Final Topics

Structures

What if we want to have a custom data type that holds combination of different types of data

⇒ Use key word **struct**

- Each attributes of struct is called a member (attributes)

Declaring Struct / Variables

```
struct  Struct Name
{
    < attribute declarations > ;
} < instances list > ;
```

} Definition of Structure Template (w/ member)

} Declaration of Struct instances

- Instance list is optional while creating struct. We can initialize instances there, or outside.

- To initialize instances :

```
struct structname  instance name = { attribute1, ... };
```

```
int  x = 5   (similar !)
```

- To access the attributes : Don't need to state structure.

```
instancename . attribute ;
```

NOTE If attribute is an array (strings), we still

CANNOT do studentName . firstName = "Hi"

CAN do studentName . firstName [0] = 'L'

Type Def

A way to avoid having to write "struct" everytime we initialize a new instance

typedef <structure definition> <type name>;

e.g.

```
typedef struct student {
    char firstName[20];
    char lastName[20];
    int mark;
} Student;
```

⇒ Now we can initialize instances by:

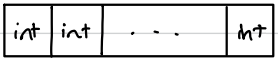
Student studentA;

We can also typedef after a structure has been defined.
We can actually typedef any type with a different name.

```
typedef struct student Student;
typedef int my_int;
```

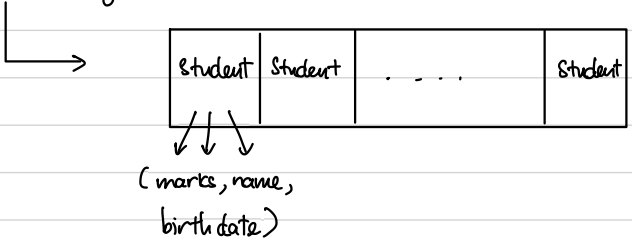
Array of Structures

Like creating an array of other data types, like

int marks[25] ⟶ 

We can do:

Struct Type (arrayname) [size];



Access each student's attributes: arrayname [index] . member ;

Pointers and Structures

Like we can have a pointer pointing to an int by using int *, I can create a pointer and point it to a Structure Instance

```
Student *myStudentPointer;
myStudentPointer = &studentA;
```

AND we can dereference & change the attributes.

Note Precedence →

```
(*myStudentPointer).mark = 88
```

or

```
myStudentPointer->mark
```

Dynamic Memory Structure

```
Student * studentBPointer = (Student *) malloc (sizeof (Student))
```

Create structure array with Dynamic memory

```
Student * studentarray = (Student*) malloc ( size * sizeof (Student));
```

size → # of Students

Calling free

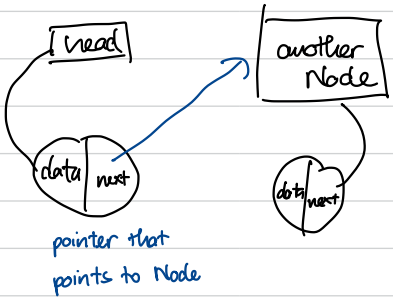
```
free (studentarray);
```

Linked Lists

List is a sequence of objects of the same type, or nodes, with a set of operations on these nodes.

To do this, we introduce a pointer to the attribute of a struct, which points to an instance of that struct

```
typedef struct node {
    int data;
    struct node * next;
} Node;
```



1. Initialize a Node

```
int main() {
    Node * head = (Node*) malloc (sizeof (Node));

    head -> data = 5;
    head -> next = NULL;        // (*head).next
```

2. Add a node to next

```
Node * anotherNode = (Node*) malloc (size of (Node));
anotherNode -> data = 6;
anotherNode -> next = NULL;
head -> next = anotherNode;
```

That's why!

- next is a pointer. It needs an ADDRESS of the (next node). That ADDRESS is stored in that (next node)'s pointer.

Linked List Operations

1. Initialize a Linked List.

```
typedef struct node {
        int data;
        struct node* next;
} Node;
```

Node type c

```
Node* mylist = NULL;
```

mylist, first has nothing

```
Node* head = (Node*) malloc (size of (Node));
head → data = 5;
head → next = NULL;
```

create head Node*

```
mylist = head;
```

mylist starts at Node* head

2. Function to create new Node : (feeding data & address of next node)

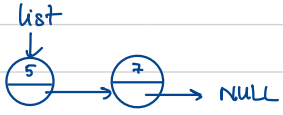
```
Node* newNode ( int data , Node* next ) {
        Node* node = (Node*) malloc ( size of (Node));
        node → data = data ;
        node → next = next ;
return node;
}
```

pointer to

Note: Since we are creating a node ∧ malloc, the data,... can be stored even outside of function ⇒ we can do this

Now we can create a linked list with 2 nodes as:

```
Node * list = NULL;
list = new Node (5, NULL);
list -> next = new Node (7, NULL);
```



We can continue by `list → next → next = new Node ( , );`

3. Assign a node to head of list

```
Node * addtohead ( int data , Node* head ) {
        Node* newhead = new Node ( data , head );
return newhead;
}
```

We CANNOT change head to newhead IN FUNCTION, because we are passing the address of head to function. When we do head = newhead in function, it's changing the address of head in local, not main

⇒ Must use outside : `head = addtohead ( 7, NULL);`

The only way to use it is to **, * and & the head.

4. Traversing a linked list

```
void traverse ( Node* head ) {
    Node* on = head ;
    while ( on != NULL) {
        <do something> ;
        on = on → next ;
    }
}
```

create an on POINTER

IN SUMMARY, when we're passing head to a function, we're passing the ADDRESS that head is pointing to.

This means we can use that passed address to change the list, but NOT head.

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct node{
    int data;
    struct node* next;
}Node;

Node* newNode (int data, Node*next){
    Node* newnode = (Node*)malloc(sizeof(Node));
    newnode->data = data;
    newnode->next = next;
}

Node* addtoHead (int data, Node* head){
    Node* newhead = newNode (data,head);
    return newhead;
}

void print(Node*head){
    Node* current = head;
    while (current != NULL){
        printf("%d ",current->data);
        current = current->next;
    }
}

int main()
{
    Node* list = NULL;
    list = addtoHead (5,list);
    list = addtoHead (7,list);
    list = addtoHead (9,list);
    print(list);
}
```

A node contains data and has a pointer that can point at another Node

Creating Node pointer by specifying data it holds and give the address of the Node to be pointed at. This address is the pointer name to that Node. Reason why we need pointers because we form Nodes on malloc.

Pass data for new node, along with address of current head (current pointer name). Set new pointer-to-Node newhead through newNode above. Return pointer.

current pointer = head means it also points to first node (the one head's pointing). → data and → next follows

Again, all Nodes we work on are pointers to Node. "list" is a pointer. EVERYTHING is pointer.

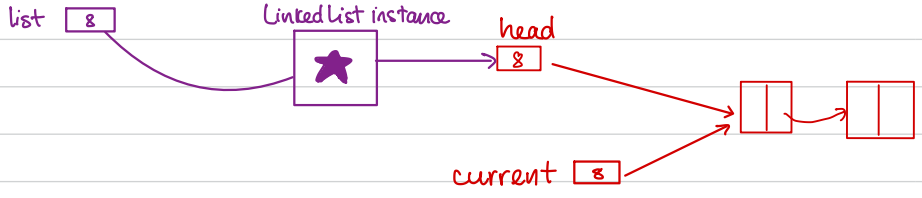
Addresses to Nodes

Linked List Type

We can change the head position in functions by a new type, LinkedList. that holds the head address (pointing to head).

```
typedef struct linkedList {
    Node* head;
} LinkedList;
```

LinkedList is a pointer of type Node which could point to a type Node structure.



Now, at main:

```
LinkedList * list = (LinkedList*) malloc (sizeof (LinkedList));
```

- As we pass head to a function, we pass it to LinkedList* list at the function parameter (to connect & fill ★)
- Now instead of calling head, we call list → head
- Since list contains the address of head, we can change where head points to

⇒ A pointer, or a block in memory, can be changed as long as we have its address.

```c
#include <stdio.h>
#include <stdlib.h>
typedef struct node{
    int data;
    struct node* next;
}Node;

typedef struct linkedlist{
    Node* head;
}LinkedList;

Node* newNode (int data, Node*next){
    Node* newnode = (Node*)malloc(sizeof(Node));
    newnode->data = data;
    newnode->next = next;
}

void addtoHead (int data, LinkedList* list){
    Node* newhead = newNode (data,list->head);
    list->head = newhead;
}

void print(LinkedList* list){
    Node* current = list->head;
    while (current != NULL){
        printf("%d ",current->data);
        current = current->next;
    }
}

int main()
{
    LinkedList* list = (LinkedList*)malloc(sizeof(LinkedList));
    list->head = NULL;

    addtoHead (5,list);
    addtoHead (7,list);
    addtoHead (9,list);
    print(list);
}
```

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct node{
    int data;
    struct node* next;
}Node;

Node* newNode (int data, Node*next){
    Node* newnode = (Node*)malloc(sizeof(Node));
    newnode->data = data;
    newnode->next = next;
}

Node* addtoHead (int data, Node* head){
    Node* newhead = newNode (data,head);
    return newhead;
}

void print(Node*head){
    Node* current = head;
    while (current != NULL){
        printf("%d ",current->data);
        current = current->next;
    }
}

int main()
{
    Node* list = NULL;
    list = addtoHead (5,list);
    list = addtoHead (7,list);
    list = addtoHead (9,list);
    print(list);
}
```

Functions applied to Linked List.

1. Initialize LinkedList

```
LinkedList* initlist ( ) {
    LinkedList* list = (LinkedList*) malloc (sizeof(LinkedList));
    list -> head = NULL;
return list;
}
```

2. Is empty :

```
return ( list -> head == NULL );
```

3. Inserting at the end

```
void insertatEnd(int data, LinkedList* list){
    Node*newnode = newNode(data,NULL);
    Node* current = list->head;
    while (current->next != NULL){
        current = current->next;
    }
    current->next = newnode;
}
```

but we have to count when LIST IS EMPTY

```
void insertatEnd(int data, LinkedList* list){

    if (isEmpty(list)){
        list->head = newNode(data,NULL);        //or list->head = addtoHead(data,list);
    }
    else{
    Node* current = list->head;
    while (current->next != NULL){
        current = current->next;
    }
    current->next = newNode(data,NULL);
    }
}
```

4. deleteAtHead

```
void deleteAtHead (LinkedList* list){
    if (isEmpty(list)){
        printf("List is empty");
        return;
    }
    else{
        Node* toRemove = list->head;
        Node* newHead = list->head->next;

        free(toRemove);
        list->head = newhead;
    }
```

5. deleteAtTail

```
void deleteAtTail (LinkedList*list){
    if (isEmpty(list)){
        printf("List is empty");
        return;
    }
    else if (list->head->next == NULL){
        list->head = NULL;
        return;
    }
    else{
        Node* on = list->head;
        while (on->next->next != NULL){
            on = on->next;
        }
        Node* toRemove = on->next;
        on->next = on->next->next;
        free(toRemove);
    }

}
```

6. deleteAllNodes

```
void deleteAllNodes(LinkedList *list) {
    while (!isEmpty(list)) {
        deleteAtHead(list);
    }
}
```

(A) Always think of reusing previous functions

```
typedef struct node {
    int data;
    struct node *next;
} Node;

typedef struct linkedList {
    Node *head;
} LinkedList;

Node *createNode(int value);
void initList(LinkedList *list);
bool isEmpty(LinkedList *list);
bool insertAtFront(LinkedList *list, int value);
bool insertAtBack(LinkedList *list, int value);
void deleteFront(LinkedList *list);
void deleteBack(LinkedList *list);
void printList(LinkedList *list);
int deleteAllNodes(LinkedList *list);
```

Sophisticated Functions

1. findFirstNode

```
Node* findFirstNode(LinkedList* list, int value){
    Node* on = list->head;
    while (on != NULL){
        if (on-> data == value){
            return on;
        }
        on = on->next;
    }
    return NULL;

}
```

2. deleteFirstMatch (fix to return bool)

```
void deleteFirstMatch (LinkedList* list, int value){
    Node* on = list->head;
    while (on != NULL){
        if (on->next->data == value){
            Node* toRemove = on->next;
            on->next = on->next->next;
            free(toRemove);
            return;
        }
        on = on->next;
    }
    printf("could not find the value");
}
```

3. int deleteAllMatches

```
int deleteAllMatches(LinkedList *list, int value) {
    int count = 0;

    while (deleteFirstMatch(list, value)) {
        count++;
    }

    return count;
}
```

4. insert To Ordered List

Care about: +) smaller than 1st node

+) Empty

+) next = null

+) Normal

5. copy list

6. join list (ascending)

Recursive List Processing

Recall: A recursive process includes:

1. A case in which the process is defined in terms of simpler version of itself.
2. Solution to the simpler case that terminates the recursion, base case

We also know that a linked list has NULL that identifies empty / end of list. with non-Null reference to a node with a link to another node.

⇒ Processing linked list can be employed with recursion.

```
// rewrite using recursion
void printRec(Node*node){
    if (node == NULL){
        return;
    }
    printf("%d ",node->data);
    printRec(node->next);
}
```

When we pass, we're passing list → head. Why is head not changed in global?

- Add into ordered list with Recursion

```
Node * addIntoOrderedListRec(Node *head, int data) {
    if (head == NULL || data < head->data) {
        return newNode(data, head);
    }

    head->next = addIntoOrderedListRec(head->next, data);
    return head;
}
```

To call Rec on LinkedList, make the Node* head functions become Helper function. Then in the list Rec function we do:

```
void printRec (LinkedList *list) {
    printRecHelper (list->head);
}
```

trigger Node