LinkedList

```
#include<iostream>
using namespace std;

class node{

public:
    int data;
    node *next;

    node(int val){
        data = val;
        next = NULL;
    }

};

class list{
    node* head;
    node* tail;

public:
    list(){
        head = NULL;
        tail = NULL;
    }

    void insert_begin(int val){
        node* newNode = new node(val);

        if(head == NULL){
            head = tail = newNode;
        }
        else{
            newNode->next = head;
            head = newNode;
        }
    }

};
int main() {
    list myList; // Create a list object
    myList.insert_begin(10);
    myList.insert_begin(5);
    return 0;
}
```

Excellent question. Let's dive deep into what's happening at the memory level. This is the key to truly understanding pointers and data structures like linked lists.

We need to talk about two main areas of memory: the **Stack** and the **Heap**.

- **The Stack**: 🥞 This is a highly organized, fast region of memory. It's used for static memory allocation. Think of it as a stack of plates. When a function is called, a new "plate" (a stack frame) is placed on top. All the function's local variables (like `int x`, `node* newNode`) are stored on this plate. When the function finishes, its plate is removed, and all those local variables are instantly destroyed.
- **The Heap**: disorganized pile of memory. It's used for dynamic memory allocation, which is what happens when you use the `new` keyword. Memory allocated on the heap is not managed automatically. It stays there until you explicitly release it with the `delete` keyword. This allows data to persist even after the function that created it has finished.

Let's trace the execution of your code with a concrete example.

Imagine we have a `main` function like this:

C++
```
int main() {
    list myList; // Create a list object
    myList.insert_begin(10);
    myList.insert_begin(5);
    return 0;
}
```

Step 1: `list myList;`

When this line runs, the program allocates memory for our `list` object. Since `myList` is a local variable in `main`, it's created on the **Stack**.

A `list` object contains two pointers: `head` and `tail`. The `list()` constructor runs and sets both of these to `NULL`. In memory, `NULL` is essentially the address `0`.

Memory Snapshot 1: After creating an empty list

Memory Area	Address	Variable	Value
Stack	`0x7FF_A`	`myList.head`	`0x0000` (NULL)
	`0x7FF_B`	`myList.tail`	`0x0000` (NULL)
Heap	-	-	*(Completely empty for now)*

Step 2: `myList.insert_begin(10);`

Now things get interesting. The `insert_begin` function is called.

1. **Function Call**: A new stack frame for `insert_begin` is pushed onto the Stack. The parameter `val` is created within this frame and holds the value `10`.
2. **`node* newNode = new node(10);`**: This is a two-part command.
   - **`new node(10)`**: You've asked for memory from the **Heap**. The system finds a free block of memory large enough for a `node` object (one integer and one pointer). Let's say it finds this memory at address **`0xABCD`**.
   - The `node` constructor runs on this memory block at `0xABCD`. It sets `data` to `10` and `next` to `NULL` (`0x0000`).
   - **`node* newNode = ...`**: The `new` operator returns the address it found: `0xABCD`. This address is stored in the local pointer variable `newNode`, which lives on the **Stack** inside the `insert_begin` frame.
3. **`if (head == NULL)`**: The code checks `myList.head`. Looking at our table, it is indeed `NULL`. The condition is **true**.
4. **`head = tail = newNode;`**: This line copies the *value* of `newNode` (the address `0xABCD`) into `myList.head` and `myList.tail`.

Memory Snapshot 2: During `insert_begin(10)`

Memory Area	Address	Variable	Value
Stack	`0x7FF_C`	`insert_begin::newNode`	`0xABCD` (An address)
	`0x7FF_D`	`insert_begin::val`	`10`
	`0x7FF_A`	`myList.head`	`0xABCD`
	`0x7FF_B`	`myList.tail`	`0xABCD`
Heap	`0xABCD`	`(node allocated with new)`	`data=10, next=0x0000`

5. **Function Return**: The `insert_begin` function is finished. Its stack frame is popped off and destroyed. The local variables `newNode` and `val` cease to exist. **BUT**, the memory at `0xABCD` on the heap is unaffected! `myList` still holds the address, so we haven't lost our node.

Step 3: `myList.insert_begin(5);`

Here we go again, adding a second node.

1. **Function Call**: A new stack frame for `insert_begin` is pushed onto the Stack with `val = 5`.
2. **`node* newNode = new node(5);`**:
   - **`new node(5)`**: The system finds another free block on the **Heap**. Let's say this one is at address **`0xEFF0`**.
   - The `node` constructor runs on the memory at `0xEFF0`, setting `data` to `5` and `next` to `NULL`.
   - **`node* newNode = ...`**: The address `0xEFF0` is stored in the new local `newNode` pointer on the Stack.
3. **`if (head == NULL)`**: We check `myList.head`. It currently holds `0xABCD`, so it is not `NULL`. The condition is **false**.
4. **`else` block**: The program jumps to the `else` block.
   - **`newNode->next = head;`**: This is the magic link.
     - `newNode->next`: Go to the address in `newNode` (`0xEFF0`) and look at its `next` field.
     - `= head`: Get the address currently in `myList.head` (`0xABCD`).
     - Copy the address `0xABCD` into the `next` field of the node at `0xEFF0`. The new node is now pointing to the old first node.
   - **`head = newNode;`**:
     - `head =`: Look at the `myList.head` variable on the stack.
     - `= newNode`: Get the address currently in the local `newNode` variable (`0xEFF0`).
     - Copy the address `0xEFF0` into `myList.head`. The list's official starting point is now updated.

Memory Snapshot 3: During `insert_begin(5)`

Memory Area	Address	Variable	Value
Stack	`0x7FF_C`	`insert_begin::newNode`	`0xEFF0` (An address)
	`0x7FF_D`	`insert_begin::val`	`5`
	`0x7FF_A`	`myList.head`	`0xEFF0`
	`0x7FF_B`	`myList.tail`	`0xABCD`
Heap	`0xABCD`	`(first node)`	`data=10`, `next=0x0000`
	`0xEFF0`	`(second node)`	`data=5`, `next=0xABCD`

5. **Function Return**: The function finishes, its stack frame is destroyed. The local `newNode` pointer vanishes.

Final State

After all the code runs, our `myList` object on the Stack points to a chain of nodes on the Heap.

- `myList.head` contains `0xEFF0`.
- The node at `0xEFF0` contains `data=5` and `next=0xABCD`.
- The node at `0xABCD` contains `data=10` and `next=0x0000` (NULL).
- `myList.tail` contains `0xABCD`.

Visually, the structure is:

`myList.head` → **[Node (data: 5, next: →)] → [Node (data: 10, next: NULL)]** ← `myList.tail`

This is how pointers allow you to build complex, dynamic structures in memory by simply storing and manipulating addresses.