Birla Institute of Technology & Science, Pilani
2nd Semester 2018-19 - CS F211 – Data Structures and Algorithms

---

Lab 2
Topics – Command Line Arguments, Compiling and Linking C files, File I/O, Pointers, Linked Lists

1. Command Line Arguments, Compiling and Linking C files

Hello, Everyone. Welcome back to programming in C. Let us try to revise our C programming skills by writing a few simple C programs and then compiling and executing them.

Open terminal on your machine, which by default opens with the home directory. Create a directory with **your name** and go into that directory. Create a directory **lab1** inside it and go into it. Then create a directory **exer_1** and then go into it. We shall write our first program for today in this directory. The sequence of commands is given below.

$ mkdir <your_name>
$ cd <your_name>
$ mkdir lab1
$ cd lab1
$ mkdir exer_1
$ cd exer_1

For subsequent exercises in this lab sheet, create a separate directory **exer_<exercise number>** in the **lab1** directory. You can use the command "cd .." to come back to the parent directory. We shall be using "**gedit**" to create and edit text files (of type .c or .h or any other text file) in this lab. And we shall be using our traditional **gcc** compiler to compile our C programs.

Let us start with our first C program for today. Create a file "test.c".

$ gedit test.c

This will create and open the file "test.c" in the current directory. Write the program in it as shown below.

```
/* File: test.c */

#include <stdio.h>
int main()
{
      printf("Hello World !!!!);
}
```

Now compile this file to get an executable file as follows:

$ gcc –o test test.c

The above command generates an executable file named **test** that can be invoked as follows:

./test

We may write a program that requires one or more of its inputs to be fed at the command line at the time of invocation. This may also be required when we want the program to be configured/customized based on inputs given at the time of invocation. This can be achieved by using command line arguments in C.

When the "main" procedure in a C program gets invoked it can be passed some arguments – referred to as "command line arguments". For this purpose, C allows the main procedure to be defined with two parameters (whose types are fixed):

```
/* File: test.c */

#include <stdio.h>
int main(int argc, char *argv[])
{
      // body of the program here
}
```

Note that the first parameter (argc) is an integer value that keeps a count of the number of command line arguments and the second parameter (argv) is an array of strings (i.e. values of type char *) that holds all the arguments passed via the command line to **this** program (that caused the invocation of **this** main procedure). In particular, argv[0] holds the name with which the program was invoked and for i > 0, each argv[i] holds a string corresponding to the i^{th} word (i.e. a string separated by space) passed as the command line argument

For instance, given the following main program:

```
/* File: test.c */

#include <stdio.h>
int main(int argc, char *argv[])
{
      printf("\n%s", argv[0]);

}
```

executing these commands

```
$ gcc –o test test.c
$ ./test
```

will output

```
      ./test
```

Exercise 1a: If you invoke the program test as follows, what will be the value of argc?

./test hello world

Exercise 1b: Modify the program test so that it prints "test hello world "for the above invocation. Generalize the program so that it prints the name of the program and all the command line arguments in sequence.

Exercise 1c: Modify the program so that if a command line argument is an integer the parameter string gets converted to an integer. Refer to man pages for conversion function **atoi**.

We will be learning on how to create multiple .c and .h files and compiling & linking them together in the linked list exercise.

2. File I/O Operations

Unlike the simple C programs which take input in the form of arguments or as inputs using **scanf()**, (which you had written in your earlier courses), we would want you to get familiar with **File I/O operations** with which you can take input from a file and then write the output to a file. All the labs in this course shall be using File I/O operations.

A file is an abstraction of the way data gets stored in external or secondary memory (i.e. hard disk, flash memory device, optical disk etc.). A file can be treated as a sequential access FIFO list by a program i.e. if the contents of a file were (say, for instance):

Q W E R T Y

the first accessible element is Q, then W, then E, and so on – i.e. to access E one has to access Q, then W, and then access E; and when say a value U is added to the above file, the contents would be

Q W E R T Y U

i.e. if and when a new element is added it is added to the end (of the list i.e. file). Linux supports text and binary files. We will deal with text files for now. Text files can be accessed character by character or word by word (i.e strings separated by space).

C provides an I/O library **stdio** that contains procedures for I/O access. The header file "stdio.h" contains the headers (i.e. dummy definitions) of these procedures. We must include this header file in our program to perform I/O Operations.

C libraries support two procedures **fscanf** and **fprintf** for reading and writing to a file. Refer to the man pages for information on how to use these procedures. They are similar to **scanf** and **printf** but take an additional (first) argument that is a file pointer.

Since files are abstractions of physical persistent storage, typically initialization and finalization are required i.e. initialization must be done before any read/write operations, and finalization must be done after all read/write operations (particularly before close of program execution). C libraries provide procedures **fopen** and **fclose** for initialization and finalization of a file. Refer to the man pages for information on how to use these procedures.

The typical structure of a program fragment that reads from and/or writes to a file is as follows:

```
FILE *f = fopen("testfile.txt", <mode_of_opening>); // returns a file pointer
      /* read / write operations using the file pointer f */
fclose(f);
```

<mode_of_opening> in the above statement refers to the mode in which you want to open your file. The table below illustrates the modes to be used.

File Mode	Meaning of Mode	During Inexistence of file
R	Open for reading.	If the file does not exist, fopen() returns NULL.
Rb	Open for reading in binary mode.	If the file does not exist, fopen() returns NULL.
W	Open for writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
Wb	Open for writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
A	Open for append. i.e, Data is added to end of file.	If the file does not exists, it will be created.
Ab	Open for append in binary mode. i.e, Data is added to end of file.	If the file does not exists, it will be created.
r+	Open for both reading and writing.	If the file does not exist, fopen() returns NULL.
rb+	Open for both reading and writing in binary mode.	If the file does not exist, fopen() returns NULL.
w+	Open for both reading and writing.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
wb+	Open for both reading and writing in binary mode.	If the file exists, its contents are overwritten. If the file does not exist, it will be created.
a+	Open for both reading and appending.	If the file does not exists, it will be created.
ab+	Open for both reading and appending in binary mode.	If the file does not exists, it will be created.

We will now see two sample program to illustrate the file pointers.

Sample 1: Write to a text file using fprintf()

```
/* Sample1.c */

#include <stdio.h>
int main()
{
   int num;
   FILE *fptr;
   fptr = fopen("program.txt","w");

   if(fptr == NULL)
   {
      printf("Error!");
      exit(1);
   }

   printf("Enter num: ");
   scanf("%d",&num);

   fprintf(fptr,"%d",num);
   fclose(fptr);

   return 0;
}
```

You can compile this as we did for the previous exercise and run the executable. After you compile and run this program, you can see a text file program.txt created in the same directory. When you open the file, you can see the integer you entered.

Example 2: Read from a text file using fscanf()

```
/* Sample2.c */

#include <stdio.h>
int main()
{
   int num;
   FILE *fptr;
   Fptr = fopen (“program.txt”, “r”);

   if (fptr == NULL){
       printf("Error! opening file");
       // Program exits if the file pointer returns NULL.
       exit(1);
   }
```

```
    fscanf(fptr,"%d",&num);

    printf("Value of n=%d",num);
    fclose(fptr);

    return 0;
}
```

This program reads the integer present in the program.txt file and prints it onto the screen. If you successfully created the file from **Sample 1**, running this program will get you the integer you entered.

Other functions like **fgetchar()**, **fputc()** etc. can be used in similar way.

You can also call **fscanf** and **fprintf** functions in a loop to read multiple lines from a file. We will see how to do that in the linked list example.

Exercise 2a: Write a C program that copies the contents of a file into a different file (given two filenames as command line arguments). Take any two sample files and do them. Error-proof your argument: i.e. detect and print messages when errors occur – for instance, *file is not present, unable to read/write*, etc. Also ensure that your program terminates gracefully when errors occur.

3. Pointers in C

Let us revise the basics on pointers in this exercise. A pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is –

```
type *var-name;
```

Here, type is the pointer's base type; it must be a valid C data type and var-name is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations –

```
int    *ip;    /* pointer to an integer */

double *dp;    /* pointer to a double */

float  *fp;    /* pointer to a float */

char   *ch     /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

How to Use Pointers?

There are a few important operations, which we will do with the help of pointers very frequently. (a) We define a pointer variable, (b) assign the address of a variable to a pointer and (c) finally access the value

at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations −

```
/* Pointers_Sample.c */

#include <stdio.h>
int main () {
   int  var = 20;   /* actual variable declaration */
   int  *ip;        /* pointer variable declaration */
   ip = &var;  /* store address of var in pointer variable*/

   printf("Address of var variable: %x\n", &var  );

   /* address stored in pointer variable */
   printf("Address stored in ip variable: %x\n", ip );

   /* access the value using the pointer */
   printf("Value of *ip variable: %d\n", *ip );

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

Address of var variable: bffd8b3c

Address stored in ip variable: bffd8b3c

Value of *ip variable: 20

NULL Pointers: It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a null pointer. The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program −

```
/* NULL_Pointer_Sample.c */
#include <stdio.h>
int main () {
   int  *ptr = NULL;
   printf("The value of ptr is : %x\n", ptr  );
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

The value of ptr is 0

To check for a null pointer, you can use an 'if' statement as follows −

```
if(ptr)     /* succeeds if p is not null */
```

```
if(!ptr)    /* succeeds if p is null */
```

We shall be seeing more of pointers in the next exercise.

4. Linked Lists

After getting familiar with creating simple C programs, I/O operations and pointers in C, let us get started with our first data structure for this course – "Linked Lists". We will use all of the above we have learnt in this exercise.

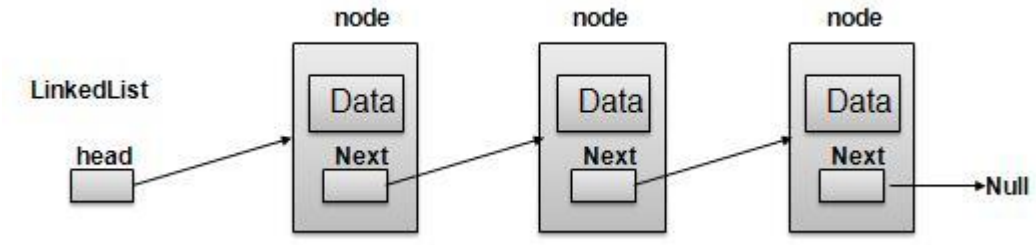
Linked List Basics

A linked-list is a sequence of data structures which are connected together via links.

Linked List is a sequence of nodes which contains items. Each node contains a connection to another node via a link. Following are important terms to understand the concepts of Linked List.

- **Node** − Each node of a linked list can store data called an element.
- **Next** − Each node of a linked list contains a link to the next node called "next". This is typically a pointer variable of type node.
- **LinkedList** − A LinkedList or head of the linked list contains the connection link to the first node, called "first".

Linked List Representation



As per above shown illustration, following are the important points to be considered.

- LinkedList / head contains a link element called "first", which points to the first node in the list.
- Each node carries a data field and a Link Field called next.
- Each node is linked with its next node using its next link.
- Last node carries a Link as null to mark the end of the list.

Types of Linked List

Following are the various flavours of linked list.

- **Simple Linked List** − Item Navigation is forward only.
- **Doubly Linked List** − Items can be navigated forward and backward way.
- **Circular Linked List** − Last node contains link of the first element as next and first element has link to last element as prev.

In this exercise we will work on **Simple Linked List** that store integers as data elements in their nodes. Following are the basic operations supported by it.

- **Insertion** − add an element at the beginning of the list.
- **Deletion** − delete an element at the beginning of the list.
- **Display** − displaying complete list.
- **Search** − search an element using given key.
- **Delete** − delete an element using given key.

Before we study the linked list operations in detail, let us now try to create multiple .c and .h files for making our code modular. Let us have two .c files: “driver.c” and “linkedlist.c”. “driver.c” shall contain all the necessary operations to read data from an external input file, creation of the linked list and linkedlist operations. “linkedlist.c” shall contain all the necessary operations for insertions, deletions and search operations in a linked list. Let us also have a file called “linkedlist.h” which shall contain all basic definitions of the structures and functions associated with operations of linked list. We will learn how to compile, link and execute all of this together, little later in this exercise. Here are the definitions of the structures and declarations of the functions in “linkedlist.h”

```
/* linkedlist.h */

#include <stdio.h>

// structure definitions

// structure of a linked list node. It contains an element
struct node {
      int element;
      struct node * next;
};

/* structure of a linked list / head. It stores the count of number of elements
in the list and also a pointer link to the first node of the list. */
struct linkedList {
      int count;
      struct node * first;
};

// function declarations

void insertFirst (struct linkedList * head, int ele);
/* inserts a given element at the beginning of the list */

struct node * deleteFirst(struct linkedList * head);
/* deletes the first element of the list and returns pointer to the deleted
node. */

void printList (struct linkedList * head);
/* prints all the elements in the list */
```

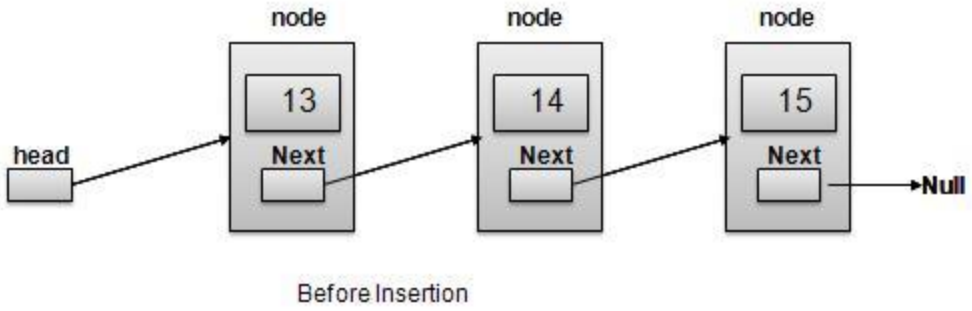
```
int search (struct linkedList * head, int ele);
/* searches for a given element in the linked list. Returns 1 if found, 0
otherwise. */

struct node * delete (struct linkedList * head, int ele);
/* deletes the first node in the list that contains the element = ele and
retuns it. If the element is not found it returns an error message saying
element not found. */

/ * End of linkedlist.h */
```

Now let us study each of the above operations we have declared and implement them in “linkedlist.c” file. You must first include “linkedlist.h” to implement these operations declared in “linkedlist.h”. Let us start with simple insertion operation in linked list.

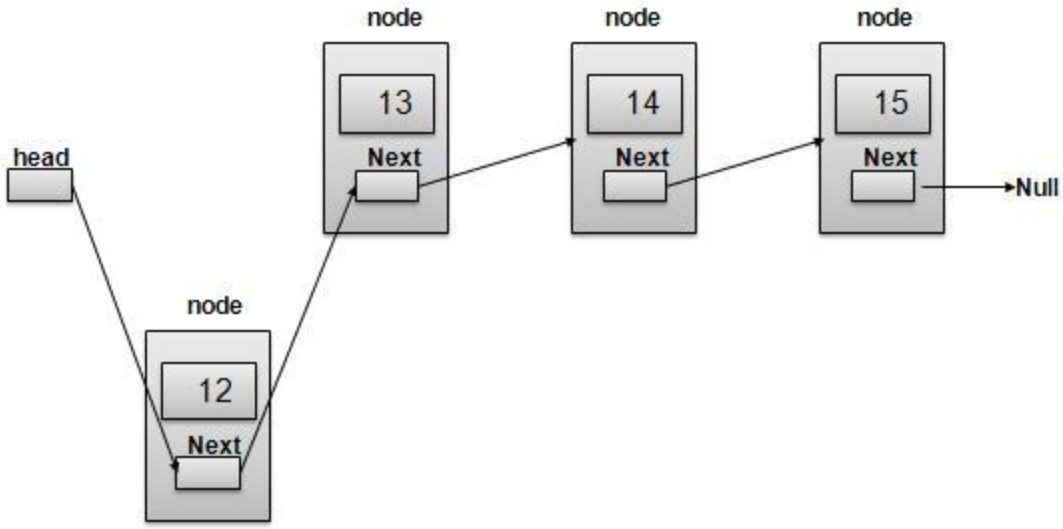
Insertion Operation (Inserts an element at the beginning of the list)

Insertion is a three step process −

- Create a new node with provided data.
- Point New node to old First node.
- Point “First” link in the head to this New node.
- Increment the count in the head.



Before Insertion



After Insertion

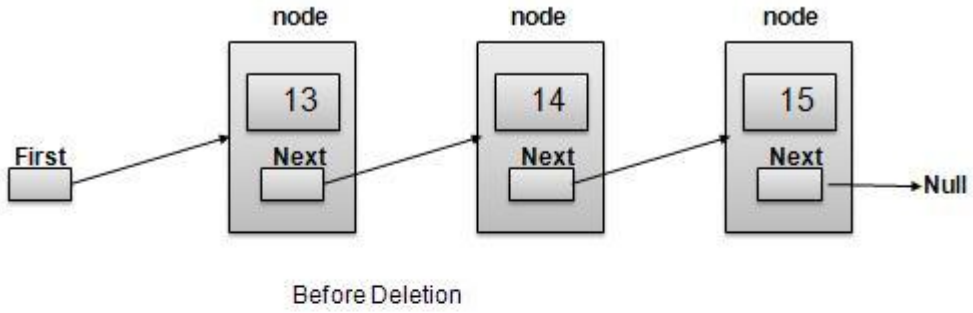
//insert link at the first location

```
/* linkedlist.c */

#include “linkedlist.h”

void insertFirst(struct linkedList * head, int ele){
   //create a node
   struct node *link = (struct node*) malloc (sizeof(struct node)); /* by this
you are creating a node whose address is being stored in the link pointer. */
   link->element = ele;

   //point it to old first node
   link->next = head->first;

   //point first to new first node
   head -> first = link;
   head -> count --;
}
```

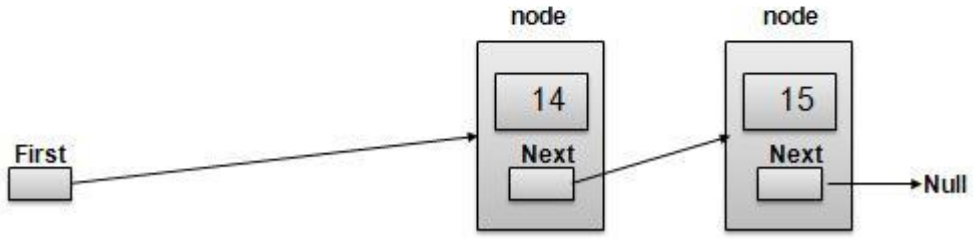
Deletion Operation

Deletion is a two step process −

- Get the Link pointed by First Link as Temp Link.
- Point First Link to Temp Link's Next Link.
- Decrement the count.



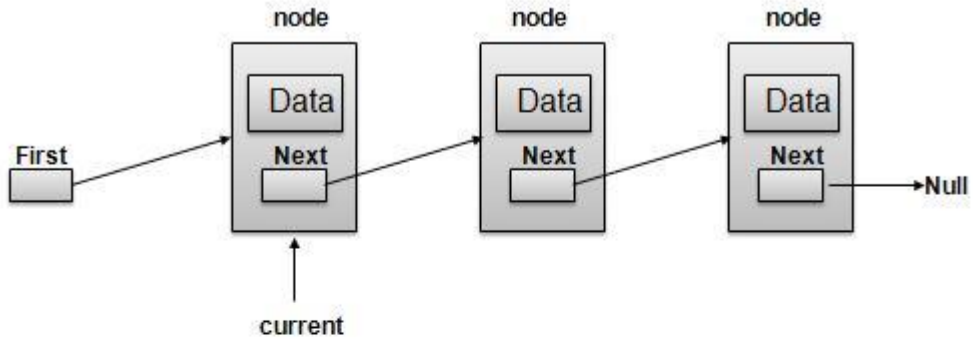
Before Deletion

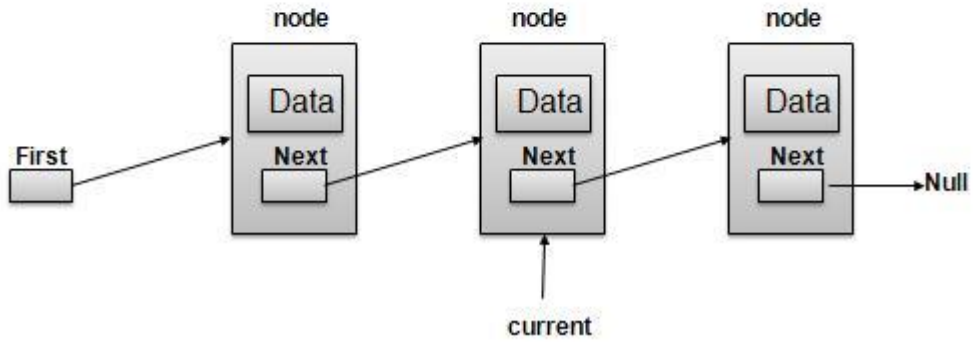


After Deletion

```
//delete first item
struct node* deleteFirst(struct linkedList * head){
      // exercise to implement this operation.
}
```

Printing the list (Navigation)





Navigation is a recursive step process and is basis of many operations like search, delete etc. −

- Get the node pointed by First Link as Current link.
- Check if Current link is not null and display it.
- Point Current link to the Next of Current Link and move to above step.

Note –

```
//display the list
void printList(struct linkedlist * head){
   struct node *ptr = head->first;
   printf("\n[ ");

   //start from the beginning
   while(ptr != NULL){
      printf("%d, ", ptr->element);
      ptr = ptr->next;
   }
```

```
   printf(" ]");
}
```

Now let us create the driver.c file. This is the file that contains our main function. This must include “linkedlist.h” header file to be able to access the functions implemented in “linkedlist.c” file. The outline of “driver.c” is as follows. Your exercise is to implement this file.

```
/* driver.c */

#include <stdio.h>

#include “linkedlist.h”

int main(int argc, char *argv[])
{
      // Create a file pointer and open the file read as command line argument.

      FILE * fileptr = fopen(argv[1], r);

      // Declare a pointer to a linked list (head) and allocate memory to it.

      struct linkedList * head = (struct linkedList *) malloc (sizeof(struct
linkedList));

      // In a loop read the file and insert elements into the linkedList.

      while (!feof(fileptr))
      {
            // read the next element and store into the temp variable.
            int temp;
            fscanf(fileptr, “%d ”, &temp);

            // insert temp into the linked list.
            insertFirst(head,temp);

      }

      // close the file pointer

      fclose(fileptr);

      // print the linked list.

      // delete the first element of the linked list.

      // print the linked list again to check if delete was successful.

      // print the linked list to a new file.

}
```

Create sampleInput.txt to contain data as follows:

32

49

45

9

14

23

17

95

Now let us try to compile, link and execute the above code. We have two .c files – linkedlist.c and driver.c. We need to compile them separately as follows:

$ gcc –c linkedlist.c

$ gcc –c driver.c

This would create two files - linkedlist.o and driver.o. You need to link them into a single executable file as follows and then run it with sampleInput.txt as the command line argument. The program would read the input required from this file.

$ gcc –o exe linkedlist.o driver.o

$ ./exe sampleInput.txt

Exercise 4a: Implement the search and delete operation.

Exercise 4b: Implement **Last-In-First-Out (Stack)** data structure using the Linked List created above. Stack supports two operations: **Push** and **Pop**. Push inserts a given element at the front of the stack. And Pop deletes the first element and returns it. Use the **linkedlist.c** and **linkedlist.h** files and additionally create two separate files – "**Stack.c**" and "**Stack.h**" within in the same directory. Make the **push** and **pop** declarations in **Stack.h** file and implement them in **Stack.c**. You need to create a new driver.c file. In this, **Push** elements in **stackInput.txt** into the stack and then **pop** them out one by one in the driver file. Compile all the files and link them as shown before and execute with stackInput.txt as command line argument. You can create stackInput.txt with following elements:

102

104

107

109

110

101

==========End of the document==========