# CS2033 Data Communication and Networks

**Department of Computer Science and Engineering**
**University of Moratuwa**

**Lab Assignment 1 – C Programming Primer**

---

To study and understand computer systems courses such as Data Communication and Networks, Operating System, among others, and to complete relevant lab assignments successfully, basic proficiency in C language is required. This C programming primer intends to provide you with a hands-on training opportunity to learn and master basic C programming as preparation for the labs you will be doing for the rest of the semester. You are encouraged to familiarize yourself with primitive C language syntax by referring to online resources provided. There will be submission links for these C programming primers on Moodle.

[Optional] You are encouraged to attempt this lab on a Unix/Linux environment and the provided compilation instructions will work only on Unix/Linux.

## Lab 1.1 Implementing a linked list

Linked list, a widely used basic data structure which you study in Data Structures and Algorithms, can be used to address many programming problems. In this exercise, you will implement main functions for creating and manipulating a linked list. Your functions should support creation of **circular doubly linked lists** and their manipulations.

The **node.h** file provided contains the definition of a linked list node (`struct NODE`):

```
typedef struct NODE {
  struct NODE *prev;
  struct NODE *next;
  char *word;
} node;
```

You may assume that data is a null-terminated string with a maximum length of 20 characters (excluding the null terminator).

The node.h also contains the definition for the linked list.

```
typedef struct {
  node *head;
} list;
```

If the list is empty, *head will point to a null pointer. For non-empty lists, head will point to a node (which is the node with index 0). Indexes are a convention used to describe a particular node's position with respect to the head of the list. In circular lists, both positive and negative indices can be used to describe a position and negative/positive indicates the direction which the list needs to be traversed to reach the respective position.

Implement the following functions in node.c:

1. void insert_node_before(list *lst, int index, char *word)

   This function inserts a new node that contains a word before the node currently at index of lst. If lst is empty, then lst->head should point to the inserted node. The caller of this function will not free memory allocated for word, so you can copy over the pointer instead of the characters one-by-one. You should use malloc to allocate memory to create the new node. Note that when a node is inserted before lst->head (assuming non-empty list), lst->head will not change.

2. void insert_node_after(list *lst, int index, char *word)

   This function behaves in the same way as insert_node_before, except that new node is inserted after the node at index of lst. (Hint: can you re-use insert_node_before to implement this function?) Note that when a node is inserted after lst->head->prev (assuming non-empty list), lst->head will not change.

3. void delete_node(list *lst, int index)

   This function deletes the node at index in lst. You should use free to release memory allocated for the deleted node and word of the node. lst here is guaranteed to be a non-empty list. If head gets deleted, next node (if any) will become the new head.

4. void delete_list(list *lst)

   This function deletes all nodes in lst. In other words, memory allocated for the nodes in lst should all be freed after calling this function using free. lst here is guaranteed to be a non-empty list.

A template is given in the node.c file. Do not change the function signatures of the above given functions. However, you are free to define any additional helper function in node.c source file. You are not supposed to modify the node.h header file. Our test codes will use only the given node.h header, thus, any change done by you to the node.h will be overwritten.

lab1-1.c contains main(), run(), and print_list() functions. You need to implement your input reading logic and call necessary functions defined in node.h from lab1-1.c. You should not modify the main() function. You should be able to complete the task without using any global variables. However, you are free to use them for any debugging or testing purposes. You will also note that some macros have been defined in the code.

Eg:

#define INSERT_BEFORE 1

This definition indicates that whenever INSERT_BEFORE appears in the code, C substitutes it with 1. You may use the defined macros in your code.

Your program will take as input a **stdin stream** containing a comma separated set of strings that should be the initial contents of the linked list. After the comma separated list (after a newline), input stream contains a list of functions that should be called in the specified order. The function is indicated using an integer which corresponds to the position of the function in the above list. (e.g. 2 refers to insert_node_after function). For each function, the list of input parameters appears on the same line after the function-indicating integer. At the end of the

input, indicated by a 0 digit, your program should print the contents of linked list as a sentence (from head to tail in the order of ascending indexes) to **stdout**.

Example:

Input:

Sri,Lanka,is,a,country
1 4 small
2 -1 in
2 -1 South
2 -1 Asia
3 5
1 5 nation
0

Output:

Sri Lanka is a small nation in South Asia

You will need to learn about malloc and free functions in C in order to complete this task. A brief note on the usage of these functions is provided at the end of this writeup. You are also encouraged to find relevant resources and examples online for further understanding.

**After you have finished implementation, use the following command using a console applications to compile your code (you need to be inside the source directory where the Makefile and source code are located):**

```
make
```

This will produce the compiled binary files **`node.o`** and **`lab1-1`**. (This comes from **`Makefile`** in the directory. You can read up more about Makefiles online.)

To execute your program and let it read inputs from stdin and output to stdout, just use the command:
**`./lab1-1`**

To make things easier, you can read any inputs from a file and output to a file with the following command:

```
./lab1-1 < input.txt > output.txt 2> errorlog.txt
```

**This command will redirect the standard input (stdin) to file input.txt, standard output (stdout) to output.txt, and the standard error (stderr) output to errorlog.txt. Normally, the error log file should be empty.**

Each time before you compile the code again, run **`make clean`** to clear the previously generated binary files.

# Lab 1.2 Matrix operations with pointers

Matrices are used in a wide variety of computing applications and needs no introduction as you have studies matrices in-depth in mathematics courses. In this exercise, you will implement the following list of matrix manipulation functions for the matrix struct defined as below in chain.h.

```
typedef struct {
  int **data;
  int num_rows;
  int num_cols;
} matrix;
```

For an arbitrary 2-D matrix stored in a struct matrix, implement the following functions.

1. matrix *create_matrix(int num_rows, int num_cols)

   This function creates a zero matrix (aka, all cells in the matrix are zero) with the specified dimensions. It is up to you to decide how to allocate memory to store values in int **data, but you must make sure matrix is stored in row-major order. That is, data[r-1][c-1] contains the element of the matrix at $r^{th}$ row and $c^{th}$ column (the first row/column is row/column zero).

2. void add_row(matrix *mat, int *row)

   This function adds a row to the bottom of the matrix, i.e., the added row will become the last row. row is guaranteed to be valid and have the same number of elements as specified by mat->num_cols. Note that row will be freed by caller after calling this function, so you need to copy over the element one-by-one. You may find realloc from stdlib.h useful.

3. void add_col(matrix *mat, int *col)

   This function adds a new column to the right of the matrix, aka, the added column will become the rightmost column. col is guaranteed to be valid and have the same number elements as specified by mat->num_rows. Note that col will be freed by caller after calling this function, so you need to copy over the element one-by-one.

4. void increment(matrix *mat, int num)

   This function increments each value in the matrix by num.

5. void scalar_multiply(matrix *mat, int num)

   This function multiplies each value in the matrix by num.

6. void scalar_divide(matrix *mat, int num)

   This function divides each element in the matrix by num. For simplicity, you can just use integer division, aka, you can just use / operator. It is guaranteed that num will not be 0.

7. void scalar_power(matrix *mat, int num)

This function raises each element in the matrix to the power of num**. You need to implement a function to calculate power.**

8. void delete_matrix(matrix *mat)

This function deletes mat. You should use free to release all memory allocated for mat.

This exercise is tested in combination with Lab1.3 exercise. Therefore, no example test cases, or inputs provided. Nevertheless, since these operations are intuitive, you should come up with few test cases to check for the correctness of your code.

Your implementations should be done in chain.c file. For your convenience, we have already provided you with a print_matrix function, which you can use for testing purposes.

## Lab 1.3 Matrix computation chain with a linked list

Given an initial input matrix, and a sequence of matrix operations (the ones you implemented in Lab 1.2), you are required to write a program to:

- perform the sequence of matrix operations on the initial matrix,
- track the transformation of the matrix in a linked list, and
- once the sequence of operations is completed, output the transformation by traversing the linked list

chain.h contains necessary function definitions for the exercise. Implement necessary functions in the same chain.c file continuing from Lab1.2. lab1-2-3.c contains main(), run(), and print_chain() functions. You need to implement your input reading logic and call necessary functions defined in chain.h from lab1-2-3.c. You should not modify the main() function. You should be able to complete the task without using any global variables. However, you are free to use them for any debugging or testing purposes.

Example:

Input:

```
3
1 1 2 5
2 5 4 7
3 1 4 7
2 2 4 5 4
4 1
3 3 4 7 2
7 2
0
```

Explanation of the input:
3 //indicates the number of rows in the matrix, thus, read following 3 lines as input matrix

1 1 2 5//matrix row1
2 5 4 7//matrix row2
3 1 4 7//matrix row3
2 2 4 5 4//function code = 2 (add row), input = 2 4 5 4 (row to be appended to matrix)
4 1//function code = 4 (increment), input = 1 (increments elements by 1)
3 3 4 7 2//function code = 3(add column), input = 3 4 7 2
7 2//function code = 7(scalar power), input = 2
0//end of input sequence

Expected output (you should output this to stdout):

1 1 2 5
2 5 4 7
3 1 4 7

1 1 2 5
2 5 4 7
3 1 4 7
2 4 5 4

2 2 3 6
3 6 5 8
4 2 5 8
3 5 6 5

2 2 3 6 3
3 6 5 8 4
4 2 5 8 7
3 5 6 5 2

4 5 9 36 9
9 36 25 64 16
16 4 25 64 49
9 25 36 25 4

Explanation of output:

1 1 2 5 //input matrix
2 5 4 7
3 1 4 7

1 1 2 5//add a row 2 4 5 4
2 5 4 7
3 1 4 7
2 4 5 4

2 2 3 6//increment all by 1
3 6 5 8
4 2 5 8
3 5 6 5

2 2 3 6 3//add a column 3 4 7 2
3 6 5 8 4
4 2 5 8 7
3 5 6 5 2

4 5 9 36 9//raise all elements to power of 2
9 36 25 64 16
16 4 25 64 49
9 25 36 25 4

- You can see that the operations performed are indication with an integer identified which matches the numbering indicated in the function list provided in Lab1.2.
- Once you read the input matrix from the input, you should create a linked list and assign the input matrix to the head node.
- For each matrix operation, you need to:
    - Insert a new node to the linked list
    - Copy the latest version of the matrix from the preceding node of the list
    - Perform the operation on the new copy of matrix attached to the newly created tail node of the list
- When the list of operations in the input is exhausted, traverse the linked list from the head node to the tail node and print (output) the matrices at each node into stdout.
- While implementing Lab1.3, you are not allowed to change the function signatures given in chain.h. However, you are free to define any additional functions you may need for this task.

A Makefile has been provided for compiling your code. Please follow the compilation instructions provided in Lab1.1 and apply them accordingly to this exercise.

## Testing your code

We have provided simple test cases to test your code on your local system.

Apart from testing your code in your local machine, we have setup quizzes on Moodle to automatically test your code for a series of test cases. The quiz is already configured by setting up the relevant header files and macros. Thus, you only need to copy and paste relevant functions to the template provided in the quiz. These quizzes will not be graded and are designed only for you to enhance your proficiency in C programming.

Note: We are new to C auto grading on Moodle, therefore you might experience some glitches. Please let us know if you come across any issues while using this facility.

## How to seek help?

1. Online resources: we have provided a set of resources on the Moodle page. Besides, there are ample resources for learning C online. Please make use of them.
2. Peer support: you are encouraged to work with your friends to resolve any issues that you might face.
3. C Language forum: If you have unsolvable issues, please post your question in the C Primer Lab Moodle forum. Don't forget to actively contribute to forum discussions.

# Appendix

## Sample usage of malloc

malloc is used to allocate memory on heap. Hence, we need to specify the size of memory to be allocated to our variable. This can be achieved by calling sizeof(<variable's type>). Upon success, malloc returns a pointer to the beginning of the memory region allocated. Note that as malloc simply allocates the specified size of memory, it is not aware of the underlying data type. Hence it is a good practice to cast the returned pointer to the correct pointer type like below:

```
int *a_ptr = (int *) malloc(sizeof(int) * 4);
```

The code above allocates memory to store 4 integers, casts returned pointer to int *, then assign it to a_ptr variable. Besides malloc, calloc is also used to allocate memory. You can find out more about it if you are interested.

## Sample usage of free

To release the memory allocated for a_ptr (malloced as above), we simply call free(a_ptr). We do not need to explicitly specify the size of memory to be released because malloc already stores accounting information about the amount of memory allocated to a_ptr. So free just reads that information to determine the size of memory to release.

However, when we have double pointers or pointers stored in struct, we cannot simply free the "outermost" pointer, because the accounting information for this pointer only records how much memory is used for this pointer itself, not the memory (nor the value) of the other pointers "nested" inside. Hence, to release memory properly, we need to free them from "inside out".