

**2018/2019 – 1<sup>st</sup> Semester**  
**COMP2123B Programming Technologies and Tools**  
**Group-based Self-learning Report**

**Topic: Implementing data structure in C – Linked-list and Stack**

Name: Kwan Hiu Hong (1 student only)

UID: 30353 75167

Disclaimers:

1. Since this report is used for peer learning, I will try to make the presentation of materials similar to Dr. Chim's style for readers' convenience and coherence. (Here great thanks to Dr. Chim's approval and support for the use of his designs 😊)
2. For most of the students at HKU, I will assume that they all know about basic C++ and C programming, such as the use of header file <string.h> in C, so I will NOT repeat the basic syntax of C.

## C programming practices – Implementing Linked-list in C programming language

Estimated time to complete this programming practice: 1 hour

### Objectives

---

At the end of this section, you should be able to:


- Define a class-like node structure in C
- Differentiate the differences in implementation of linked-list between C++ and C
- Know how to implement a linked-list in C, such as insertion, traversal, reversal and advanced usage such as sorting a linked list in C using merge sort.

### Section 1.1 Node class definition in C

---

A node is a class that contains the element and the pointer to next node. It is a very important part of a Linked-list. In C++, we can directly define the class node like the following:

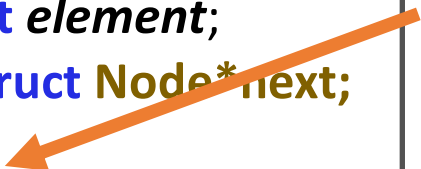
```
#include <iostream>
using namespace std;
Class Node(){
Public:
    int element;
    Node*next;
};
```




However, since we do NOT have Class in C, we cannot do that!

Instead, we can use **struct** to achieve the similar class node in C++!.

```
#include <stdio.h>
struct Node(){
    int element;
    struct Node*next;
}
typedef struct Node
node;
```



 **typedef** is commonly used in C to let the compiler treat anything between typedef and the last word, be the last word. In this example, when we create an object a “node a”, it is same as coding “struct node a”.

## Section 1.2 Create a linked-list (Inserting element in C linked-list) and traverse it

- Let's open the linkedlist.c and follow the instructions to create a linked-list in C.

```
$ gedit linkedlist.c &
```

- Suppose we are implementing a phonebook app, which is to store all our friends name and their phone numbers using a linked-list structure in C. We ask user keep inputting his friends' name and phone, and end the program by inputting "end".

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct Node(){
    char name[20];
    int number;
    struct Node*next;
}
typedef struct Node node;
int main(){
    char name[20];
    int number;
    node* head = NULL;
    while(scanf("%s", name)){
        if(strcmp(name, "end") == 0)
            break;
        scanf("%d", &number);
        //insertion happens
    }
```

Note that we will insert the pairs of name and phone at the end of the linked list.



linkedlist.c

- Please copy the following code below the comment – “//insertion happens” in the program, it inserts element into the linked list.

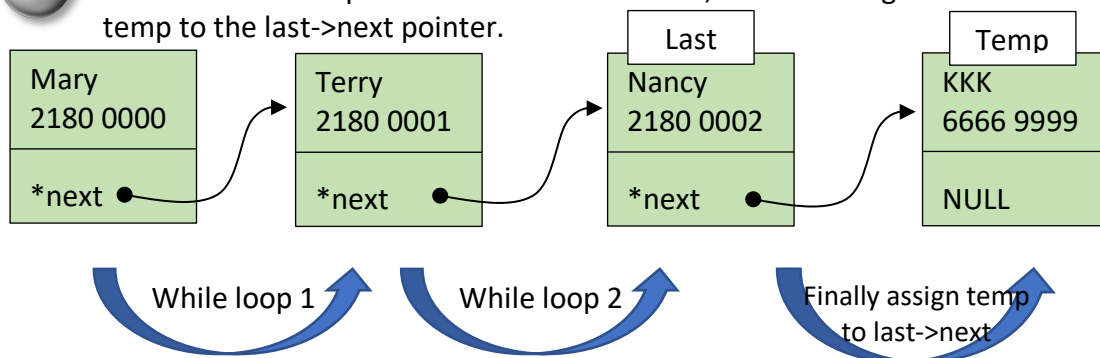
```
node *temp = (node*)malloc(sizeof(node));
strcpy(temp->name, name);
temp->phone = phone;
if (head == NULL) head = temp;
else{
    node *last = head;
    while(last->next) last = last->next;
    last->next = temp;
}
```

Remember in C, we don't have type string. To assign one string to another, we either use strcpy or use a loop to assign every single char from one to another.



### Code explanations:

- 1 For the first line, we allocate dynamic memory just enough to store a node object and convert its type to a node pointer type in order to allow the node pointer temp to point to this block of memory. Indeed, we will use temp to store our user's input.
- 2 Then for the 2<sup>nd</sup> and 3<sup>rd</sup> line, we assign user input to the dynamic memory where temp is pointing to.
- 3 For the 4<sup>th</sup> line, we check if there is nothing in the list, if yes, we assign the address of the first pair of name and phone number to head.
- 4 Then for the 2<sup>nd</sup>, 3<sup>rd</sup>, and .... pairs of name and numbers, we create a last pointer, and let the last loop until its last->next is NULL, then we assign the address stored in temp to the last->next pointer.



- Finally, let's traverse the linked-list and see if we do it correctly! (please copy the following traversal code outside and below the outset while loop in the program.

```
node *trav = head;
while(trav){
    printf("%s %d\n", trav->name,
    trav->phone);
    trav = trav->next;
}
```

- Let's see if we successfully build the linked-list! Type the following on your terminal

```
$ gcc linkedlist.c -o linkedlist
$ ./linkedlist
Mary 21800000
Terry 21800001
Nancy 21800002
KKK 66669999
end
```

- After that, you should have the following output on your terminal!

```
The telephone book now has:
Mary 21800000
Terry 21800001
Nancy 21800002
KKK 66669999
```

Do you find it very similar between the linked list in C and C++ ? They are only different in terms of memory allocation, class/structure definition, the manipulation on string and potentially, the use of pass by reference!



### Checkpoint 1a – Reverse a linked list in C

Assume we are using the `Reverse_list.c` as our source code, which is the same as the code as above. Suppose you are given a sample code to reverse a linked-list in C++, can you now translate the program into a C program? Let's try it! (\*\*For the details of the following C++ code about reversing a list, you may refer to Dr.Chim's PPP questions - "Practices on Linked List".\*\*)

```
1 //This code is written in C++ syntax
2 void reverseList (node *& head){
3     node *current = head;
4     node *previous = NULL;
5     node *temp;
6     while ( current!=NULL ){
7         temp = current->next;
8         current->next = previous;
9         previous = current;
10        current = temp;
11    }
12    head = previous;
13 }
```

**\*\* Please copy the above code in the program under the first comment.**

**We will call `reverseList( &head )` to reverse the linked-list in C ! \*\***

- For the 1<sup>st</sup> line, since we do not have pass by reference (&) in C, we should only pass in a pointer of the head pointer, denoted as PoP – Pointer of Pointer. So let's change it to the following:

```
void reverseList (node **PoP){ // same as passing "node *& PoP" in C++
```

So now, PoP is the pointer of the head pointer – Since we will pass the address of the head pointer into this function. (Sounds complicated right?)

Since every variable , once defined, has its own address in the main memory, **so does a pointer !** So PoP now is actually pointing to the head pointer (Or specifically speaking, PoP is storing the address of the head pointer!). By de-referencing the PoP, say `*PoP`, we actually get the head pointer.



- Then for 2<sup>nd</sup> line, since “current” should be storing the head pointer, to access the head pointer from PoP, we here have to first dereference PoP to get the head pointer. So please also change the 2<sup>nd</sup> line to the following:

```
node *current = *PoP;
```

- For 12<sup>th</sup> line, we also need to dereference PoP to get the head pointer and then assign node “previous” to head, change to the following:

```
*PoP = previous;
```

- For other parts of codes, nothing needs to be changed. Then we call this function by transferring the address of the head pointer into the function call:

```
reverseList( &head );
```

- After all, let’s try to compile the program and see what we get!

```
$ gcc Reverse_list.c -o reverse
$ ./reverse
Mary 21800000
Terry 21800001
Nancy 21800002
KKK 66669999
end

The telephone book now has:
Mary 21800000
Terry 21800001
Nancy 21800002
KKK 66669999
The reverse list is:
KKK 66669999
Nancy 21800002
Terry 21800001
Mary 21800000
```

It works!

Please submit the Reverse\_list.c.



**Checkpoint 1b below is optional**, it is for those who want to learn a powerful sorting algorithm – Mergesort() to sort a linked list in C. If you are interested in other sorting algorithms, you can refer to Dr.Chim lab6.6 – “C Programming (Sorting Algorithm)”. The reason why I introduce Mergesort here, is, first it is not included in Dr Chim’s slide, and second, Mergesort takes  $O(n \log n)$  complexity on worst, average, and best case. With multi-threading, it can even sort an array or linked-list using  $O(n)$  complexity by sorting different parts of linked-list simultaneously.



### Checkpoint 1b (**Optional – Advanced** application of the concept - pointer of pointer in C linked-list) – Using Mergesort() to sort a linked-list in C

- In this part, we will first learn what a Mergesort() is and what are the rationales behind. Then we will apply the phonebook linked-list we just created in the above section, and sort the phonebook according to the lexicographical order of names. (Lexicographical order means it follows dictionary order, say “Apple” is occurred before “Application”.)
- The idea of *merge sort* is to (1) divide the list into 2 roughly equal parts, (2) sort the 2 parts individually and (3) merge them afterwards (We need an extra container, say an array of the same size of the input array if we are sorting an array, or a new linked-list if we are sorting a linked-list). Say, for example, we have an array like this:

Input array:

14	9	23	88	41	33	2	52
----	---	----	----	----	----	---	----

♦ **Example:**

```

9 14 23 88 || 2 33 41 52
9 14 23 88 || 33 41 52
  14 23 88 || 33 41 52
    23 88 || 33 41 52
      26 || 33 41 52
        || 33 41 52

```

First split the array at the middle, **sort the two arrays respectively**, and then compare every single items from two arrays at the beginning, and put the smaller into a container (maybe array or linked-list) – This is called “Merging”. So after every line in the example on the left, we have the newly sorted array:  
 {2} -> {2,9} -> {2,9,14} -> {2,9,14,23} ->  
 {2,9,14,23,26} -> {2,9,14,23,26,33,41,52} -> **Sorted!**

- Okay, let’s open the linkedlist\_mergesort.c given and I already included the functions declaration we need for this program ☺, other parts of codes are the same as “linkedlist.c”. We’ll then fill in the functions definition one by one.

```
$ gedit linkedlist_mergesort.c &
```

- First, please copy the following definitions for the “`int get_list_size(node *head)`” function. This function will return the length of the linked list for the sake of finding the middle node for `Mergesort()` function below.

```
int get_list_size(node *head){
    int cnt = 0;
    while(head){
        cnt++;
        head = head->next;
    }
    return cnt;
}
```

The code should be easy to understand, we basically traverse the list and count



- Second, we **first complete the `Mergesort()` function**. Please copy the following code to the “`void Mergesort(node **head)`” function.

```
Void Mergesort(node **head){
    If((*head)->next != NULL){
        int size = get_list_size(*head);
        node *head1 = NULL;
        node *head2 = *head;
        for(int cnt = 0; cnt < size/2; cnt++){
            head1 = head2;
            head2 = head2->next;
        }
        head1->next = NULL;
        head1 = *head;
        Mergesort(&head1);
        Mergesort(&head2);
        (*head) = Merge(&head1, &head2);
    }
}
```

//We pass in Pointer of Head Pointer.  
 //If the list has nothing or 1  
 //element, no sorting needed.

//Here we try to separate the original linked-list into two sub-linked list with roughly equal length. We first get the size, and traverse the list to half, and assign the head1 to store the first half list and head2 to store the second half list.

//Finally, we **recursively** call the `Mergesort` to sort the left and right sublist. Then merge them together and change the head pointer to the smallest node of a sorted list.

Note that the reason why `Mergesort()` accepts the pointer of pointer as parameter is because we will call the `Mergesort(& head)` in the main function, where head is the pointer to the head of an unsorted list. And because we will probably change this head pointer, so we need to pass the address of this head pointer in order to allow modification outside the main function! For other details, like the usage of dereferencing the pointer of pointer, it's the same as Checkpoint 1a.





- Finally, we will implement the **Merge()** function, which is used to merge two sub-lists together in a single sorted list. Then we will return the pointer to the head of this sorted list. So please copy the following code to the Merge function definition.

```

1. node* Merge(node **head1, node **head2){
2.     node *new_head = NULL;
3.     if(*head1 == NULL) return *head2;
4.     if(*head2 == NULL) return *head1;
5.
6.     if(strcmp((*head1)->name, (*head2)->name) < 0){
7.         new_head = *head1;
8.         new_head->next = Merge(&((*head1)->next), &*head2);
9.     }else{
10.        new_head = *head2;
11.        new_head->next = Merge(&*head1, &((*head2)->next));
12.    }
13.    return new_head;
14. }

```

#### Code explanations:

**1** For line 1, again, we will send the pointer of head pointer to the first half sub-list and the second half sub-list as parameters in this function. But notice that the type of **return value is a node pointer**, as we will return the head pointer to the sorted list to the main function. And **this Merge function is also a recursive function**.

**2** Then for line 2-4, we define a new\_head pointer to store the head pointer of the sorted list. Then we define the base case of this recursive algorithm. We check if the first half list is empty or not, and if yes, it will return the head pointer of second half list. The same applies if second half list is empty. Why?

#### ♦ Example:

9 14 23 88 || 2 33 41 52

... ..

26 || 33 41 52

|| 33 41 52

Do you still remember the example we used on previous 2 pages?

When it comes to this line, first sublist is empty, the only thing we need to do is to concatenate the second sub-list to the end of the newly sorted linked list. Since the second sub-list is sorted, no more sorting/comparisons required ☺ Since the merging will end at this step, it is the base case.



- 3 For line 6-8, we compare the lexicographical order of the name, if the name, say Axx compared with Bxx, then Axx should occur before Bxx. Then we assign the address to this node to pointer new\_head. And then we call Merge() again to return the subsequent node that is just larger than Axx to its next. Here be careful that we **need to pass the address of pointer** as parameters when calling this function. And since head1 is compared and inserted to the newly sorted linked list, when calling Merge(), we simply pass in the address of "(\*head1)->next", but not &\*head1.
- 4 For line 9-12, the logic is the same as 3
- 5 Finally, for line 13, we return the head pointer of the newly sorted linked-list to where the function call happens. Then the whole mergesort is done. 😊

- Let's test if the program works!

```
$ gcc linkedlist_mergesort.c -o mergesort
$ gcc ./mergesort
Mary 21800000
Terry 21800001
Nancy 21800002
KKK 66669999
end
The telephone book now has:
KKK 66669999
Mary 21800000
Nancy 21800002
Terry 21800001
```

**It works! 😊**  
**Done!**

Please submit the linkedlist\_mergesort.c.



## Section 2 - Implementing Stack Data Structure in C

---

In this section, we will try to learn the basic of stack and learn how to implement a stack using existing knowledge that we have (No fancy tricks or new skills), basically using an array to mimic the operations of a stack! Why do we do this? Because there is no standard template library in C that provides stack container 🤔, unlike C++. And we will implement some major operations of stack, namely, push(), pop() and top(), and apply stack to solve a simple interesting problem.

### Objectives

---

At the end of this section, you should be able to:

- Know what a Stack is and how it works
- Create a stack library yourself and apply it to solve some practical problems

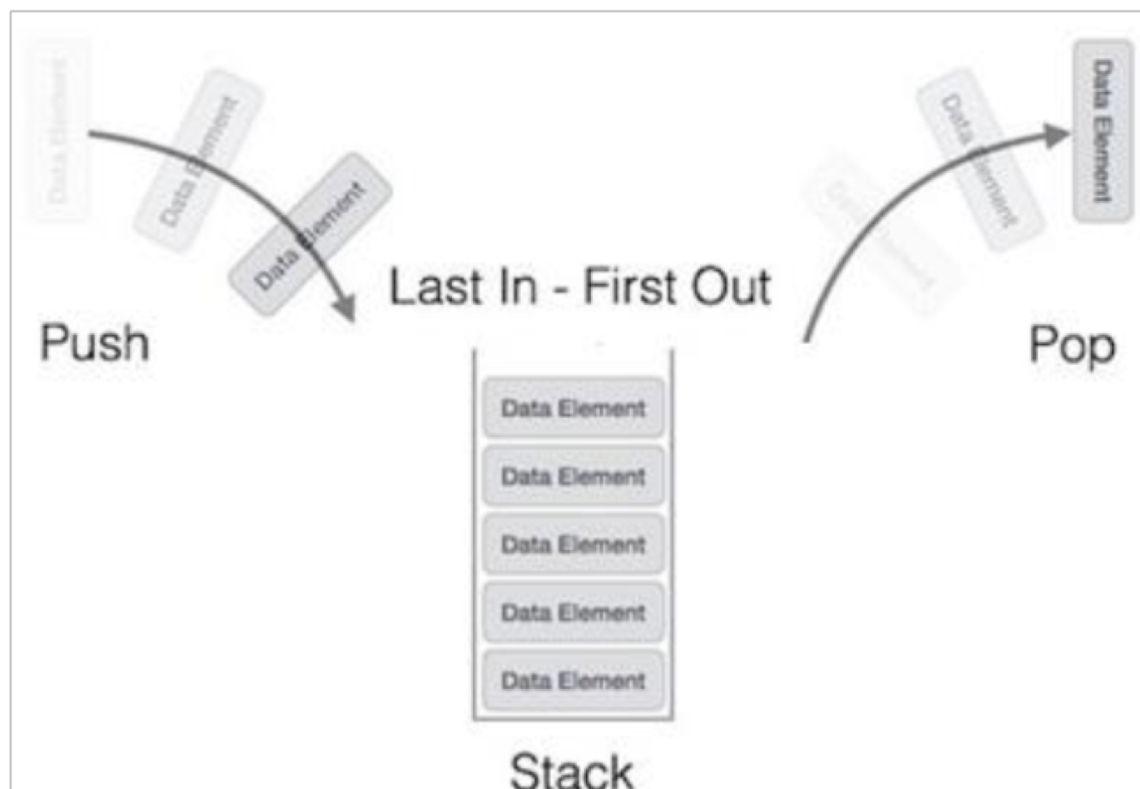
### Brief Introduction to Stack Data Structure

---

Stack is an array-like data structure which operates on the principle of **Last-in-first-out (LIFO)**, which means when everytime you **Push()** (means insert) an element into a stack, say you push,4,3,5,6 and then delete one item from stack using **Pop()**, it must return and delete 6 (As it is the last one inserted into the stack). And it also supports **Top()** which is used to output the most recent inserted element/the top element in stack.

More formal definition of Stack could be found on:

<https://www.geeksforgeeks.org/stack-data-structure/>



## Section 2.1 Set-up of the stack

Before we implement the Push(), Pop() and Top(), let's see what we need for creating a stack first.

- Let's open the stack.c and see what we have.

```
$ gedit stack.c &
```

```
1.  #include <stdio.h>
2.  #include <string.h>
3.  #define stack_storage 1000
4.  int stack_size = -1;
5.  void push(int stack[], int user_input);
6.  int pop(int stack[]);
7.  int top(int stack[]);

8.  int main(){
9.      int stack[stack_storage];
10.     for (int cnt = 0; cnt < stack_size; cnt++) stack[cnt] = 0;
11.     char command[5];
12.     int input;
13.     printf("Please push/pop/top the stack and end the input by typing 'end'\n");
14.     do{
15.         scanf("%s", command);
16.         if(strcmp(command, "end") == 0){
17.             break;
18.         }
19.         if(strcmp(command, "push") == 0){
20.             if(stack_size == stack_storage-1){
21.                 printf("Stack overflows!\n");
22.                 break;
23.             }
24.             printf("The value that you want to push:");
25.             scanf("%d", &input);
26.             push(stack, input);
27.         }else if(strcmp(command, "pop") == 0){
28.             pop(stack);
29.             if(stack_size == -1){
30.                 printf("Stack underflows!\n");
31.                 return 0;
32.             }
33.         }else if(strcmp(command, "top") == 0){
34.             printf("%d\n", top(stack));
35.         }
36.     }while(strcmp(command, "end"));
37.     printf("\nThe stack now contains the following\n");
38.     for(int count = 0; count <= stack_size; count++){
39.         printf("%d ", stack[count]);
40.     }
41.     printf("\n");
42.     return 0;
43. }
```

stack.c

### Code explanations:

1. On line 3, we define the stack storage to be 1000 integers. This does not mean the size of the stack, but the maximum items that the stack can contain.
2. On line 4, we first define the size of the stack to be a global variable with value equal to -1 when it's empty, because the array/stack index starts from 0.
3. Line 5-7 is for section 2.1-2.3 which we will implement later.
4. For line 9 and 10, we initialize the stack to be empty.
5. For line 11-13, we ask the user to input what he/she would like to do, maybe push/pop/top or end the program.
6. For line 14-36, we implement the stack operations according to user's input.
7. And for line 20-23, we check if the numbers of items that user push into the stack exceed the maximum storage of the stack, if yes, we stop the program and output stackoverflow.
8. Similarly for line 29-32, we check if there is still element in the stack, if not, then when user still requests to pop (actually nothing the user can pop when the stack is empty), we therefore stop the program and output stackunderflow.
9. Finally, for line 37-41, we print out the content of the stack and see if we operate correctly. Usually this part is for debugging purpose, when you really use the stack to solve practical questions, you can simply comment out this part.



After setting up the framework of stack, we can now build up other functions for the stack!

### Section 2.2 push() – Insert a new element into stack and increase stack size by 1

---

- push() functions will insert one element into the stack and increases the stack size by one, try to fill up the missing part in our program with the following:

```
1. void push (int stack[], int user_input){  
2.     stack[stack_size+1] = user_input;  
3.     stack_size++;  
4. }
```

- It's easy to implement actually! We just put one element at the end of the array-stack, then we increase the stack\_size global variable by one. Done!

Note that pop() is also pretty similar to push(). But we have to be careful about **Stack Overflow and Underflow** when we are doing push() and pop(), which we just introduced in Point 7 and 8 in the code explanations above on this page.



### Section 2.3 pop() – Delete and return the most recently inserted item and reduce size by 1

---

- pop() functions will delete one recently inserted element from the stack and **return that element** and decreases the stack size by one:

```
1. int pop (int stack[]){  
2.     int x = stack[stack_size];  
3.     stack_size--;  
4.     return x;  
5. }
```

- Done!

**Question: Why I find that the pop() function on the internet will not return anything, but just simply delete one item from stack?**

Ans: Good observation! The reason why we will return the item deleted is for the sake of the construction of top() below! Having returned value makes us easier and interesting to implement the top().



### Section 2.4 Top() – returns the top element / most recently inserted element of stack

---

- top() functions will return the upper element in the stack, or more precisely speaking, return the most recently inserted element in the stack.

```
1. int top (int stack[]){  
2.     int temp = pop(stack);  
3.     push(stack, temp);  
4.     return temp;  
5. }
```

- top() is achieved by first popping out the recently inserted element and store it into a temp variable, then we do a push() to insert the element back to the stack and finally we return the temp variable. So finally, it **will NOT change any elements or the size of the stack**. Done!

So finally, please add all these functions definitions into stack.c and submit it to us.



## Checkpoint 2a – A simple application of stack using stack.h

- In this checkpoint, we will utilize the stack.c file that we just created. And **apply stack to reverse an integer**😊. Please type the following on terminal:

```
$ cp stack.c stack.h
```

- We copy the stack.c to a header file called stack.h, then we open it

```
$ gedit stack.h &
```

- Then, add the pre-processing command `#ifndef .....#define .... #endif` to the stack.h in order to avoid repeatedly declaring variables.

```
1. #ifndef _STACK_H
2. #define _STACK_H
3. #include<stdio.h>
4. #include<string.h>
5. #define stack_storage 1000
6. ....
7. ....
8. }      //last line of the program
9. #endif
```

- After that, please **delete the whole main function** in the header file :

```
0.  #ifndef _STACK_H
0.  #define _STACK_H
1.  #include<stdio.h>
2.  #include<string.h>
3.  #define stack_storage 1000
4.  int stack_size = -1;
5.  void push(int stack[], int user_input);
6.  int pop(int stack[]);
7.  int top(int stack[]);
8.  int main(){
9.  ...
41. printf("\n");
42. return 0;
43. }
44. void push (int stack[], int user_input){
45. ...
46. }
47. int pop(int stack[]) {...}
48. int top(int stack[]) {...}
49. #endif
```

stack.h

- Then let's create another C file called StackApp.c and open it!

```
$ touch StackApp.c
```

- Let's copy the following code into the StackApp.c. In this program, we will reverse the number that the user inputted and we assume that the number will not have digits more than 100 units. (Actually you can change to any number of digits you like ☺). We will first treat the number inputted as a string for the sake of convenience.

```
#include<stdio.h>
#include<string.h>
#include"stack.h"
int main(){
    char word[100];
    printf("Please input a number: ");
    scanf("%s", word);
    //Here we start using the stack
    //...
    //...
    return 0;
}
```

- Please copy the following code under the comment:

```
int stack_size = (int)strlen(word);
int stack[stack_size];
for(int cnt = 0; cnt < stack_size; cnt++){
    push(stack, word[cnt] - '0');
}
for(int cnt = 0; cnt < stack_size; cnt++){
    printf("%d", pop(stack));
}
printf("\n");
```

So let me explain how it works. We create a stack according to size of user's input (Say if A inputs 6798, then the stack size is 4). Then we push every single digit into the stack, the sequence is 6, 7, 9, 8. Since stack follows LIFO (Last-in-first-out), when we pop the stack, it pops 8, 9, 7, 6 in sequence ! Done!





- You can now run your program, and you should be able to have the same result as mine 😊

```
$ gcc StackApp.c -o StackApp
$ ./StackApp
Please input a number: 1234567899999999999
999999999987654321
```

**Questions: It seems to be very easy to use a Stack !  
But why do we have to learn this ?**

Ans: Using a stack actually is not that easy, if you take Comp2119 – Algorithms and data structures, you will learn more about the powerful usage of stack. (Many advanced data structures are also implemented by Stack!). At this point, if you are interested, you can check out the link below:

<https://www.quora.com/What-are-the-real-life-applications-of-stack-data-structure>



Please submit StackApp.c and stack.h



**Congratulations! You  
have successfully  
completed all the  
labs , hope you enjoy.**



## Reference:

1. GeeksforGeeks. (n.d.). *Stack Data Structure*. Retrieved December 20, 2018, from <https://www.geeksforgeeks.org/stack-data-structure/>
2. Quora. (Nov 15, 2016). *What are the real life applications of stack data structure?*. Retrieved December 20, 2018, from <https://www.quora.com/What-are-the-real-life-applications-of-stack-data-structure>

Final feeling after completing this assignment😊:

First I really appreciate the course designer designing every Lab sheet in a great details! Since after completing this assignment, I know it's absolutely time-consuming.... And they created more than 40 hours of lab sheets ! – But I just created this 1-hour lab sheet – And it already spent me nearly half a week to do ..... From designing the outline, to creating the sample programs and making the slides and presentation clear... All take so much effort and energy. All I want to say is, thank you Dr Chim, for his approval of using his slides' design, thank you the course designer (I am not sure who he/she is) for putting so much effort in this course. And thank you Dr Chim to let me have this opportunity to create this lab sheet. It is fun, challenging and rewarding.

I really treat myself as a teacher while I am designing this lab myself. So I really hope that my assignment could be used by others to learn more about programming. It is full of fun😊