Intro to C

Variable Declarations

- "normal"
  - int x
- pointer
  - int* x
    - **NOTE:** is not initialized yet!!

Pointers

```
int *a;
int c=10;
a = &c; //getting the address of c
printf("The address %p\n", a);
printf("The actual value %d\n", *a); //content
```

Uninitialized Pointers

- getting "garbage"

  ```
  int a[8];
  printf("a[1] will get garbage: %d", a[1]);
  ```

- causing a segmentation fault

  ```
  int* d;
  printf("%d", *d); //would cause a seg fault
  ```

Pointer as a Parameter

```
int example(int* pointerThing)
```

- sending the address of pointerThing
- any changes made to pointerThing within the method will persist
  - this is unlike non-pointers which are passed by value

Static Array Allocation

```
 int a[8]; //allocating array on the STACK
 int i;
for (i =0; i<8; i++) { //loop over and set positions
        a[i] = i;
   printf("%d ",a[i]); //output: 0 1 2 3 4 5 6 7
}
```

Dynamic Array Allocation

```
int size =10;
int* b = (int*) malloc(sizeof(int)*size);
```

- can cast pointer if you wish
- doing *(b+1) is the same as b[1]

NOTE: arrays in C have no bounds checking or length/size property

Structs

```
struct pillowPet {
        int size;
        char *color;
};
struct pillowPet n;
struct pillowPet* p = &n;
```

accessing struct members

- not a pointer
  - . notation
  - ex: n.size
- pointer
  - -> notation
  - ex: p->size

Linked List Implementation

What does a LL need?

- a node struct
- a head pointer
- insert method
- remove method
- print method - not required by helpful

NOTE: if reviewing these slides try writing implementation yourself before going on

Node Struct

```
typedef struct node{
        int val; //value held by this node
        struct node* next; //the next node
} node;
```

NOTE: typedef will keep you from having to type struct in front of every declaration of a node

Append Function

```
//insert a node at the end of the list
//returns the newly added node
node* append(node* head,int value){
        node * newNode = (node *)malloc(sizeof(node));
        newNode->val = value;
        newNode->next = NULL;
    if(head == NULL){ //list empty
          return newNode;
        }
//adding node to end of list otherwise
node* iter = head;
while(iter->next != NULL){
        iter = iter->next;
}
iter->next = newNode;
return newNode;
```

Remove Function

```
//takes a node out of the list
void remove(node* head, node* elem){
        if(elem == NULL) return;
        node* iter = head;
        //find the node b4 elem

        while(iter != NULL && iter->next != elem){
                iter = iter->next;
```

```
        }
        if(iter == NULL){//not in list
                return;
        }else{//elem in middle
                iter->next = elem->next;
                free(elem);
        }
}
```

Print Function (optional)

```
void print(node* head){
        node* temp = head;
        while(temp != NULL){
                printf("%d--->",temp->val);
                temp =temp->next;}
        printf("\n");
}
//example output: 1--->2--->3
```

Sample Main

```
int main(){
        node* head = insert(NULL, 5);
        node* n2 = insert(head, 4);
        node* n3 = insert(head, 2);
        print(head);
        removenode(head, n2);
        print(head);
        removenode(head, n1);
        print(head);
}
```

NOTE: feel free to try your own implementation.

Maybe even do a doubly linked list.

SUGGEST AN EDIT