peek called, data is 6 task3B, #allocations is 9

pop called, data is 6 task3B, #allocations is 8

peek called, data is 2 task4B, #allocations is 8

pop called, data is 2 task4B, #allocations is 7

peek called, data is 3 task4A, #allocations is 7

pop called, data is 3 task4A, #allocations is 6

peek called, data is 9 task2B, #allocations is 6

pop called, data is 9 task2B, #allocations is 5

peek called, data is 7 task3A, #allocations is 5

pop called, data is 7 task3A, #allocations is 4

peek called, data is 2 task2A, #allocations is 4

pop called, data is 2 task2A, #allocations is 3

peek called, data is 1 task1, #allocations is 3

pop called, data is 1 task1, #allocations is 2

Before deleteStack called, #allocations is 2

After deleteStack called, #allocations is 0