Objective:

- You will experience a new concept i.e. command line arguments.
- Review/refresher of file streams.
- Another attempt to review your CString library in case if you left any bugs in it.
- And most important part is to experience the use of Link Data Structure by applying the use of Linear Double Link List.

Command Line Arguments

Before you read the actual task (Challenge-X), we need to understand the command line arguments which will be used in the actual task.

It is possible to pass some values from the command line to our C/C++ programs when they are executed. These values are called command line arguments and many times they are important for our program especially when we want to control our program from outside instead of hard coding those values inside the code.
The command line arguments are handled using main() function arguments.

To pass command line arguments into our C/C++ program, we typically define main() with two arguments: first argument is the number of command line arguments and second is list of command-line arguments.

Assume we have written following code in visual studio in a project named as 'TestCommandLine'

```
int main(int argc, char * argv[])
{
    for (int i=0; i<argc; i++)
        cout<<"\n"<<argv[i];
    return 0;
}
```

Compile and run the code:
You will see an executable file in debug sub-folder of your project.
Open the command line window by writing CMD in Run.
Type the executable file name along with complete path and give the arguments to it like:

```
C:/>TestCommandLine my first project with command line
```

And press enter, you will see the following output on concole:

```
C:/>TestCommandLine
my
first
project
with
command
line
```

It means that when we type executable name on command line along with arguments, then these arguments are stored in the array of pointers which we mentioned in main function second parameter, where first parameters represent the count of arguments passed to main.
It is to be noted that argv[0] is the path and name of the program itself.

We are not forced to use argc and argv as identifiers in main( ), those identifiers are only conventions (but it will confuse people if we don't use them). Also, there is an alternate way to declare argv:

```
int main(int argc, char * * argv)
{}
```

Both forms are equivalent.

All we get from the command-line is character arrays; if we want to treat an argument as some other type, we are responsible for converting it inside our program.
Let's look at another example:
An executable named 'add' contains the following code:

```
int addIntegers( int a, int b )
{
    return a+b;
}
int main(int argc, char * argv[])
{
    cout<<addIntegers(atoi(argv[1]),atoi(argv[2]))<<endl;
    return 0;
}
```

So, when we type following on console:

```
C:\>add 30 40
70
```

The addition result 70 is displayed on the console.

Feel free to discuss, if you have any confusions ☺.

Challenge-X

IEditor Application

You have to develop a simple line-based text editor.

Our application/executable name will be IEditor. IEditor will help us to create and open text files and do different editing operation on the text.
Below we shall explore the commands/features and working of IEditor. I have also given a detailed sample run to understand different scenarios.

There will be two versions of your Challenge-X, i.e. X.1 and X.2.

Challenge-X.1:

In this phase, you will define data structure needed to store data/text along with supporting features/commands. All this editing will be done in RAM and has nothing to do with file streams and command line arguments.

Challenge-X.2:

In this phase, you will enhance the ***X.1*** by adding command line argument feature and associated file stream with it. So, basically, in ***X.1***, whatever you do will be lost when program ends but in ***X.2***, the text/data will be saved in file and you will be able to use it from command line.

Details: Challenge-X.1:
Before we decide the data structure that we shall use to develop IEditor, let's look at a ***basic/initial sample Run*** below.

	Comments for Understanding

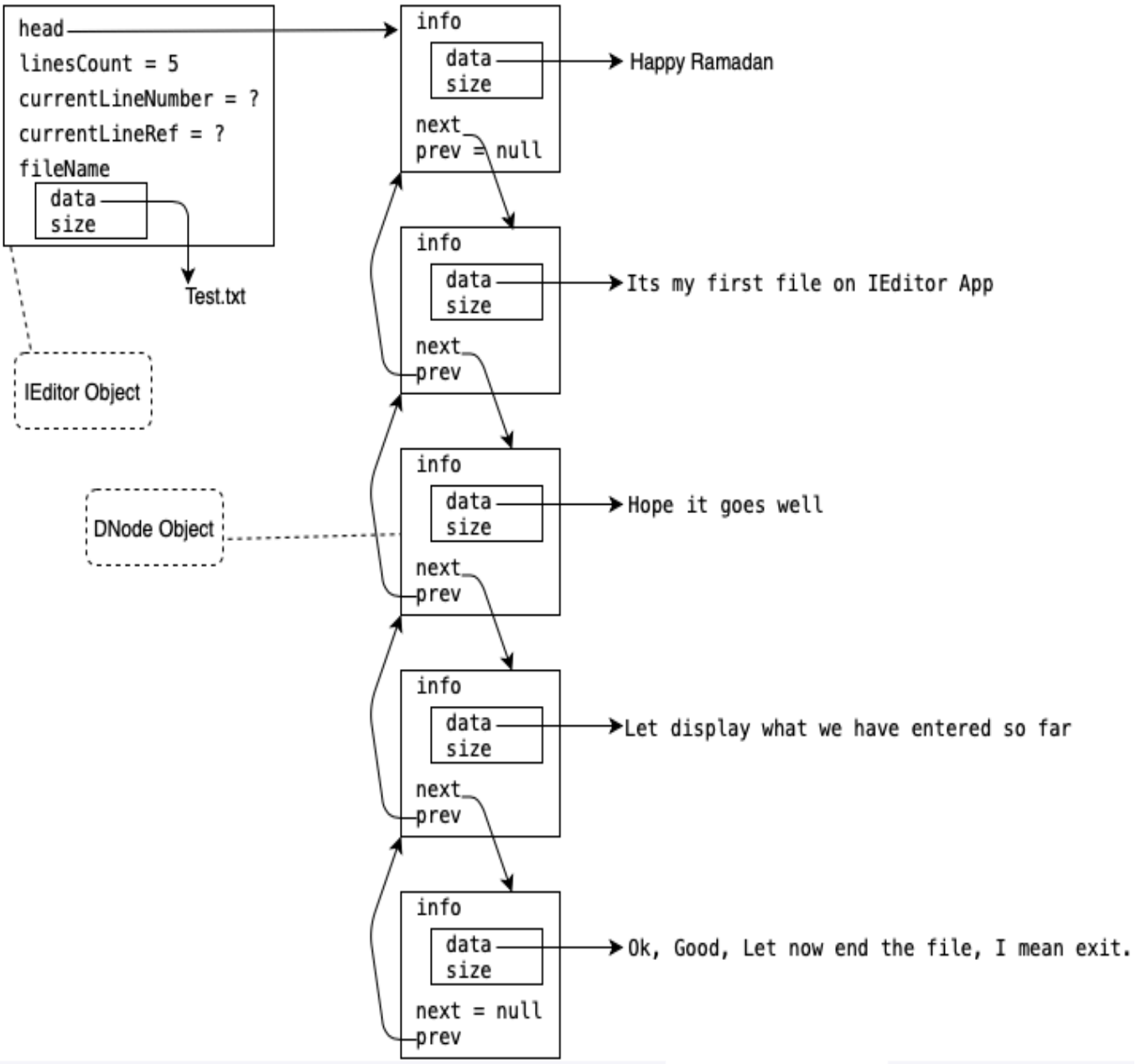
`int main( )`
`{`
`  IEditor ie;`
`  ie.start();`
`  return 0;`
`}`
	By executing ie.start(), you will start the IEditor and it will start taking text from console. Open SR-1 named screen recording file to get the actual feel of IEditor working.

Console	
1> Happy Ramadan **2>** Its my first file on IEditor App	_L command display the file contents

3> Hope it goes well **4>** Let display what we have entered so far **5>** _L **1>** Happy Ramadan **2>** Its my first file on IEditor App **3>** Hope it goes well **4>** Let display what we have entered so far **5>** Ok, Good, Let now end the file, I mean exit. **6>** _E **Program ended with exit code: 0**	_E command exit the program. All data will be lost. number> represents the line number. It's not part of text. See sample Run video file named as 'SR-1'.

Below is the description of data structure used for the *IEditor*:

Basically, *Linear Double Link List* will be used as IEditor, where each *DNode<CString>* will represent one line of text editor.
For the sample run (SR-1), the following object layout will be conceived in memory:



Classes and structures used to develop IEditor APP:

	Purpose
template<**typename** T> **struct** DNode {	Each *DNode<CString>* object represents a line in text editor. A line is a text which ends at '\n'
T info;	
DNode<T> * next;	
DNode<T> * prev;	
DNode();	
{next=prev=**nullptr**;}	
DNode(T val) {info=val; next=prev=**nullptr**;}	
};	

	Purpose
class IEditor {	The class responsible for handling *IEditor* commands and storage of text.
DNode<CString> * head;	Will point to the head node (the first *DNode<CString>* in linear double link structure) i.e. the first text line of the editor.
int linesCount;	Total lines in the file.
int currentLineNumber;	It acts like a cursor in text editor, means the *DNode* number, where your current operation will be done.
DNode<CString> * currentLineRef;	It points to the *DNode<CString>* object which represents your current line. I purposefully left *currentLineNumber*, and *currentLineRef* uninitialized so you should decide yourself that how they should be handled and where they should exactly point.
CString fileName="Test.txt";	Represents the file name where your data will be saved and later retrieved too. Default file name will be "Test.txt". Its actual role will be visible in *X.2* sub-task.
public:	
void start();	Calling this function will start the line editor as the above sample shows as well.
void setFileName(**const** CString & fn);	Setter for the file name.
CString getFileName();	Getter for the file name.
~IEditor();	You know what it should do when the IEditor app close.
};	

	Purpose
enum EditorCommands {LIST_ALL, UP, DOWN, OPEN, SAVE, EXIT, APPEND, REMOVE, MODIFY, NO_COMMAND};	This enumeration may help you name your line editor commands. Its use should increase your code readability. Although you are free to define your own enumeration as per your comfort to achieve such purpose.

	Purpose
class CString {	You are very well familiar with this class. I have just made a couple of additions/modification in it. Make sure, your CString class must not have any public members other than

	listed below and their prototypes must exactly match. Although you may add private functions as per your need but obviously you can't add/minus any data member.
`friend istream & operator >> (istream &, CString &);`	
`friend ostream & operator << (ostream &, const CString &);`	
`char * data;`	
`int size;`	
`public:`	
`CString ();`	
`CString (const char c);`	
`CString(const char *);`	
`CString ( const CString & );`	
`CString & operator = ( const CString & ref );`	
`CString( CString && ref);`	
`CString& operator = (CString && ref);`	
`~CString ();`	
`int getSize() const;`	
`const char * getData();`	Return the address of heap char array pointed by data pointer.
`void shrink();`	
`char & operator [] (const int index);`	
`const char & operator [] (const int index) const;`	
`bool operator!( ) const;`	
`int getLength() const;`	
`int find(const CString & subStr, int start=0 ) const;`	
`void insert(int index, const CString & subStr);`	
`void remove(int index, int count=1);`	
`int replace(const CString & old, const CString & newSubStr );`	
`void trimLeft();`	
`void trimRight();`	
`void trim();`	
`void makeUpper();`	
`void makeLower();`	
`void reverse();`	
`void reSize(int);`	
`int operator == (const CString & s2 ) const;`	
`int operator != (const CString & s2 ) const;`	
`CString left( int count );`	
`CString right( int count );`	
`explicit operator long long int () const;`	
`explicit operator double() const;`	
`CString operator + (const CString & s2 ) const;`	
`void operator += (const CString & s2 );`	
`CString operator () (const CString & delim );`	
`};`	
`istream & operator >> (istream &, CString &);` `ostream & operator << (ostream &, const CString &);`	The only global function in your *CString.h* will be these. Actually, the only global functions in your whole project of *IEditor* app. You are not allowed to add any other global functions anywhere in your project.

Commands Supported by IEditor:

All the commands will be single letter and will start with underscore.

- List All
  - Commands Syntax: _L
  - Purpose: display all the lines of file.
- Move Up
  - Commands Syntax: _U [number]
  - Purpose: If no number is given then _U means to move the cursor one line up. In other words, it moves the *currentLineRef/currentLineNumber* pointer one node previous. If number given then it moves the cursor given number of times up.
- Move Down
  - Commands Syntax: _D [number]
  - Purpose: If no number is given then _D means to move the cursor one line down. In other words, it moves the *currentLineRef/currentLineNumber* pointer one node next. If number given then it moves the cursor given number of times down.
- Append
  - Commands Syntax: _A
  - Purpose: It appends the next entered line text to the current line text. See the sample run for detail understanding.
- Modify
  - Commands Syntax: _M
  - Purpose: It overwrite/modify the text of current line with next entered line text. See the sample run for detail understanding.
- Remove
  - Commands Syntax: _R
  - Purpose: It removes the current line from text.
- Exit
  - Commands Syntax: _E
  - Purpose: It closes the *IEditor* app. Obviously, all data entered will be lost and you have to free the memory at this point.

I have added three screen recordings named as (SR-1, SR-2, SR-3) in the assignment folder which are samples runs. They may help you to easily understand the purpose and behavior of the commands listed above.

Details: Challenge-X.2:
Do think/read about following when you are fully done with X.1.
In this sub-challenge, you will enable your app to be used from command line and will also enable your app to store your entered data on some given file and then eventually retrieve it from any given file.

How to use from command line:

- Name your executable file as *IEditor*
- Your *IEditor* should be able to receive the command line arguments in following way:
  - It will receive two arguments:
    - C:\>IEditor C D:\Files\Test.txt
      - Through this command, I am telling your executable to create a file named Text.txt at the path D:\File\. If the file already exists then the previous contents will be erased. C for create.
        You will save your data stored in *IEditor* structure to this file when you type _E command.
    - C:\>IEditor O D:\Files\Test.txt
      - Through this command, I am telling your executable to open a file named Text.txt at the path D:\File\ and populate your *IEditor* link structure with the text store in it.

Once the *IEditor* structure is populated with the text in file then your app will perform all operation in RAM and when you pass the _E, then it save the data to file again.
So, all the editing will be done in RAM and once you need to close the editor then the data will be overwritten on the file. O for open.

- To understand the behavior of *IEditor* on command line, you should see the sample run screen recording (SR-4), It should clarify any ambiguity left in the description.

In X.2, you will add following two public methods in your *IEditor* class.

	Purpose
`void open();`	Its job is to open the file whose name/path is stored in *fileName* data member. It then populates the *IEditor* link structure with data stored in the file.
`void create();`	Its job is to create the file whose name/path is stored in *fileName* data member.

Instructions and Assumptions

- All your commands of *IEditor* must perform the operations in constant time. Obviously, I am ignoring here the cost incurred by *CString* related operations like concatenation, resizing etc. For example, if there are 10 lines in editor and I insert a text at line number 3 then the lines from 3 onwards should go down in O(1).
  The only exception to the commands is/are _U/_D number. In this case, you have to move `currentLineRef/curretLineNumber` up/down according to given number times. Means the command _U/_D without any given number must also perform its goal in constant time.
- Although you know it already but to remind you, you are not allowed to access any private data member by explicitly typecasting. In short, no explicit typecasting anywhere in your project.
- You are not allowed to use any C/C++ library functions including string class.
- The structure which is used to represents the lines of text editor is *Linear Double Link Structure*, you are not allowed to make it *Circular Double Link Structure* at all
- You are free to add any utility/private functions in *IEditor* class but no public member.
- You may assume that given command syntax will be valid.
- The command letters will be capital.

General Guidelines and Submission Instructions:

- How to Submit:
  - You will create a folder named as RollNo_Assignment2.
    - You will create two subfolders in it. Their names will be X1 and X2.
      - The X1 will contain the code files related to challenge X.1.
      - The X2 will contain the code files related to challenge X.2.
    - In place of RollNo, you will write your complete roll-no in capital.
    - When you are done with the above listed steps, you will compress your main folder RollNo_Assignment2.
      - The main folder and compressed file must have same name.
  - You will submit your compressed file to piazza privately.
    - You must tag your post with assignment_2.
  - Your code files must only be [`CString.h, CString.cpp, DNode.h, DNode.cpp, DNode_Implementation.cpp, IEditor.h, IEditor.cpp, main/driver.cpp`]. Nothing more nothing less.
  - It is advised to test your compressed file properly before sending. Test whether it really compress/decompress the actual files or not.
- Reminder:

- No help from internet/books/humans/machines etc. allowed in order to develop this task. You have to do it at your own.

- Relaxation was given previously but this time, your quiz may not be graded in case of:
  - Late submission
  - Multiple submissions or post editing.
  - Failing to meet any of the submission instructions.
- **You may receive heavy penalty in case of coding conventions violations (like indentation, naming functions, name local identifiers, code distribution etc.).**

I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

[-- Linus Torvalds --]

"Just when the caterpillar thought the world was ending, he turned into a butterfly."

[-- --]