CMSC202
Computer Science II

Lecture 12 - Linked Lists

CMSC 202 Faculty

Based on slides by Chris Marron, Katie Gibson, and Walter Savitch

Last Class We Covered

- To begin to cover dynamic memory allocation
- Show how to use a destructor (and why)
- Memory Leaks

Any Questions from Last Time?

Today’s Objectives

- Review dynamic memory allocation
- Introduce Linked Lists
  - Creation
  - Insertion
  - Deletion

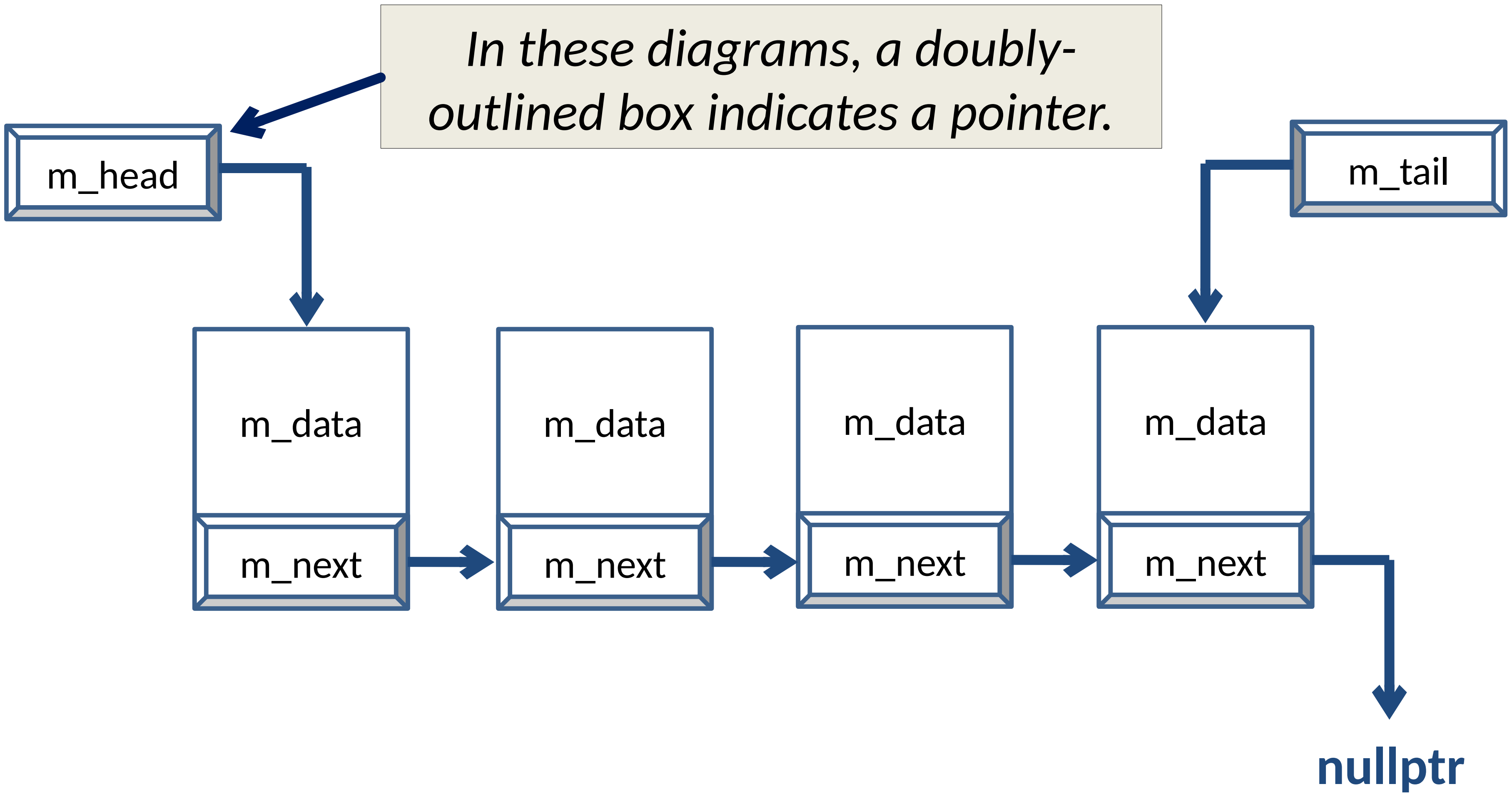
Linked Lists

What is a Linked List?

- Data structure
  - Dynamic
  - Allow easy insertion and deletion

- Uses nodes that contain
  - Data
  - Pointer to next node in the list

Example Linked List

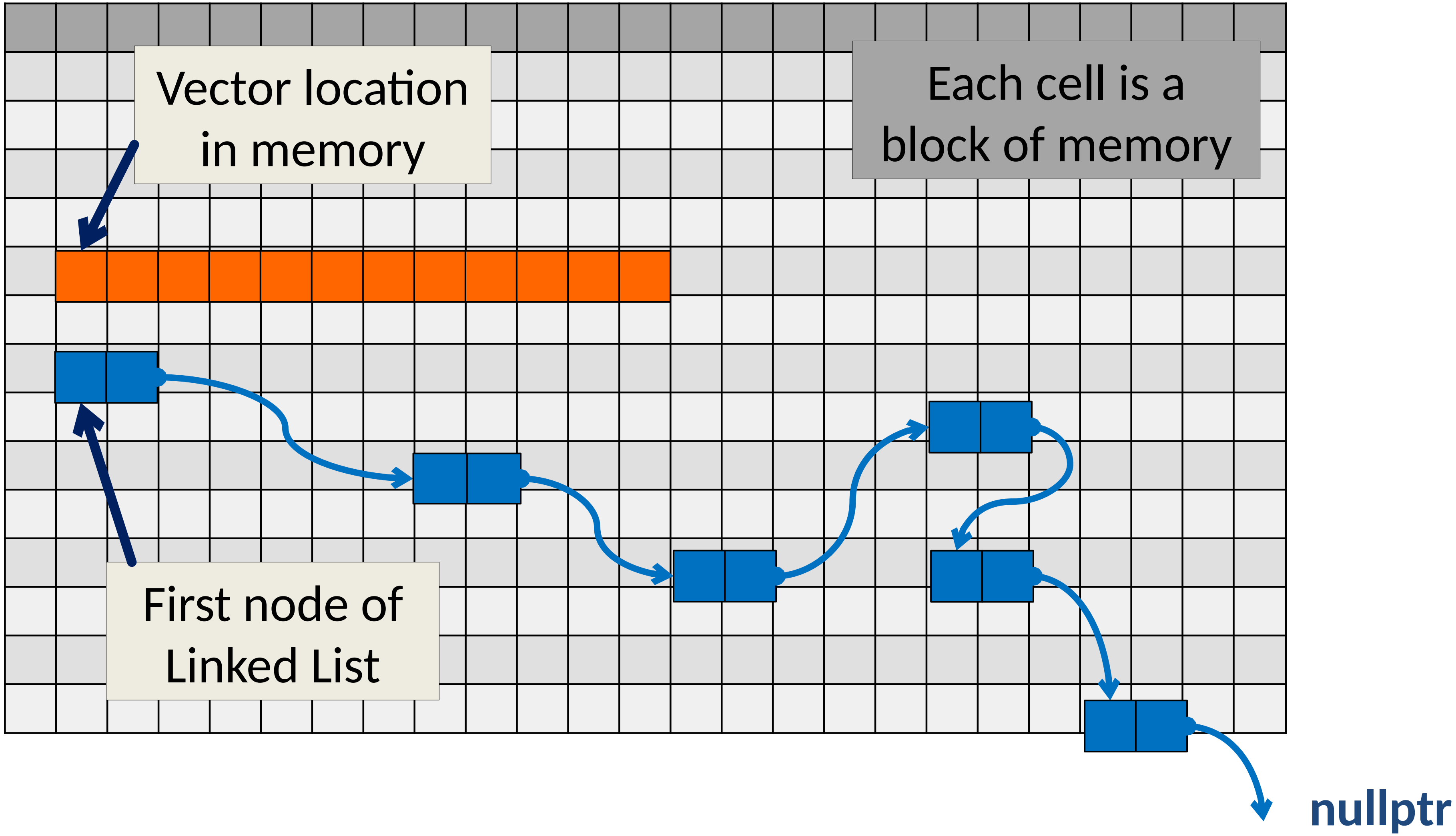


Why Use Linked Lists?

- We already have vectors!

- What are some disadvantages of a vectors?
  - Inserting in the middle of a vector takes time
  - Deletion as well
  - Sorting
  - Requires a *contiguous* block of memory

Representation in Memory



(Dis)Advantages of Linked Lists

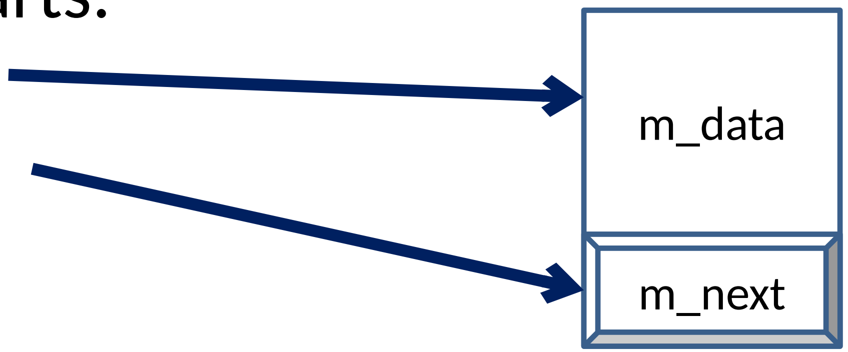
- Advantages:
  - Change size easily and constantly
  - Insertion and deletion can easily happen anywhere in the Linked List
  - Only one node needs to be contiguously stored
- Disadvantages:
  - Can't randomly access data without sequential iteration
  - Requires management of memory
  - Pointer to next node takes up more memory

Nodes

Nodes

- A node is one element of a Linked List

- Nodes consist of two main parts:
  - Data stored in the node
  - Pointer to next node in list

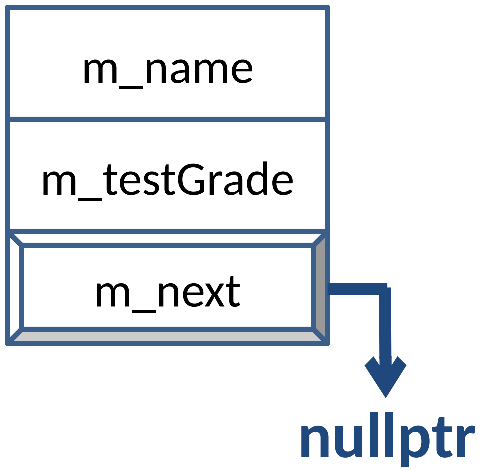


- Often represented as classes
- Sometimes represented as a struct (although all data is public then!)

Code for Node Class

```
class Node {
    string m_name;
    int    m_testGrade;
    Node  *m_next;

    // constructor
    // accessors
    // mutators
};
```



link can point to other nodes

two options:

1. another Node
2. nullptr

Linked List Details

Important Points to Remember

- Last node in the Linked List points to **`nullptr`**

- Each node points to either another node in the Linked List, or to **`nullptr`**
  - Only one link per node

Managing Memory with LLs

- Hard part of using Linked Lists is ensuring that none of the nodes go "missing"
- Think of Linked List as a train
  - (Or as a conga line of Kindergarteners)
- Must keep track of where links point to
- If you're not careful, nodes can get lost in memory (and you have no way to find them)

Linked Lists’ “Special” Cases

- Linked Lists often need to be handled differently under specific circumstances
  - Linked List is empty
  - Linked List has only one element
  - Linked List has multiple elements
  - Changing something with the first or last node

- Keep this in mind when you are coding
  - Dummy nodes may alleviate some of these concerns

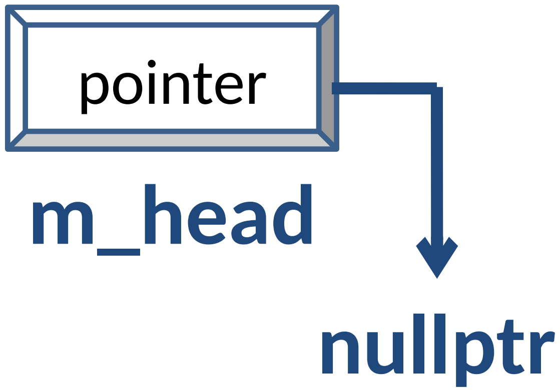
Linked List Functions

- What functions does a Linked List class implementation require?

- Linked_List constructor
- `insert()`
- `remove()`
- `printList()`
- `isEmpty()`

Do we need to delete anything when we are finished with the linked list?

Empty Linked List



Initially, a linked list just has a pointer named m_head that points at nullptr

Linked List Constructor

- There is not a lot to a Linked List Constructor:
- Minimally:

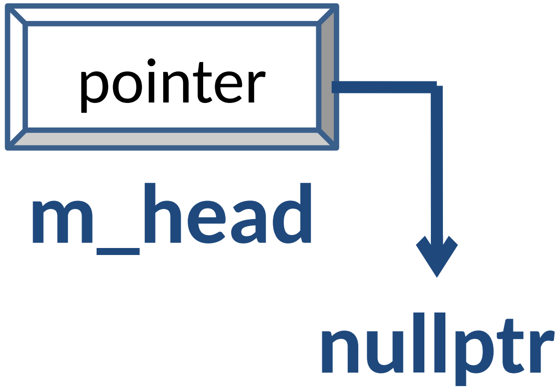
```
LinkedList(){
  m_head = nullptr;
}
```

- More Functional:

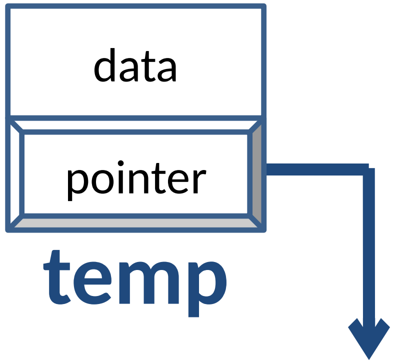
```
LinkedList(){
    m_head = nullptr; //Required for linked lists
    m_tail = nullptr; //Used to insert at the end
    m_size = 0; //Keeps track of the total num of nodes
}
```

Insert Nodes

Insert Nodes



Step 1: Create New Node



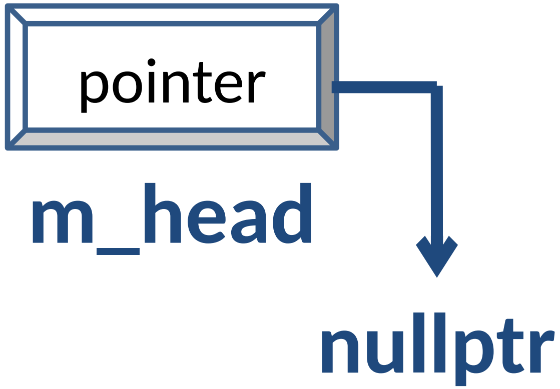
Node *temp = new Node(); //Builds a new node

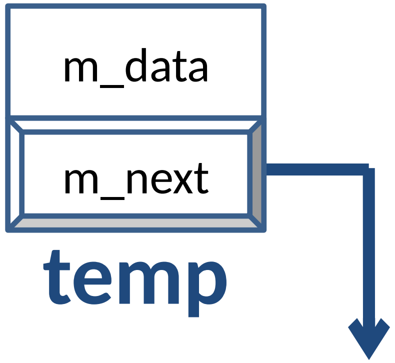
Node Pointer

Dynamically-Allocated Node

Insert Nodes

Step 2: Populate Data





temp->m_data = 10; //Sets new node's data to 10

Node Pointer   Arrow Operator   Data Member Variable

Insert Nodes

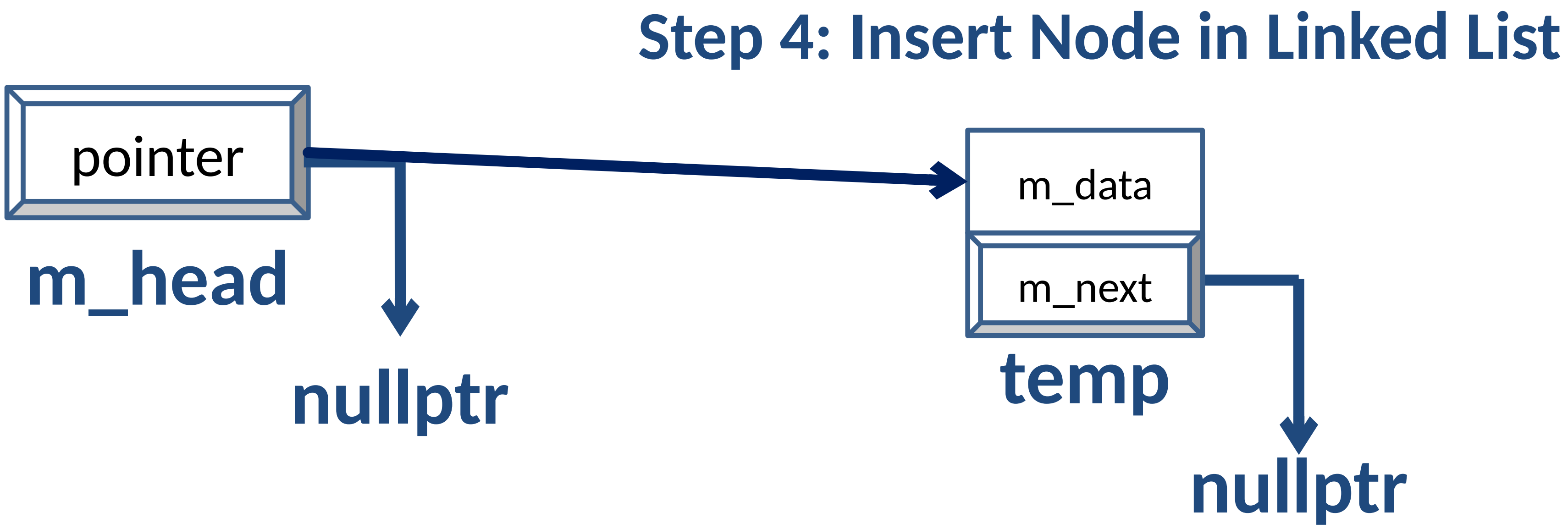
- Once we have the new node instantiated and the data in the node populated, we need to figure out where to insert the new node
- We can insert a new node in three places:
  1. At the beginning of the linked list
  2. At the end of the linked list
  3. Somewhere in the middle of the linked list

Insert Nodes

Step 3: Set m_next to m_head

temp->m_next = m_head; //If first node, points to nullptr
//If LL already has nodes, points to first node

Insert Nodes (Beginning)



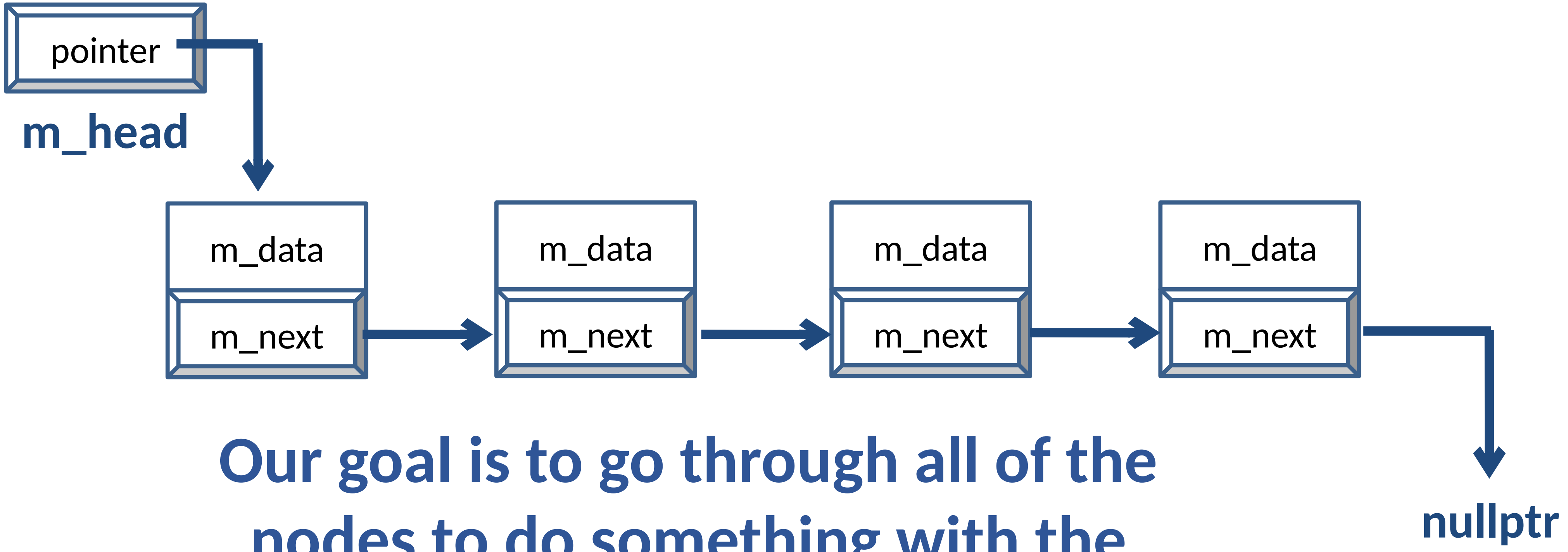
m_head = temp; //Sets m_head to point to new node

Linked List Node Pointer

New Node

Traverse Linked List

Traverse Linked List

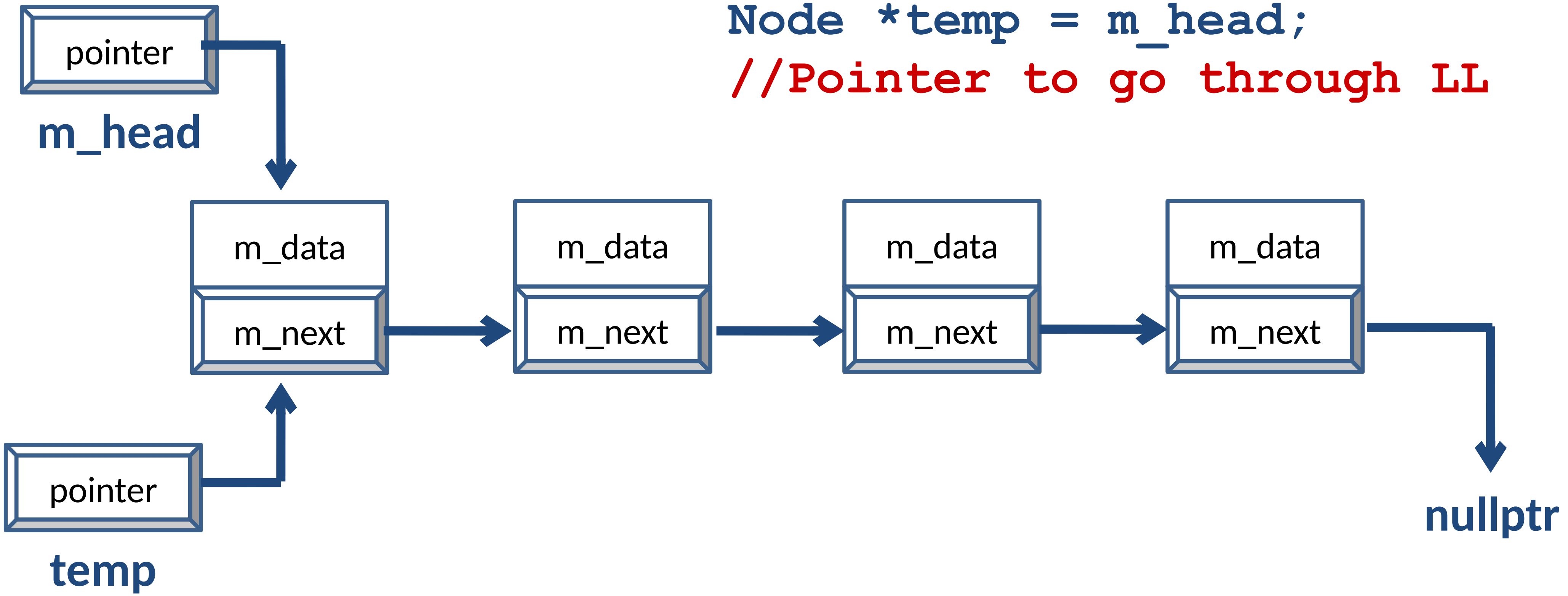


Our goal is to go through all of the nodes to do something with the data (output for example)

Traverse Linked List

Step 1: Create Node Pointer

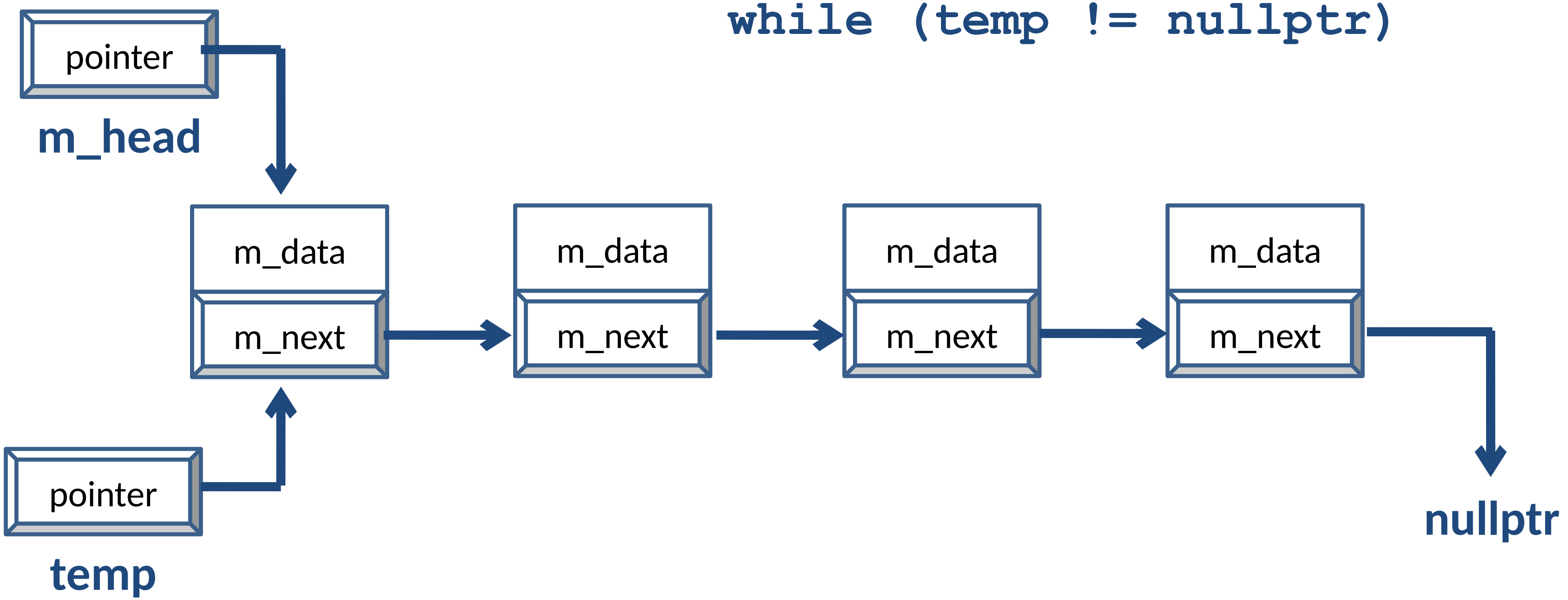
```
Node *temp = m_head;
//Pointer to go through LL
```



Traverse Linked List

Step 2: Create Loop

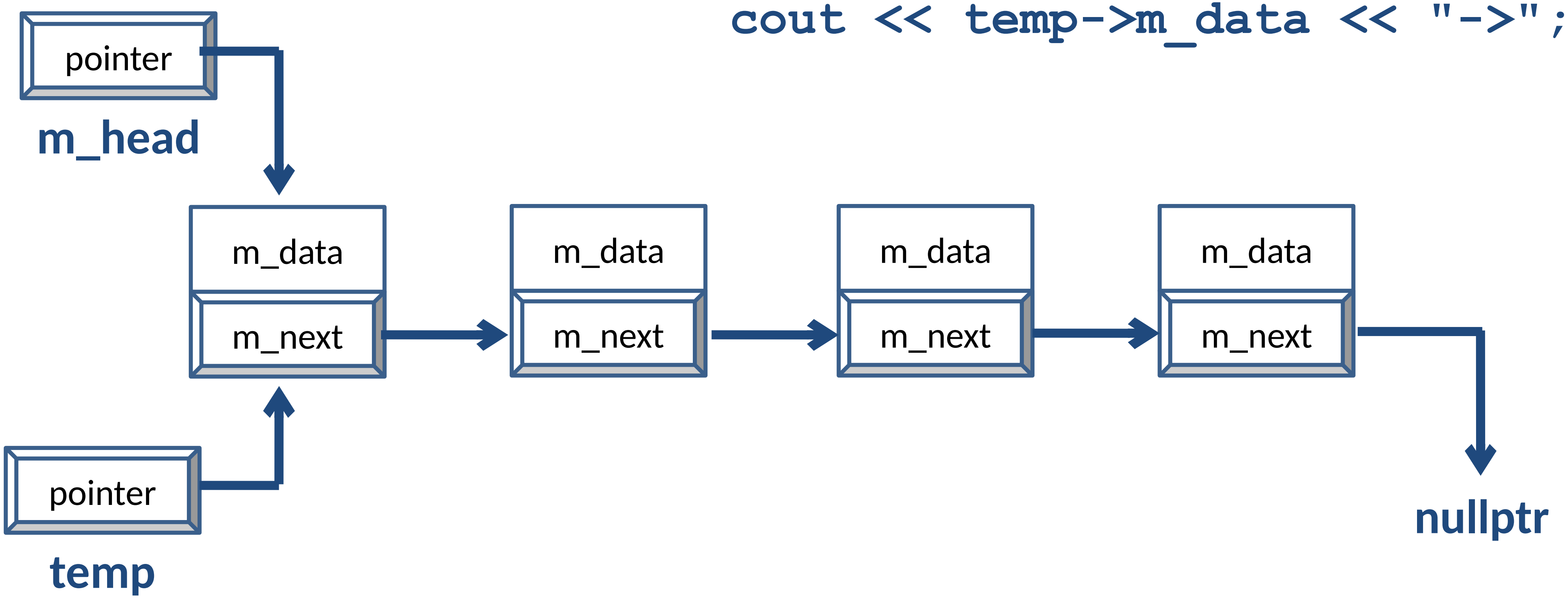
```
while (temp != nullptr)
```



Traverse Linked List

Step 3: Output Data

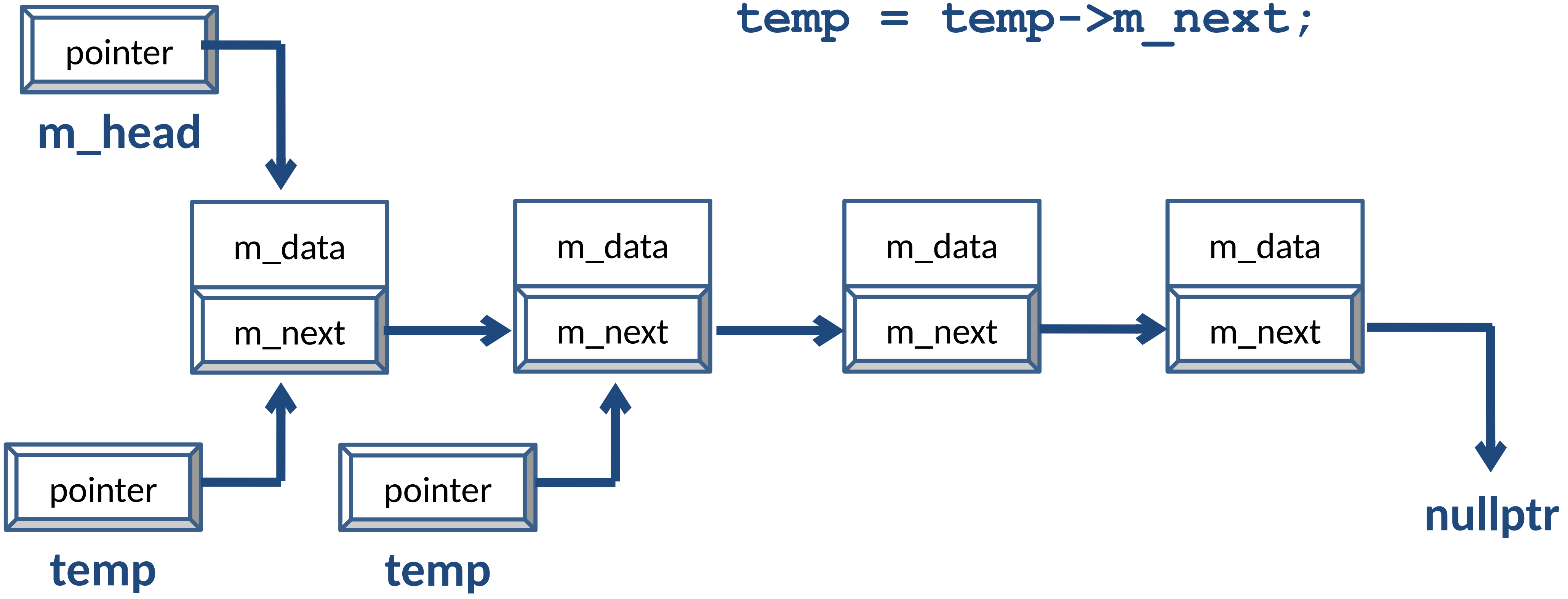
```
cout << temp->m_data << "->";
```



Traverse Linked List

Step 4: Move temp

```
temp = temp->m_next;
```



Live Coding

lec12 –> Linked1.cpp

Remove Node in X Position

Remove Node

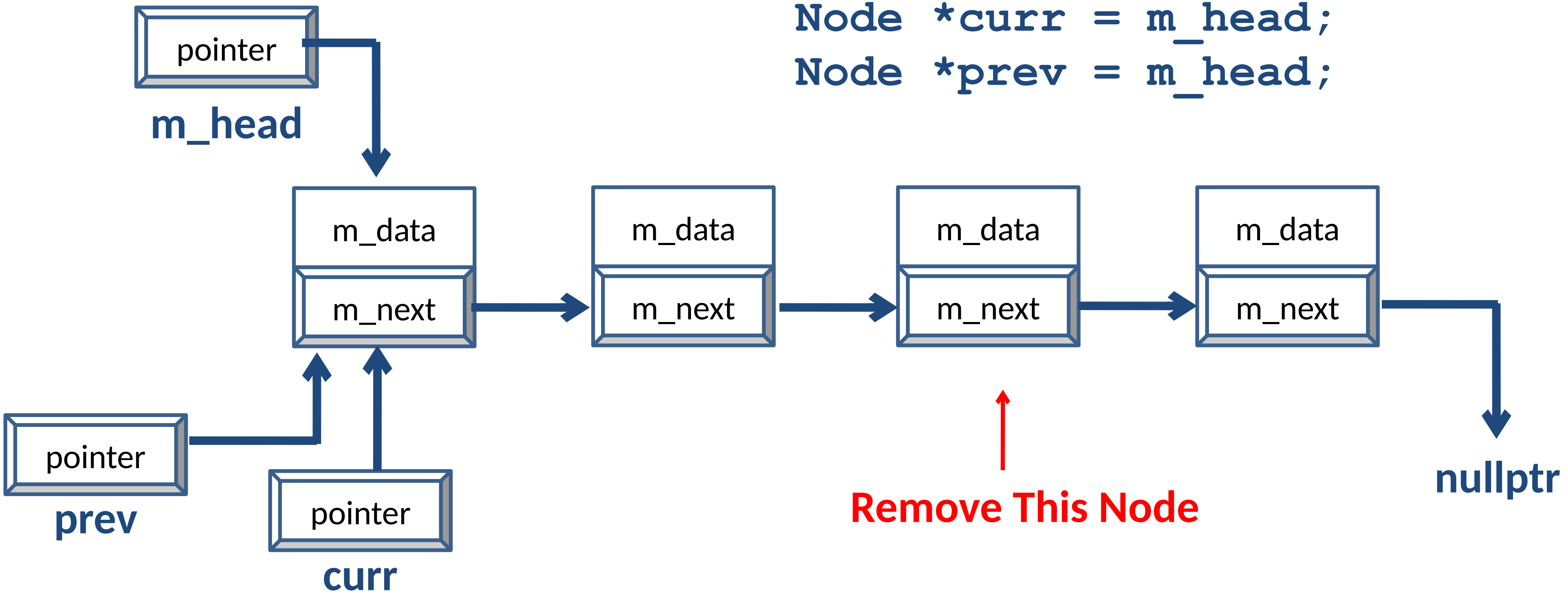
- If we wanted to **remove** a node in a specific location of our linked list, we will have to do some additional logic so that we do not lose our linked list.
- Why?

- When a node is removed from a linked list, we must keep track of the node prior to it. Otherwise, we lose the rest of the linked list.

Remove Node

Step 1: Create Two Node Pointers

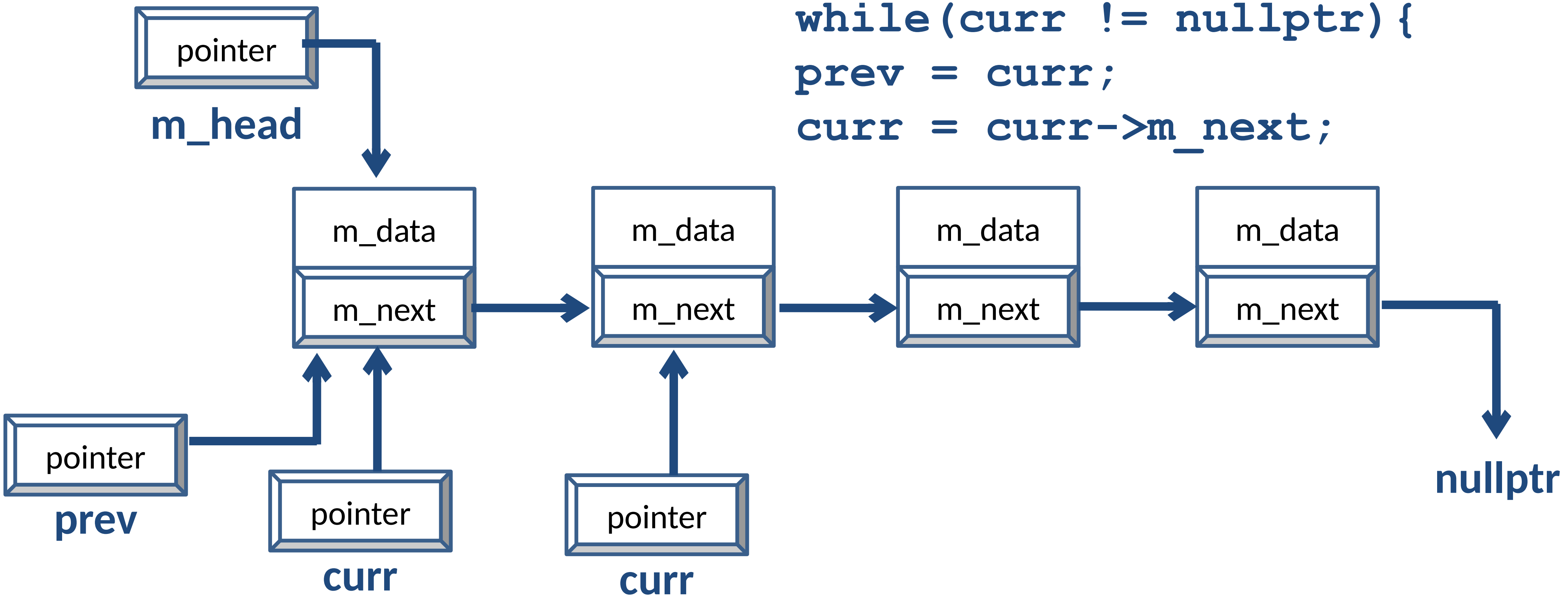
```
Node *curr = m_head;
Node *prev = m_head;
```



Remove Node

Step 2: Iterate until curr is at the node you want to remove

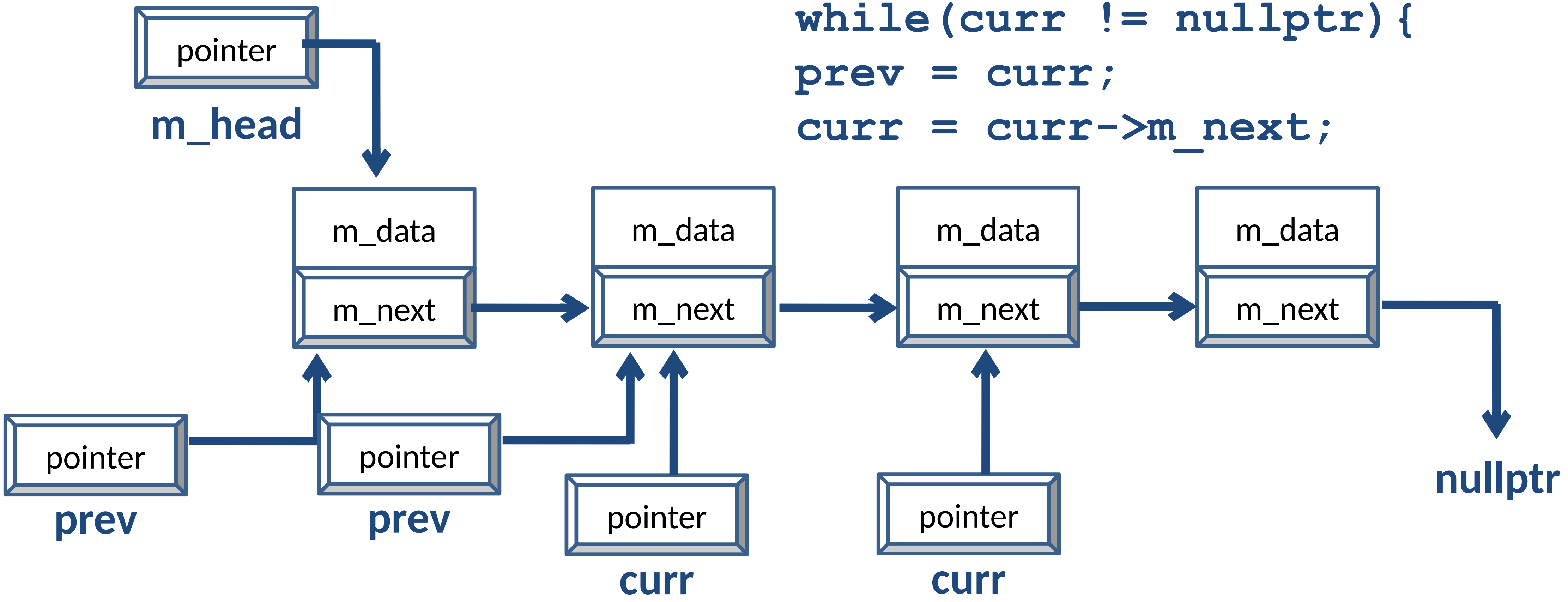
```
while(curr != nullptr){
prev = curr;
curr = curr->m_next;
```



Remove Node

Step 2: Iterate until curr is at the node you want to remove

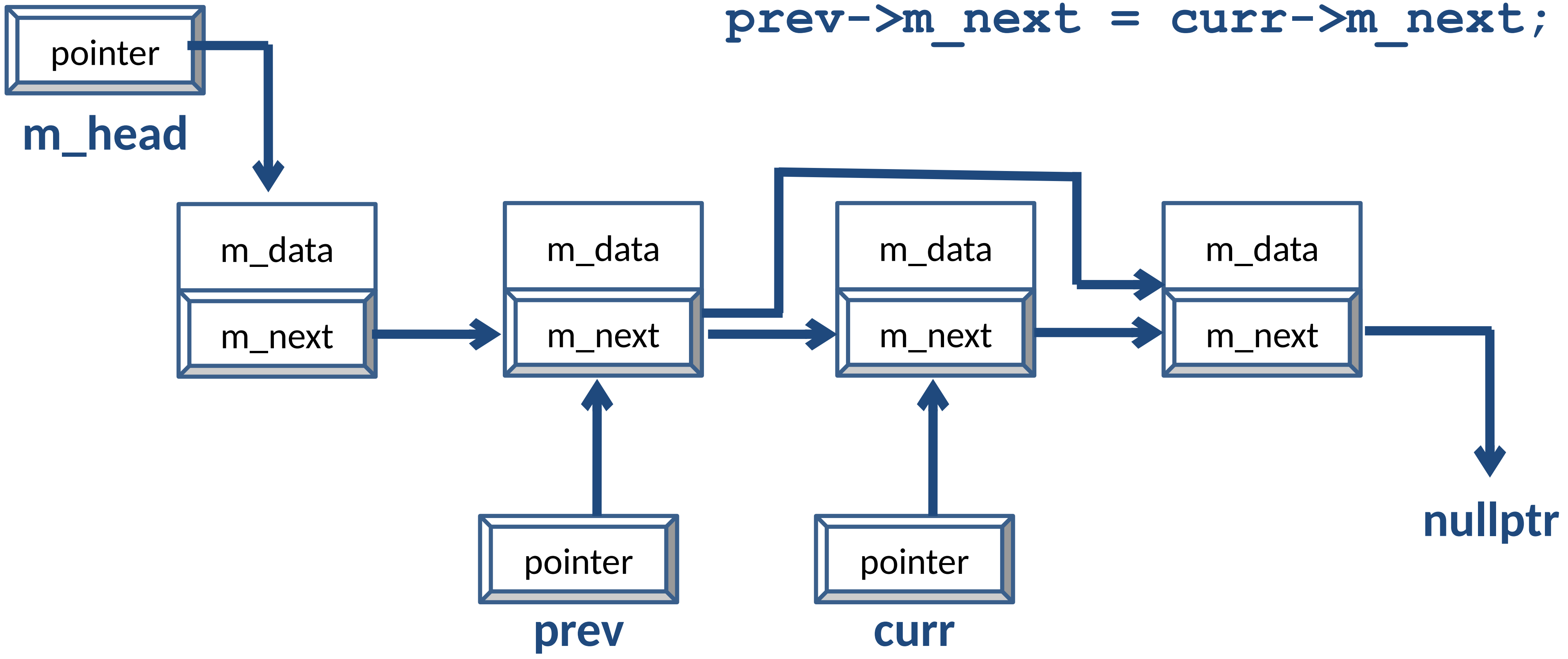
```
while(curr != nullptr){
prev = curr;
curr = curr->m_next;
```



Remove Node

Step 3: Link prev to curr->m_next

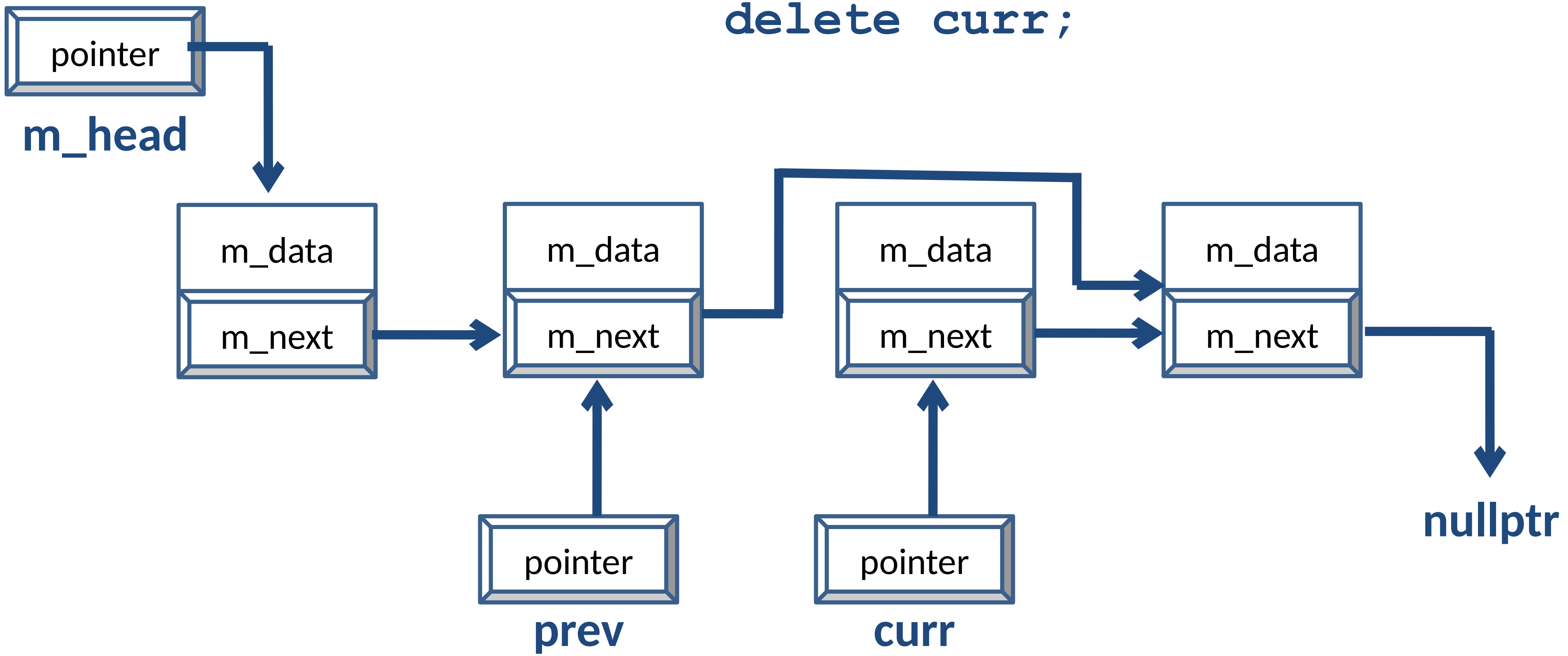
```
prev->m_next = curr->m_next;
```



Remove Node

Step 4: Delete curr

```
delete curr;
```



Live Coding

lec12 –> Linked2.cpp

Announcements

- Prelab Quizzes (4 pts)
  - Released every Friday by 10am on Blackboard
  - Due every Monday by 10am on Blackboard
- Lab (6 pts)
  - In Engineering building during scheduled time!
- Project 2
  - Due on Tuesday, March 11th at 8:59pm on GL
- Exam 2 Review
  - On Friday, April 4th from 2-4pm in LH 1 (Movie Theater)
- Exam 2
  - In person during scheduled lecture on Wednesday, April 9 and Thursday, April 10th
- **Spring Break**
  - **No class from Monday, March 17th until Thursday, March 20th**

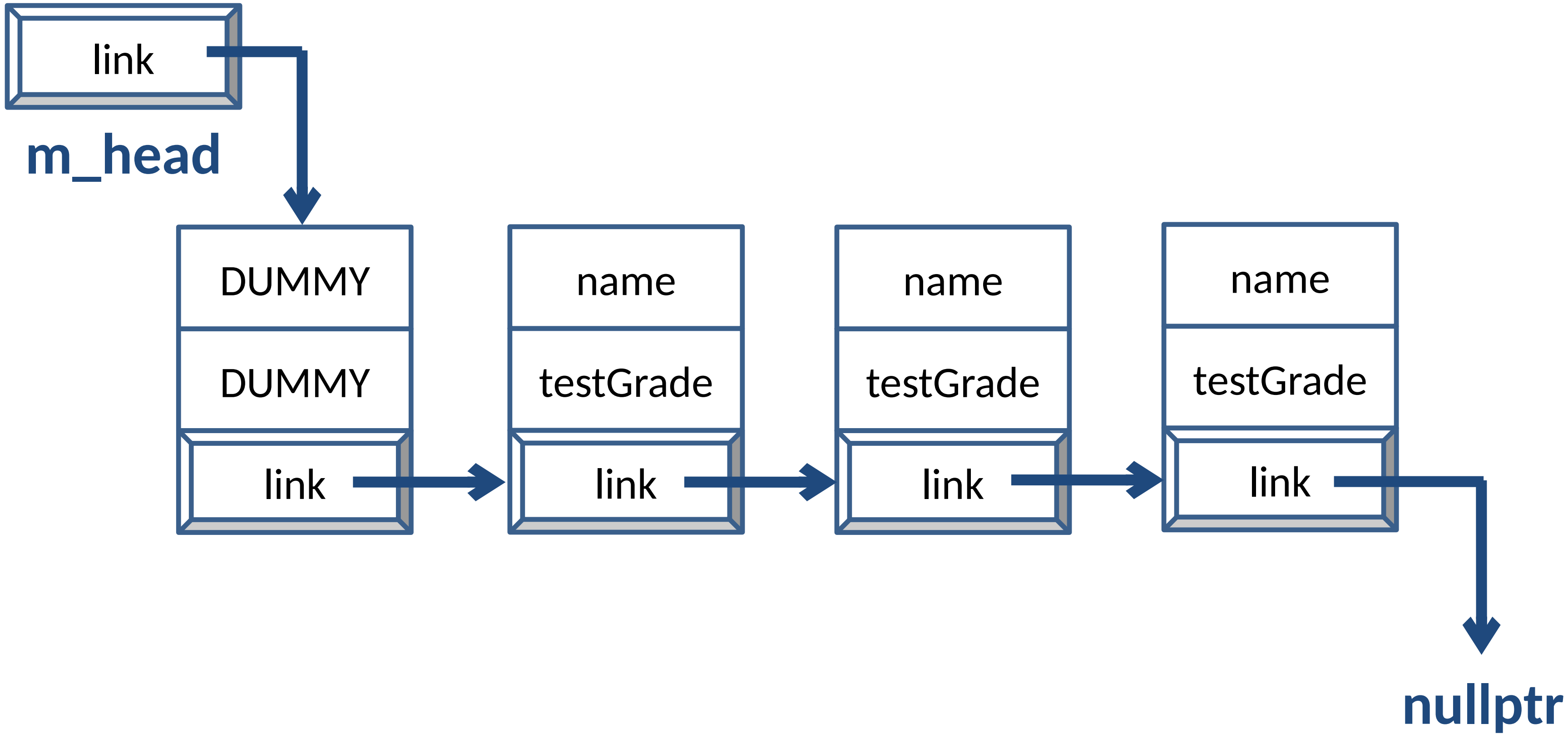
Next Time: Enums and Overloaded Operators

Using Dummy Nodes

Dummy Nodes

- When writing linked list functions (or algorithms in general), one has to consider the ”edge cases”.
- For linked lists, the edge cases are the first and last elements.
- These cases require special attention since m_head may point at nullptr and m_tail may point at nullptr. These can cause errors in your functions if you are not careful.
- To avoid such errors, it is common to define linked lists by using a “dummy” head node and a “dummy” tail node, instead of m_head and m_tail reference variables.
- The dummy nodes are objects of type Node just like the other nodes in the list. However, these nodes have a null element.
- Dummy nodes do not contribute to the size count, since the purpose of size is to indicate the number of elements in the list.

Example Linked List



Traversing a List with Dummy Nodes

Traversing the List

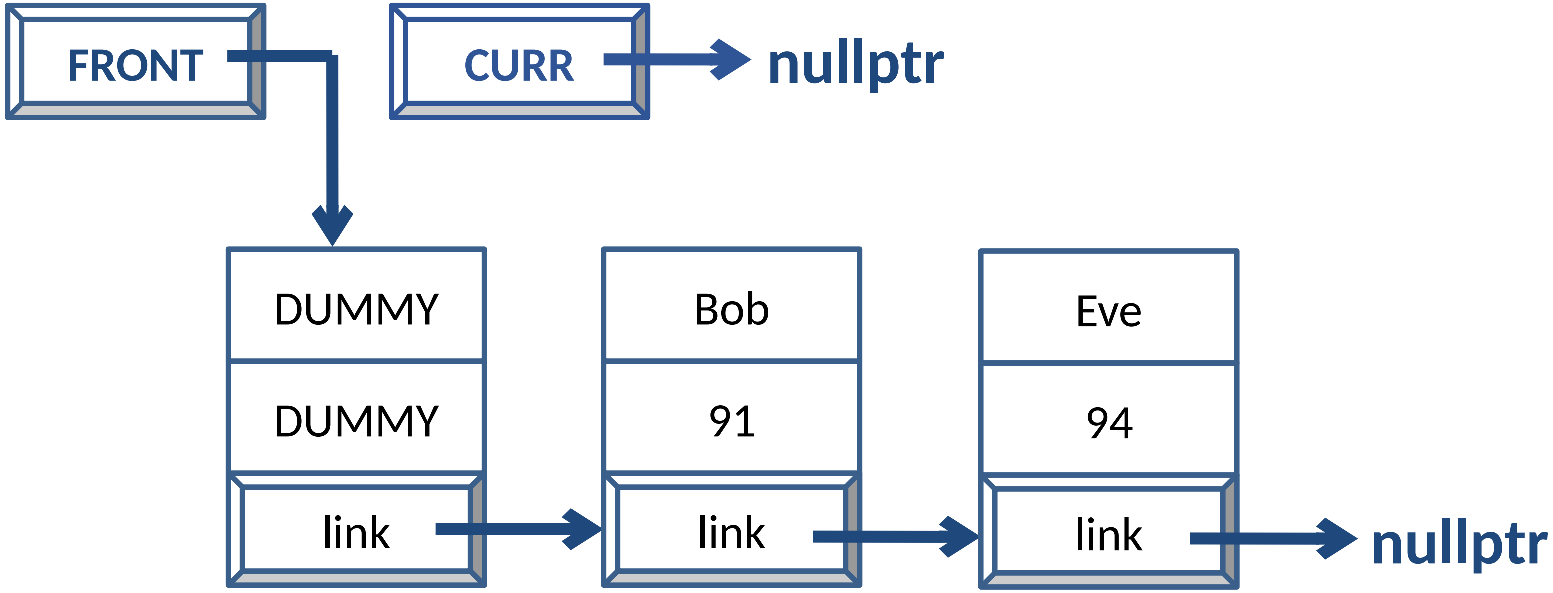
- To control our traversal, we'll use a loop
  - Initialization, Termination Condition, Modification

1. Set **`CURR`** to the first node in the list

2. Continue until we hit the end of the list (**`nullptr`**)

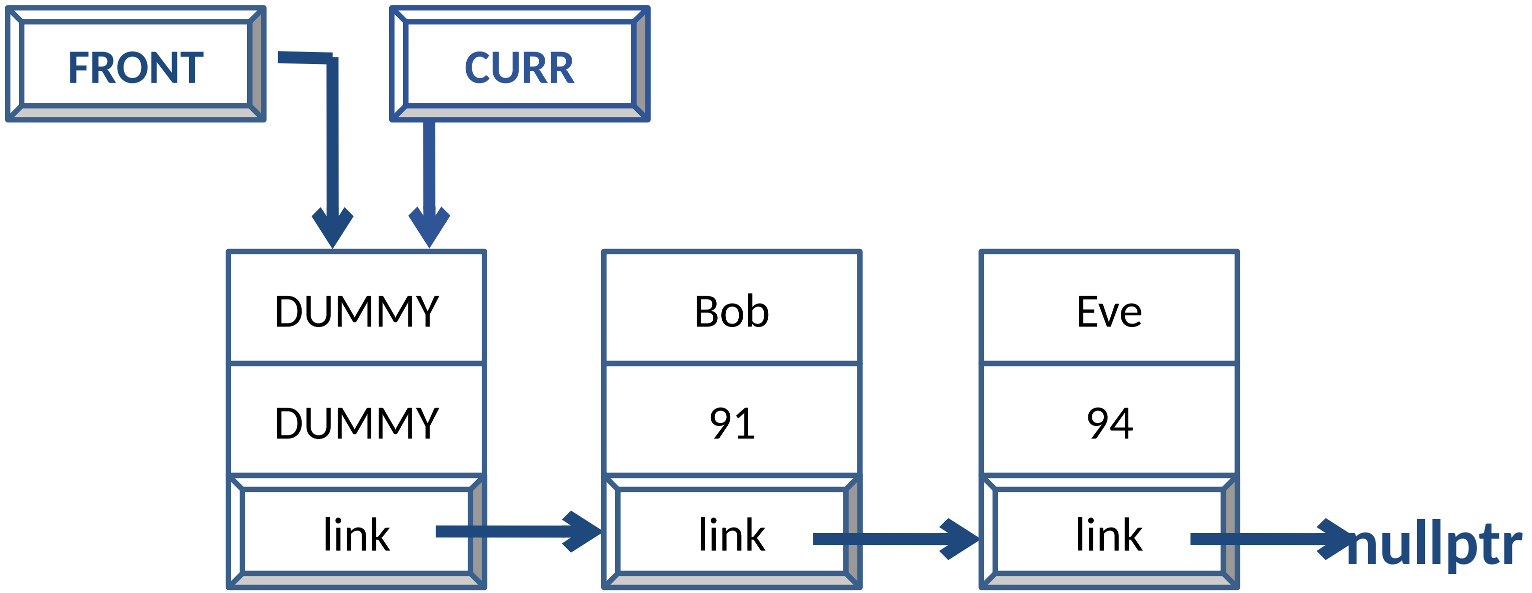
3. Move from one node to another
   (using **`m_next`**)

Demonstration of Traversal



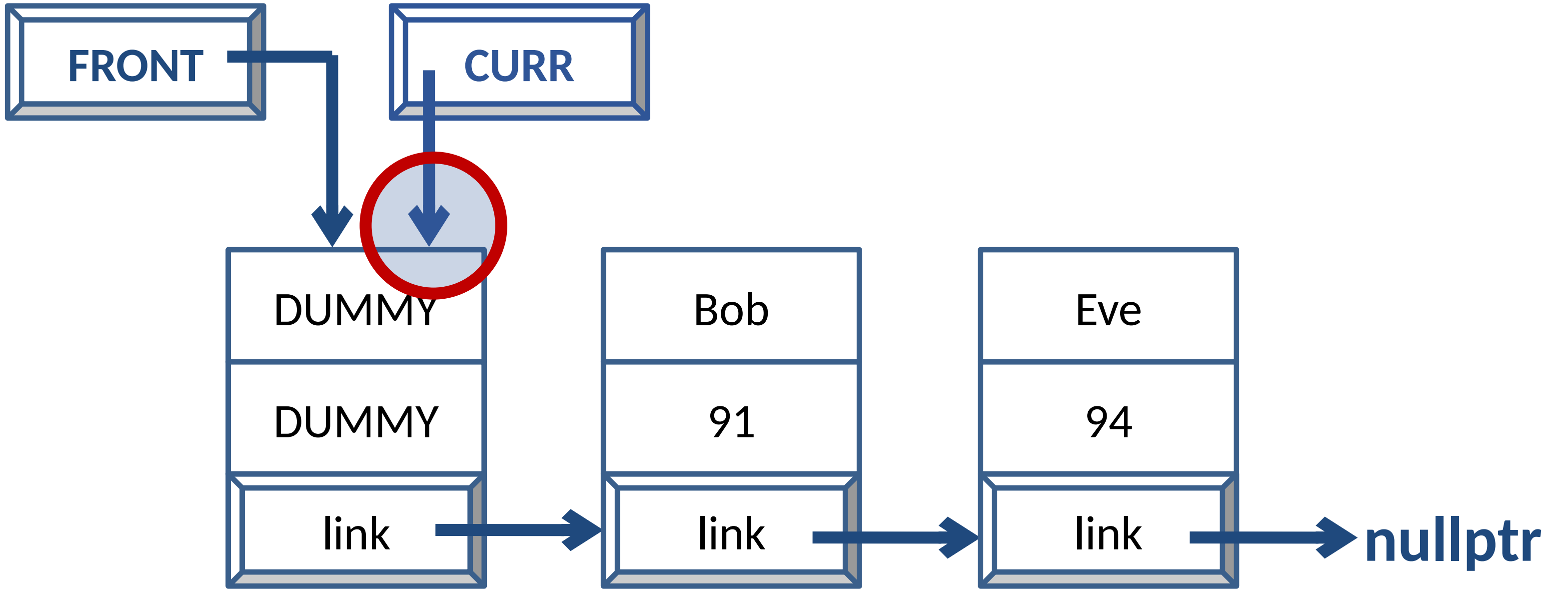

```
for (CURR = FRONT; CURR != nullptr; CURR = CURR->link) {
```

Demonstration of Traversal



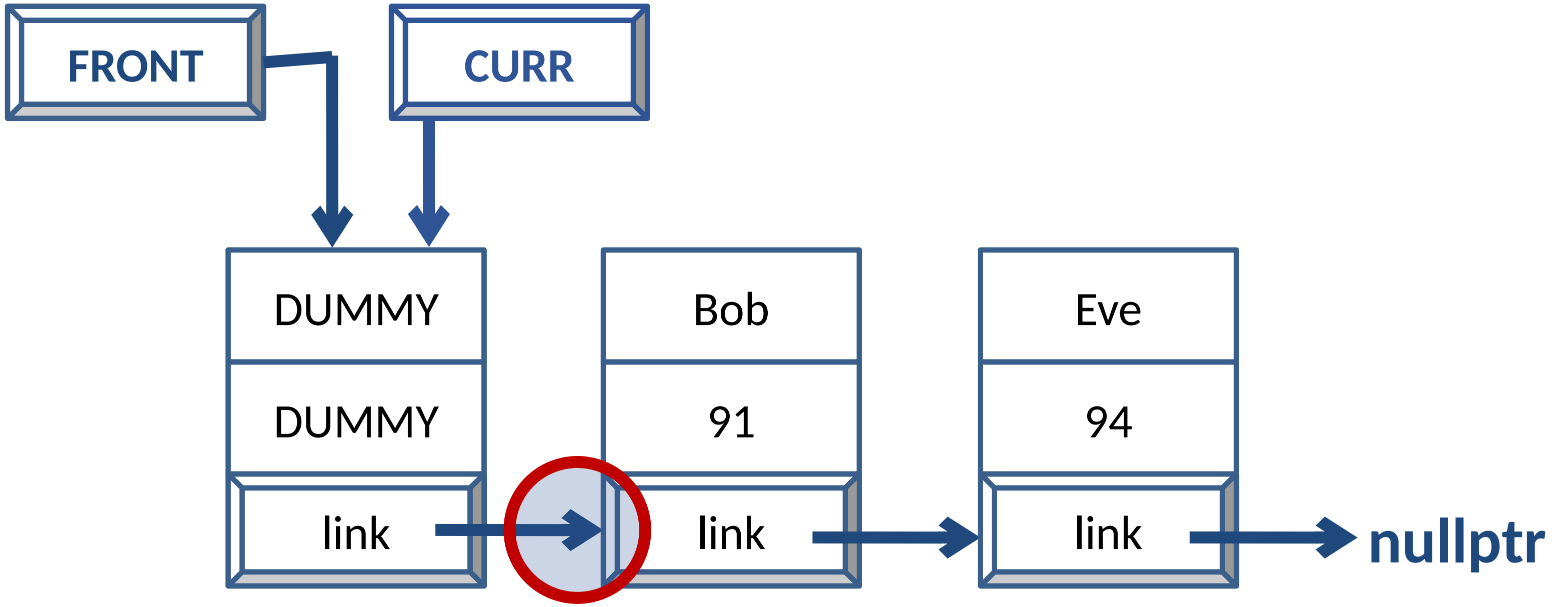

```
for (CURR = FRONT; CURR != nullptr; CURR = CURR->link) {
```

Demonstration of Traversal



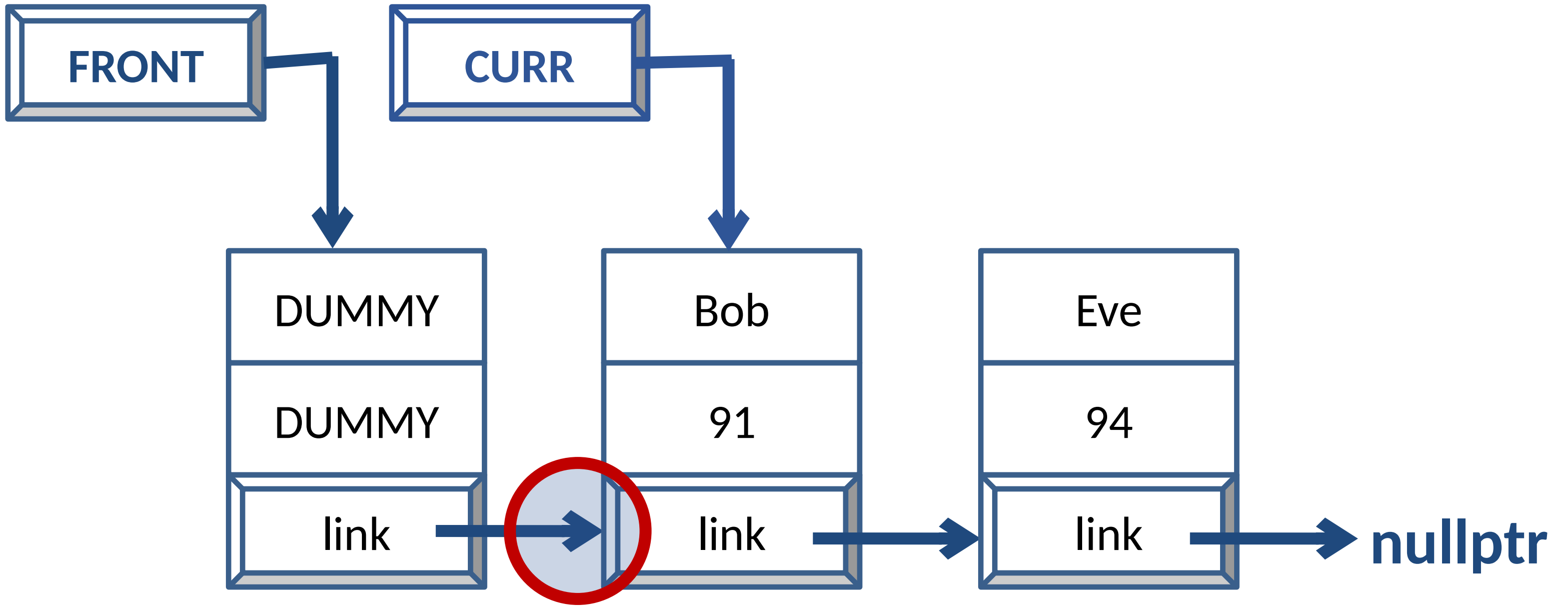

```
for (CURR = FRONT; CURR != nullptr; CURR = CURR->link) {
    // ignore, dummy node
```

Demonstration of Traversal



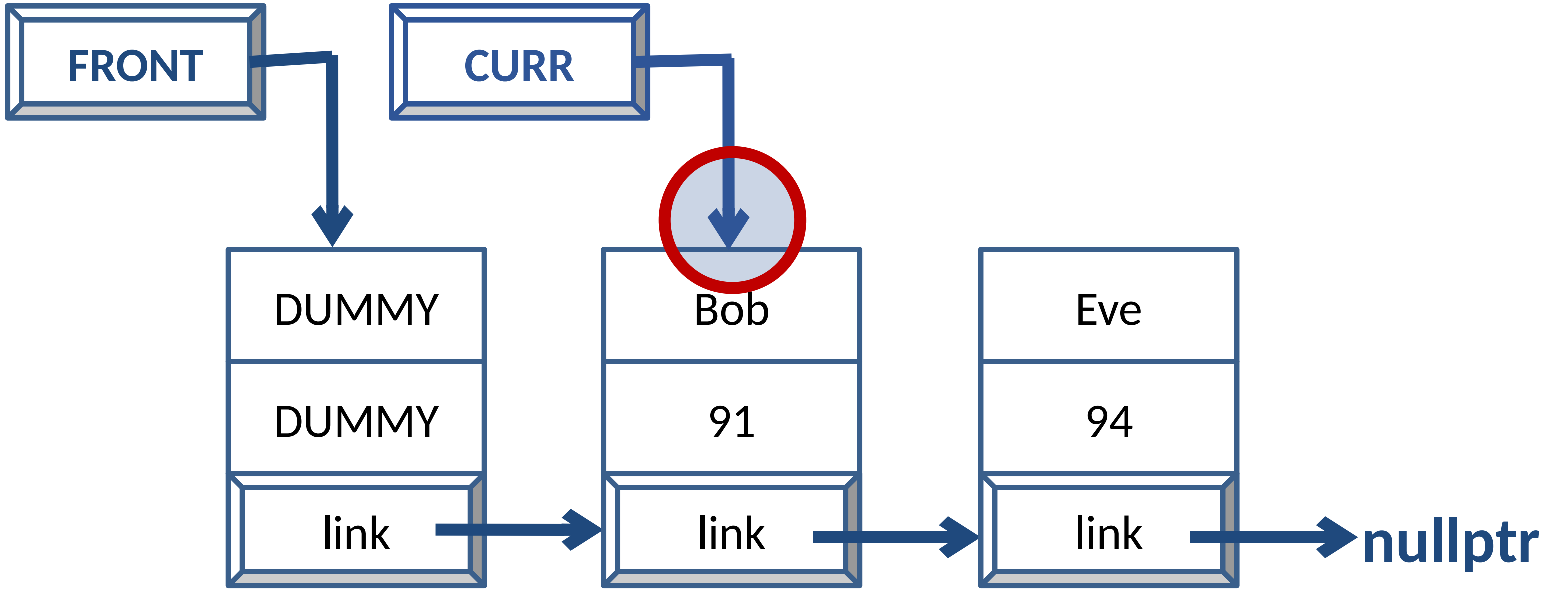

```
for (CURR = FRONT; CURR != nullptr; CURR = CURR->link) {
```

Demonstration of Traversal



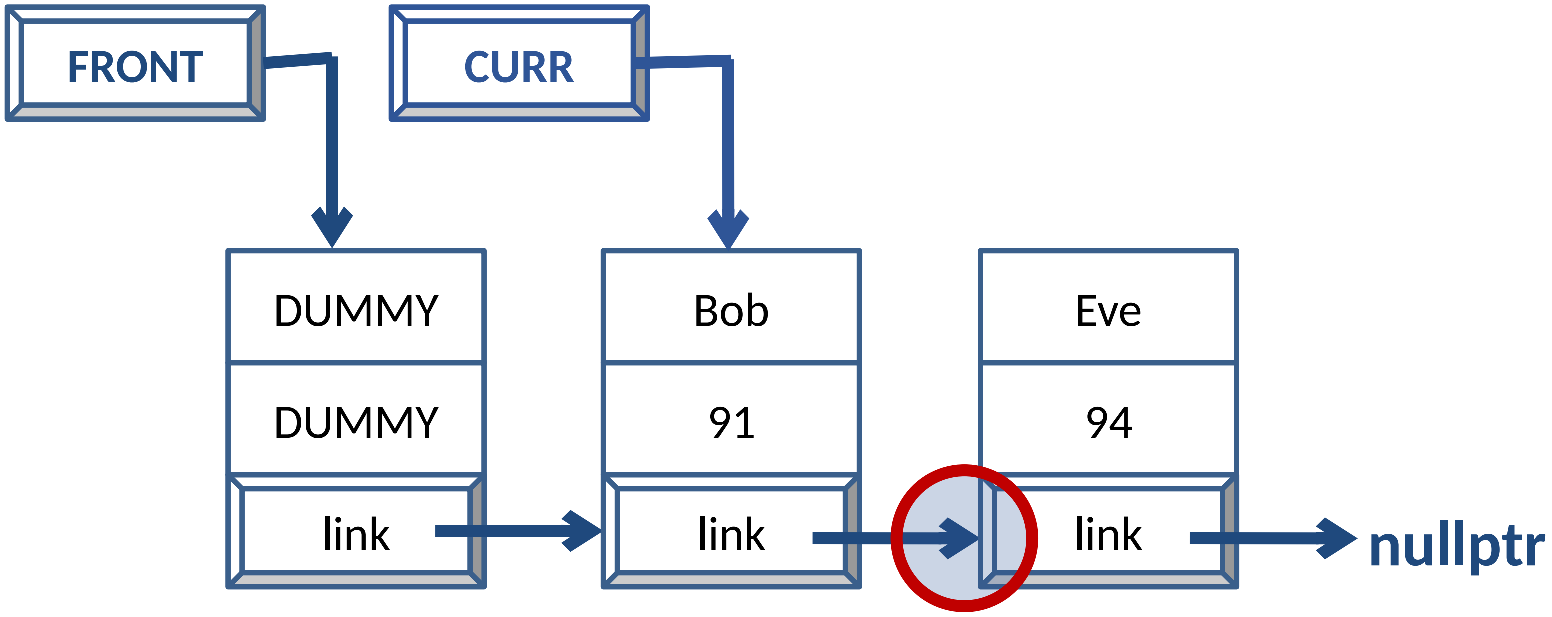

```
for (CURR = FRONT; CURR != nullptr; CURR = CURR->link) {
```

Demonstration of Traversal



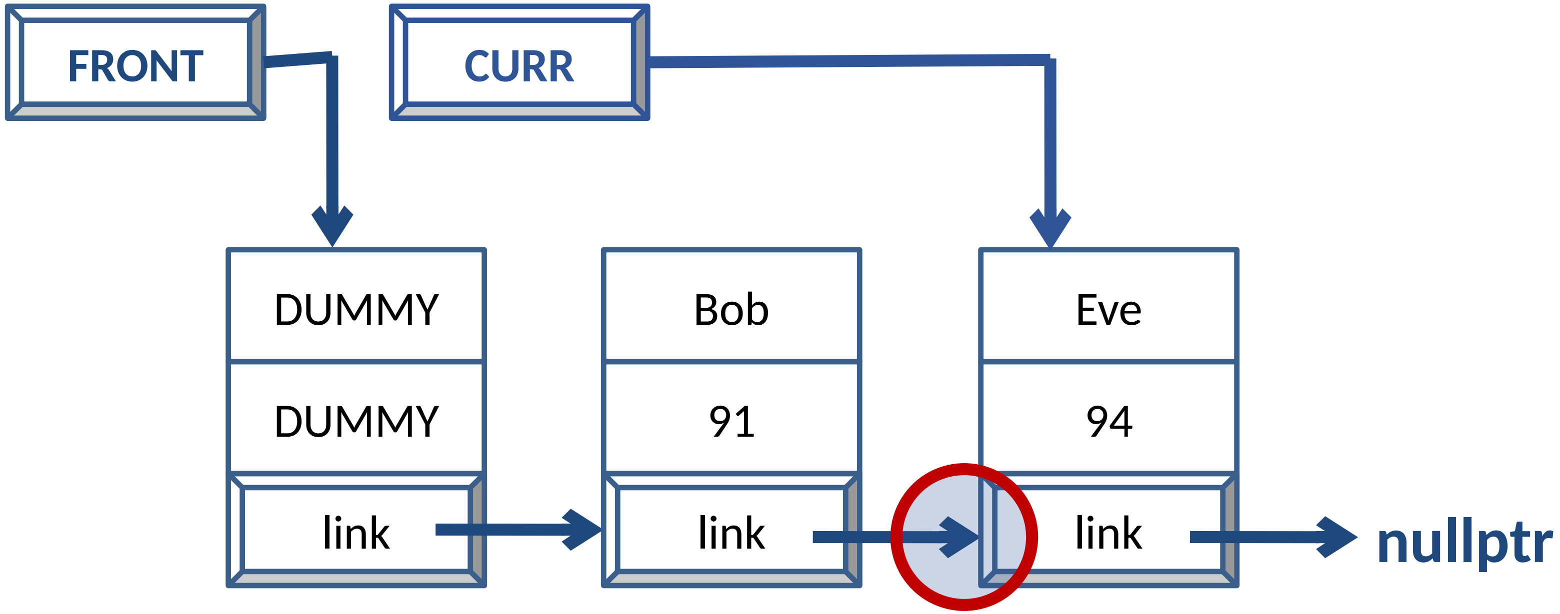

```
for (CURR = FRONT; CURR != nullptr; CURR = CURR->link) {
    // print information (Bob)
```

Demonstration of Traversal



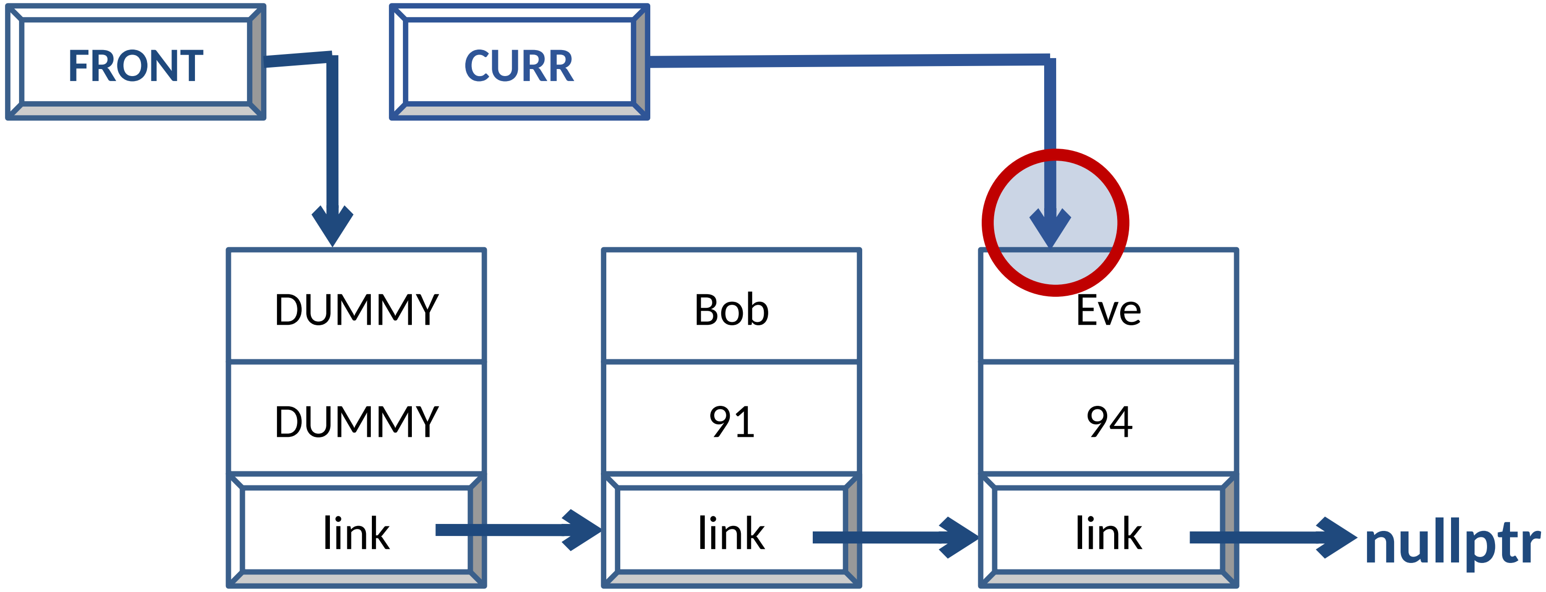

```
for (CURR = FRONT; CURR != nullptr; CURR = CURR->link) {
```

Demonstration of Traversal



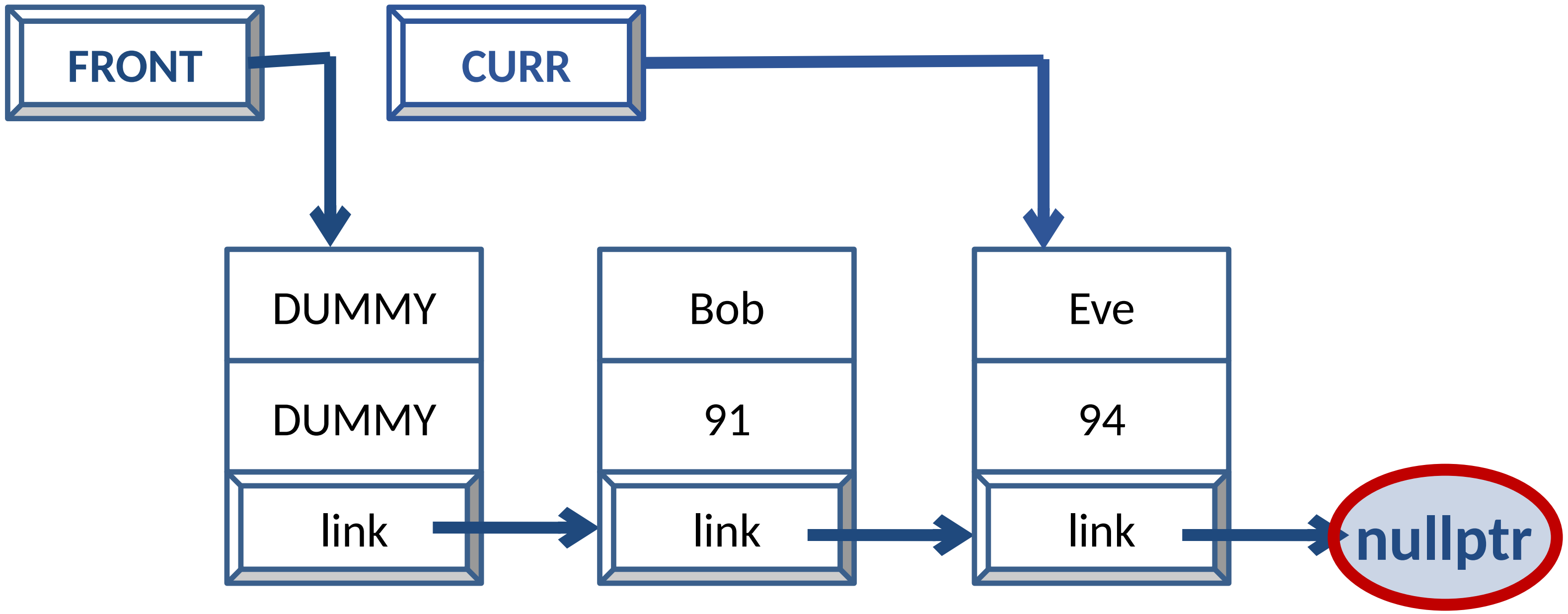

```
for (CURR = FRONT; CURR != nullptr; CURR = CURR->link) {
```

Demonstration of Traversal



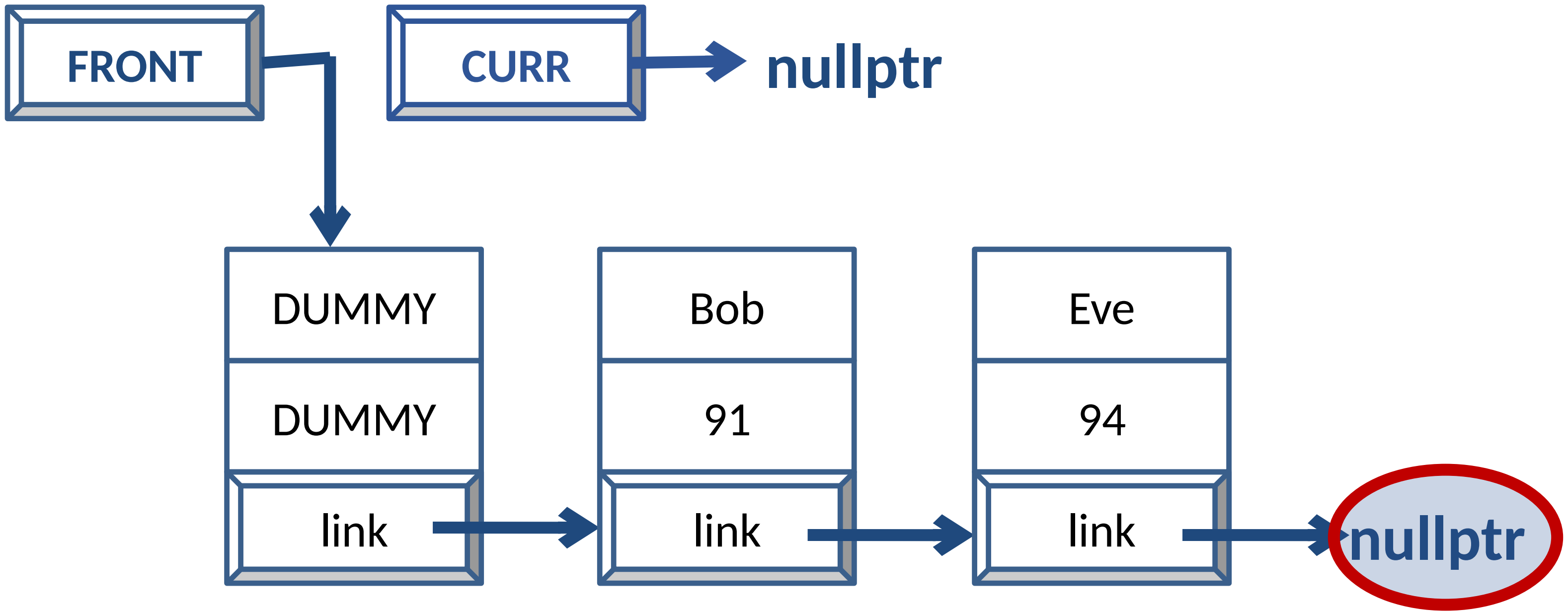

```
for (CURR = FRONT; CURR != nullptr; CURR = CURR->link) {
    // print information (Eve)
```

Demonstration of Traversal



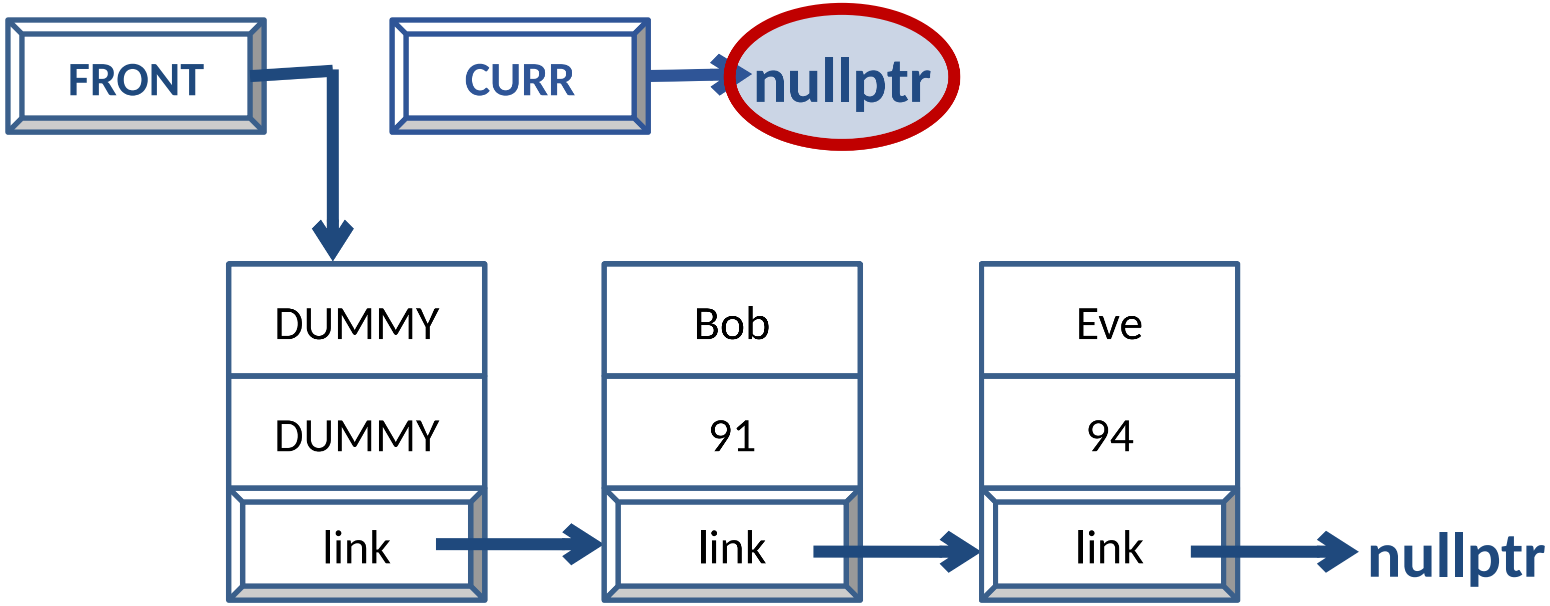

```
for (CURR = FRONT; CURR != nullptr; CURR = CURR->link) {
```

Demonstration of Traversal




```
for (CURR = FRONT; CURR != nullptr; CURR = CURR->link) {
```

Demonstration of Traversal




```
for (CURR = FRONT; CURR != nullptr; CURR = CURR->link) {

}  // exit the loop
```

Reverse Linked List

Reverse List

- If we wanted to **reverse** a linked list (in place), we will have to do some additional logic so that we do not lose our linked list.

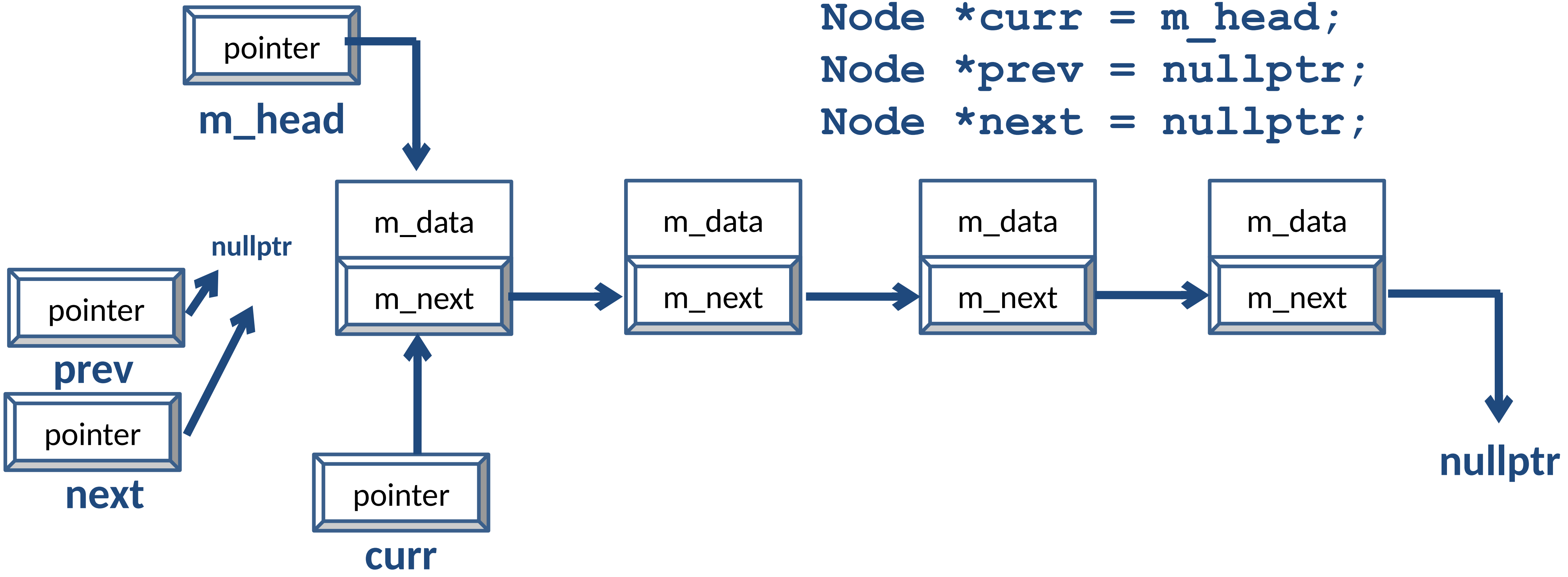
- Why?

- We need to keep track of all of the nodes and because each node only has one pointer, we need to update the pointers.

Reverse List

Step 1: Create Three Node Pointers

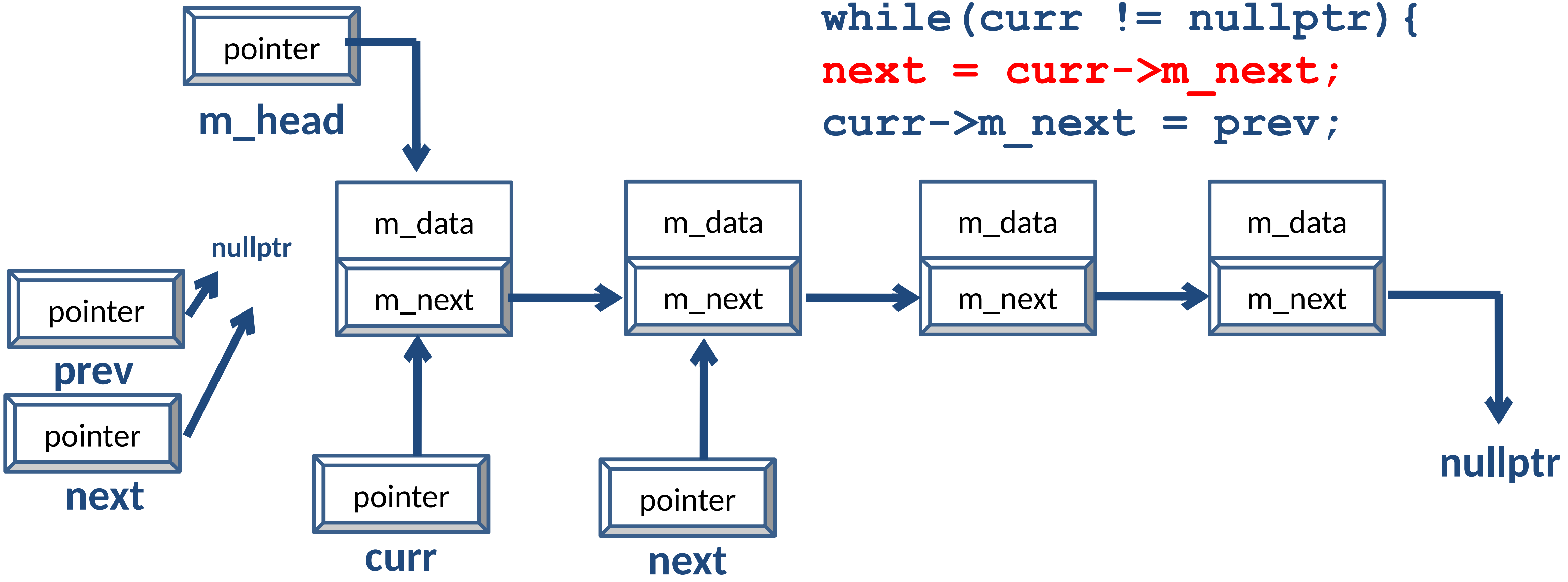
```
Node *curr = m_head;
Node *prev = nullptr;
Node *next = nullptr;
```



Reverse List

Step 2: Iterate and update pointers until end

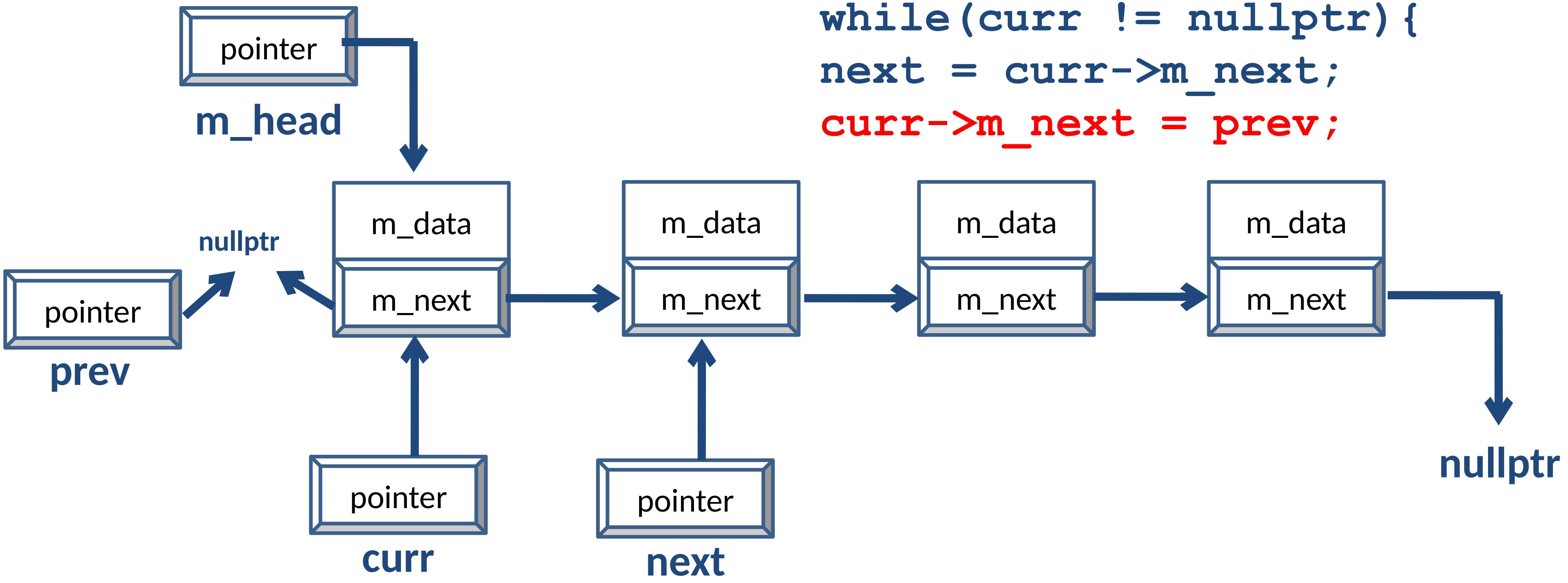
```
while(curr != nullptr){
next = curr->m_next;
curr->m_next = prev;
```



Reverse List

Step 2: Iterate and update pointers until end

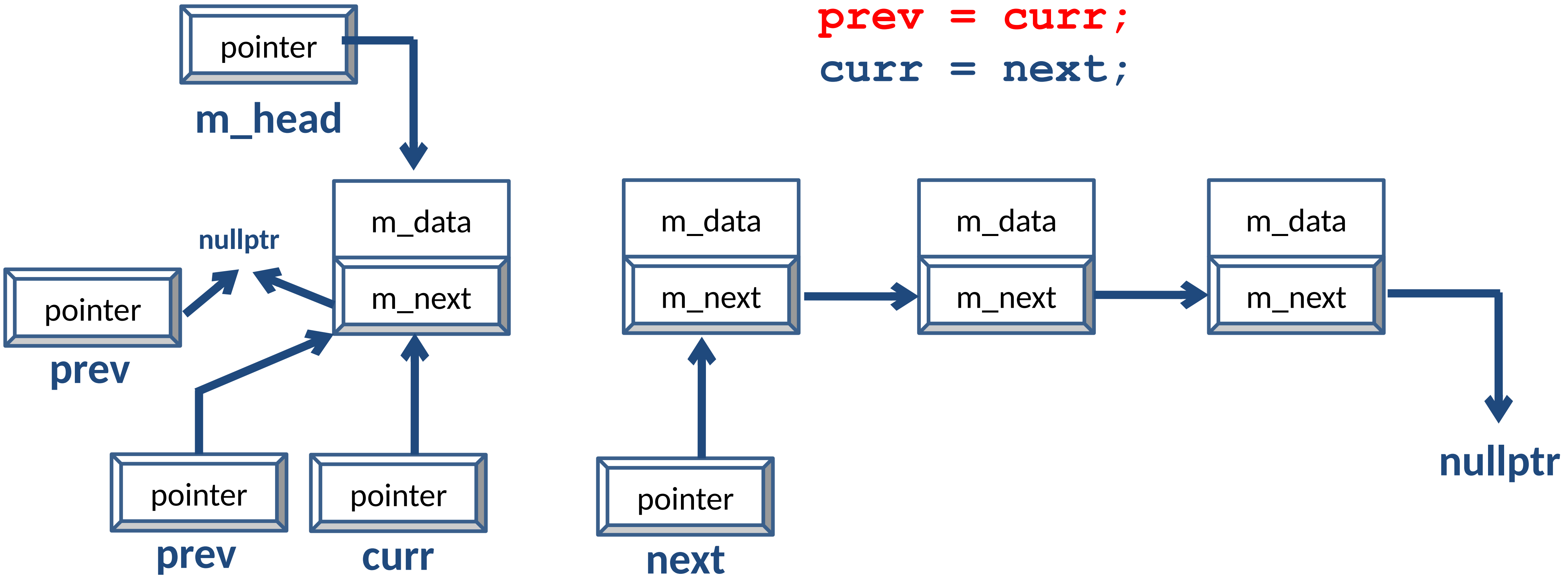
```
while(curr != nullptr){
next = curr->m_next;
curr->m_next = prev;
```



Reverse List

Step 3: Move pointers

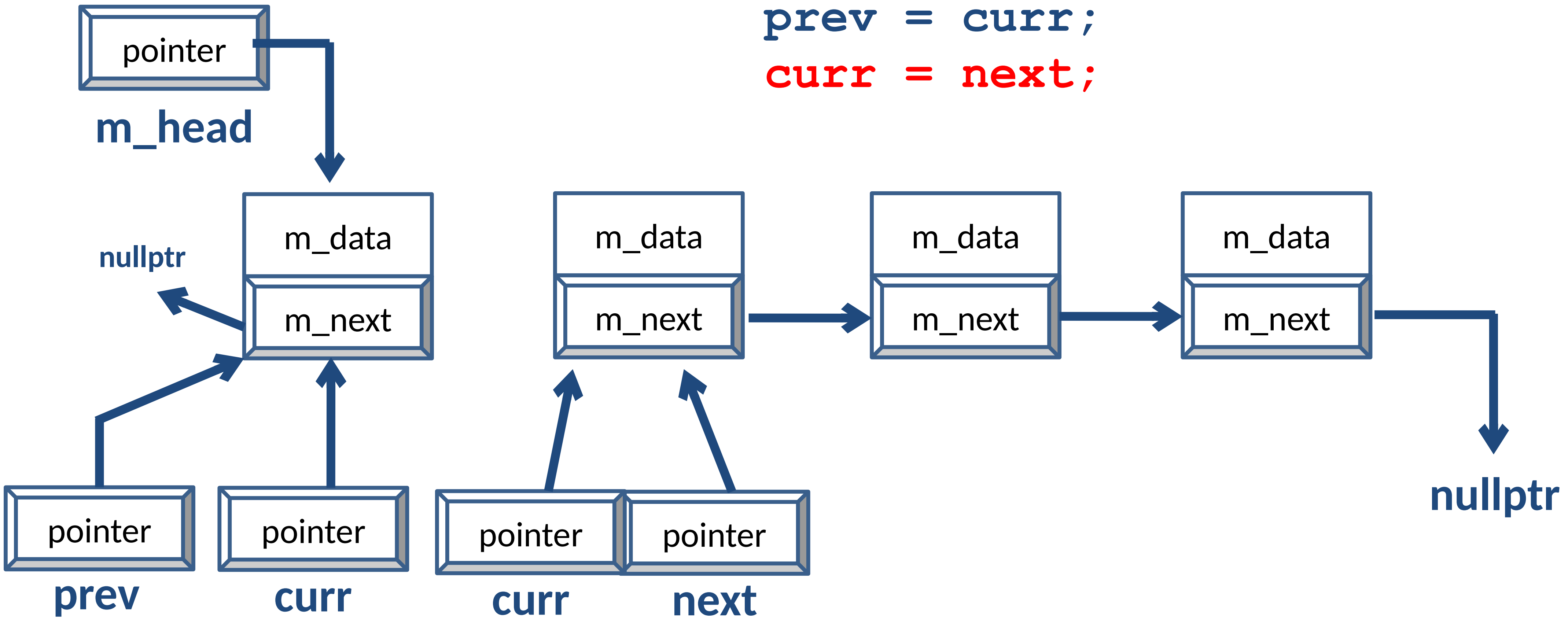
```
prev = curr;
curr = next;
```



Reverse List

Step 3: Move pointers

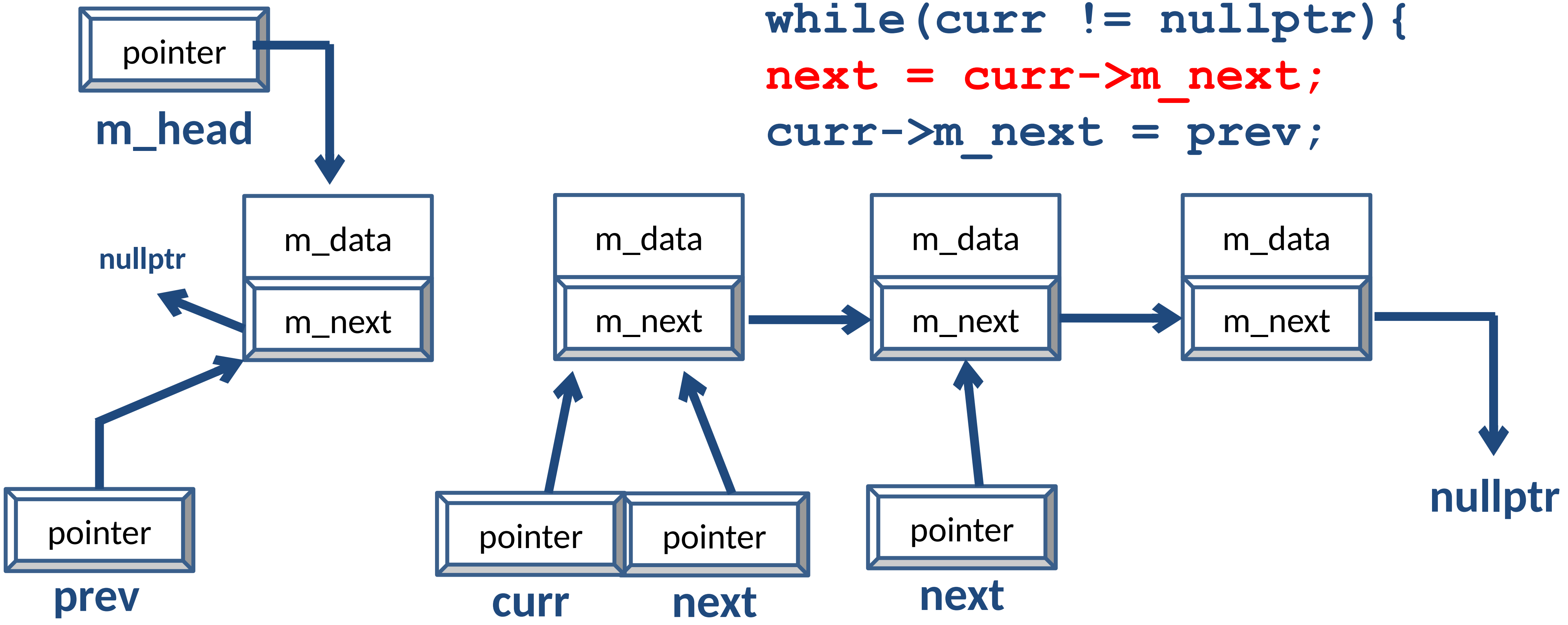
```
prev = curr;
curr = next;
```



Reverse List

Step 4: Iterate and update pointers until end

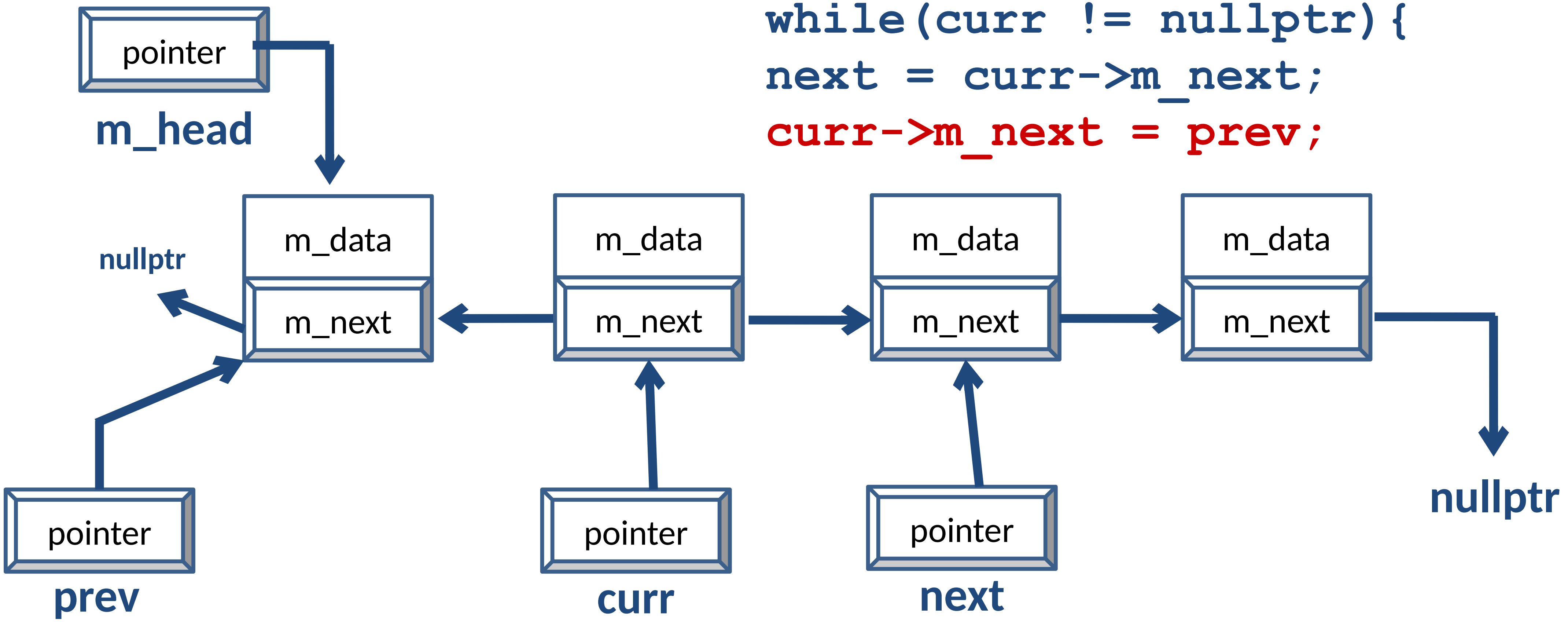
```
while(curr != nullptr){
next = curr->m_next;
curr->m_next = prev;
```



Reverse List

Step 4: Iterate and update pointers until end

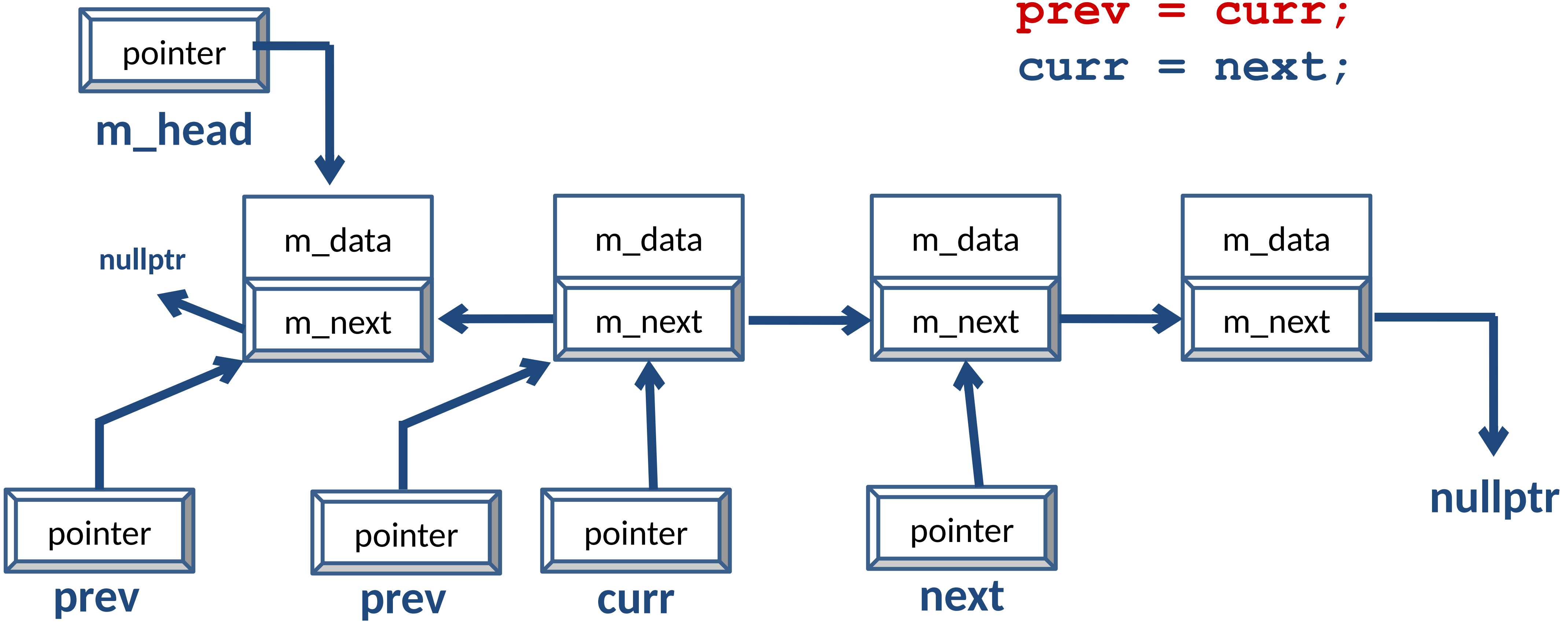
```
while(curr != nullptr){
next = curr->m_next;
curr->m_next = prev;
```



Reverse List

Step 5: Move Pointers

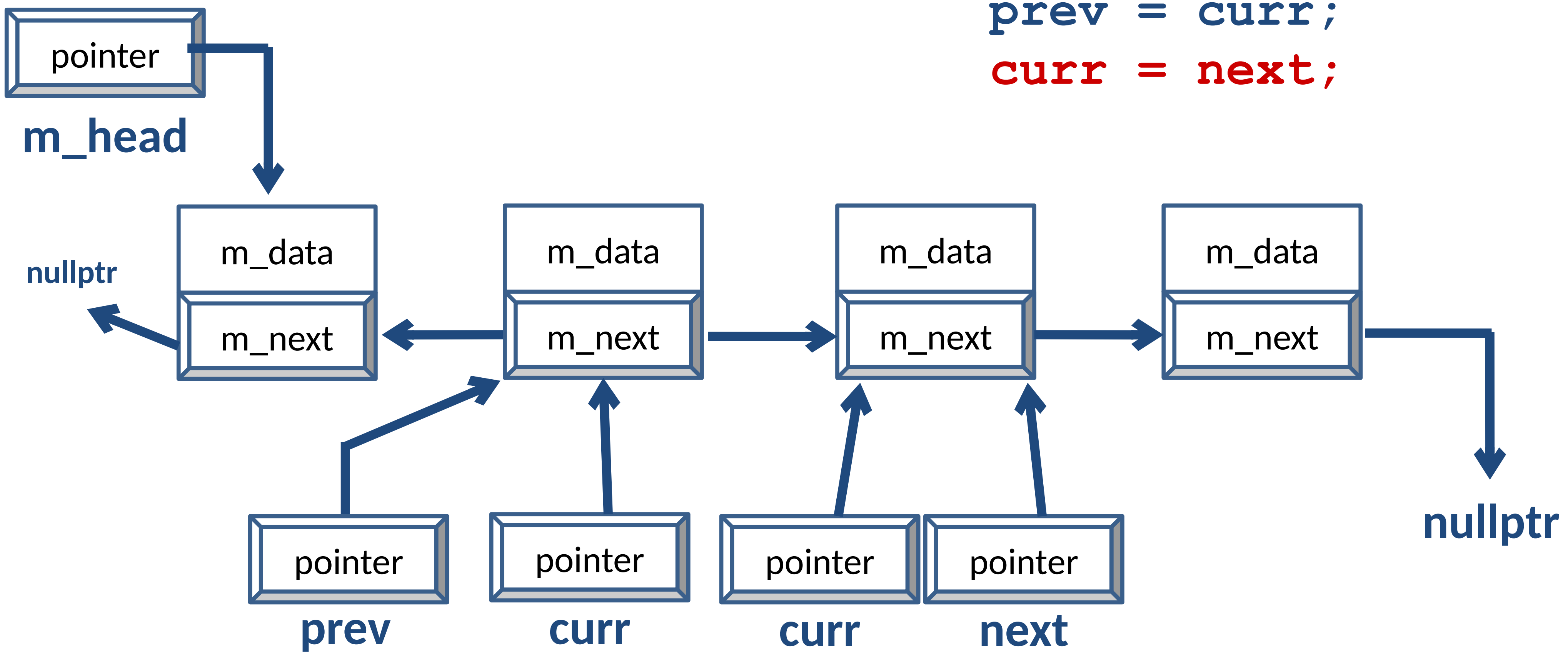
```
prev = curr;
curr = next;
```



Reverse List

Step 5: Move Pointers

```
prev = curr;
curr = next;
```



Reverse List