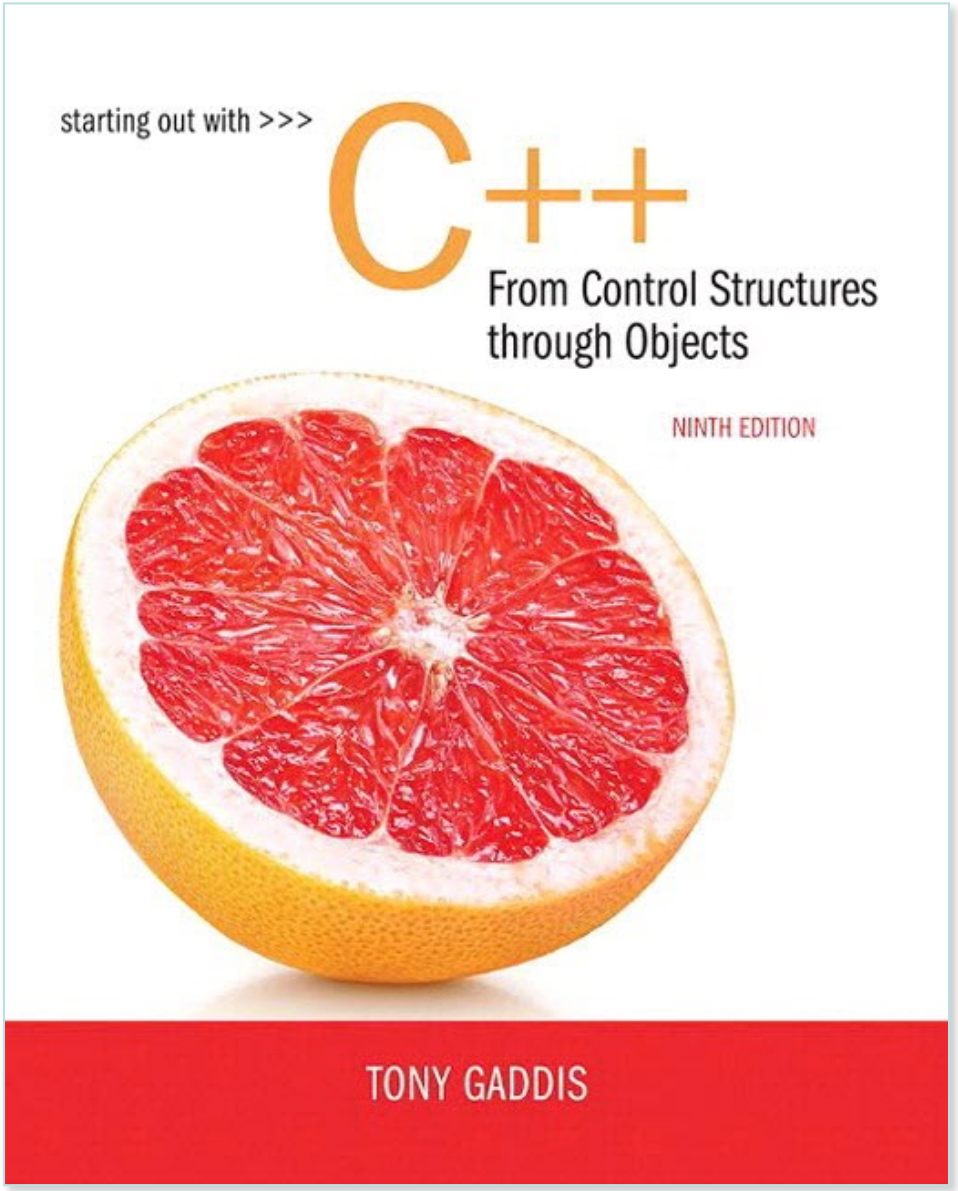
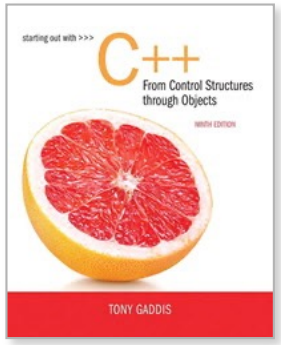
Linked Lists

(no ADT version)

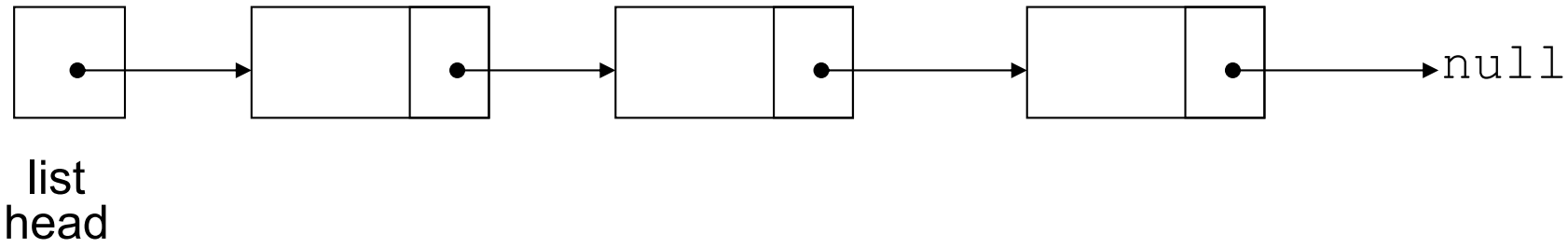




Introduction to the Linked List

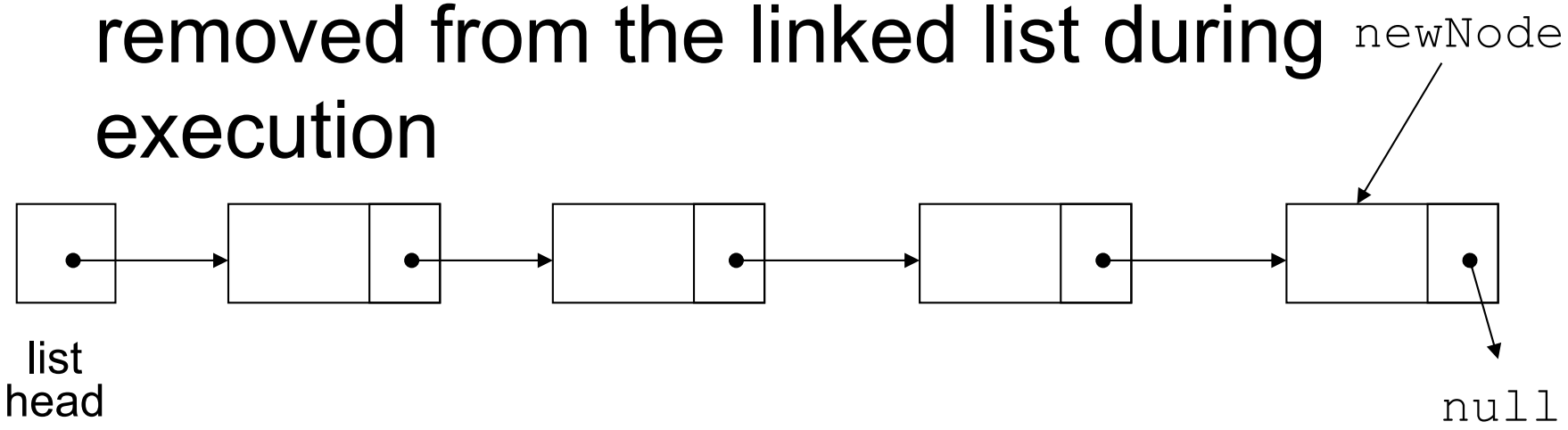
Introduction to the Linked List ADT

- Linked list: set of data structures (nodes) that contain references to other data structures



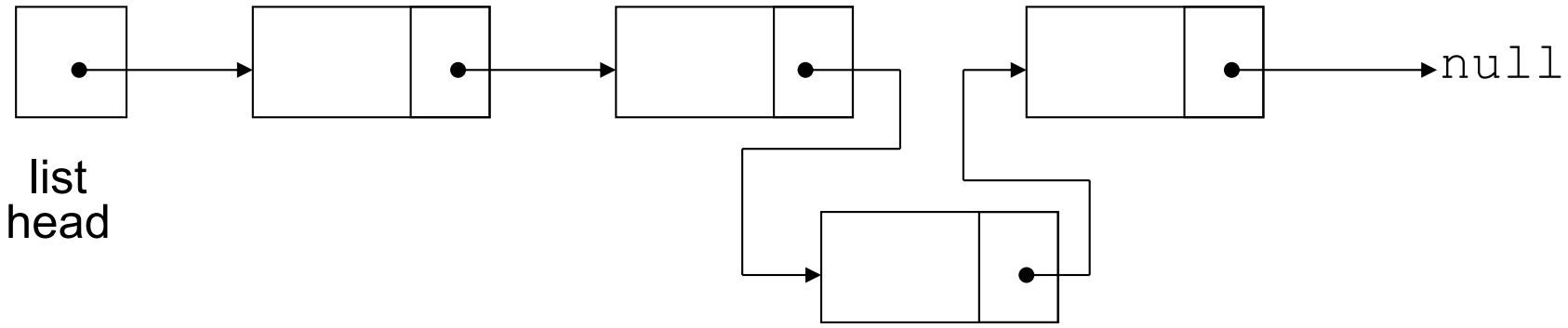
Introduction to the Linked List ADT

- References may be addresses or array indices
- Data structures can be added to or removed from the linked list during execution



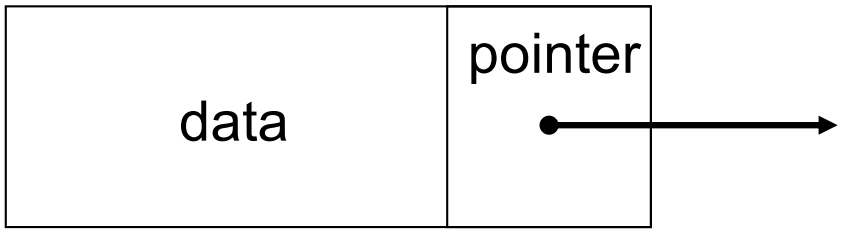
Linked Lists vs. Arrays and Vectors

- Linked lists can grow and shrink as needed, unlike arrays, which have a fixed size
- Linked lists can insert a node between other nodes easily



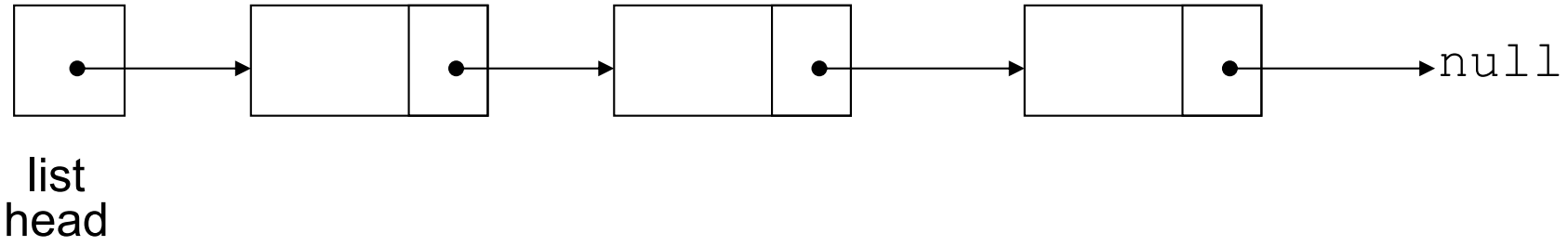
Node Organization

- A node contains:
  - data: one or more data fields – may be organized as structure, object, etc.
  - a pointer that can point to another node



Linked List Organization

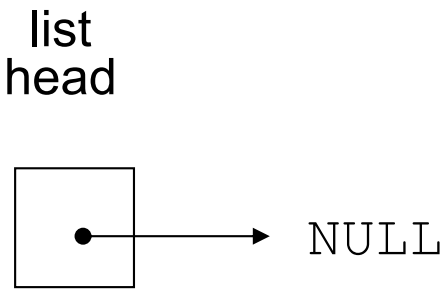
- Linked list contains 0 or more nodes:



- Has a list head to point to first node
- Last node points to `null` (address 0)

Empty List

- If a list currently contains 0 nodes, it is the empty list
- In this case the list head points to `null`



Declaring a Node

- Declare a node:

```
struct ListNode
{
      int data;
      ListNode *next;
};
```

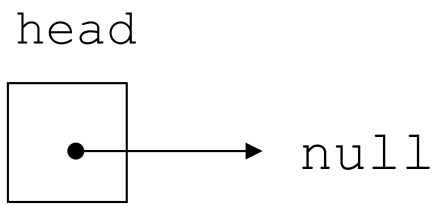
- No memory is allocated at this time

Defining a Linked List

- Define a pointer for the head of the list:

```
ListNode *head = nullptr;
```

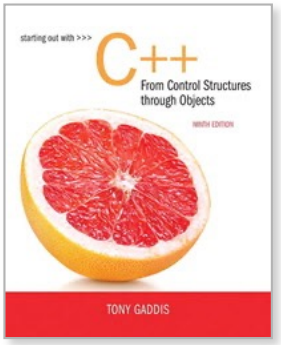
- Head pointer initialized to `nullptr` to indicate an empty list



The Null Pointer

- Is used to indicate end-of-list
- Should always be tested for before using a pointer:

```
ListNode *p;
while (!p)
```



18.2

Linked List Operations

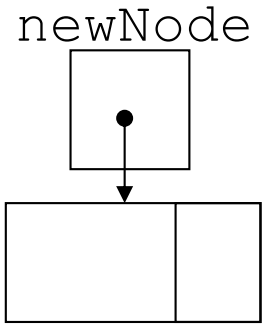
Linked List Operations

- Basic operations:
  - append a node to the end of the list
  - insert a node within the list
  - traverse the linked list
  - delete a node
  - delete/destroy the list

Create a New Node

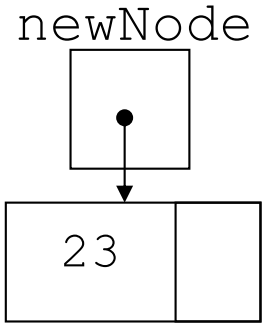
- Allocate memory for the new node:

```
newNode = new ListNode;
```



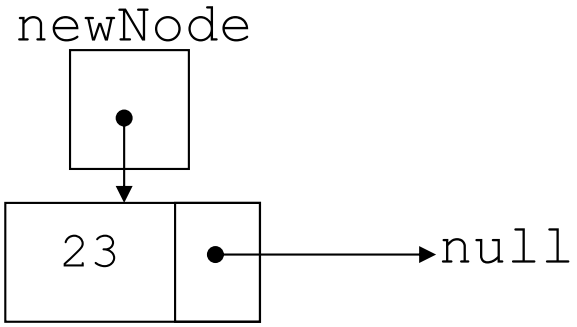
- Initialize the contents of the node:

```
newNode->value = num;
```



- Set the pointer field to `nullptr`:

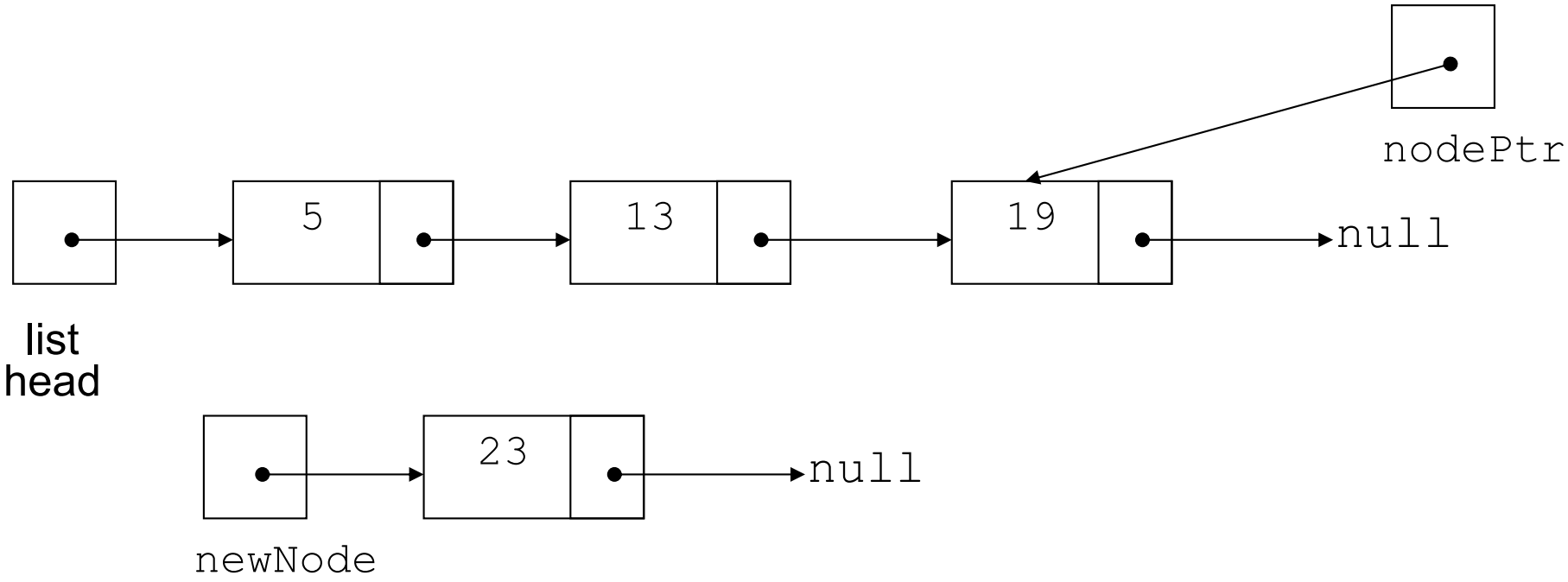
```
newNode->next = nullptr;
```



Appending a Node

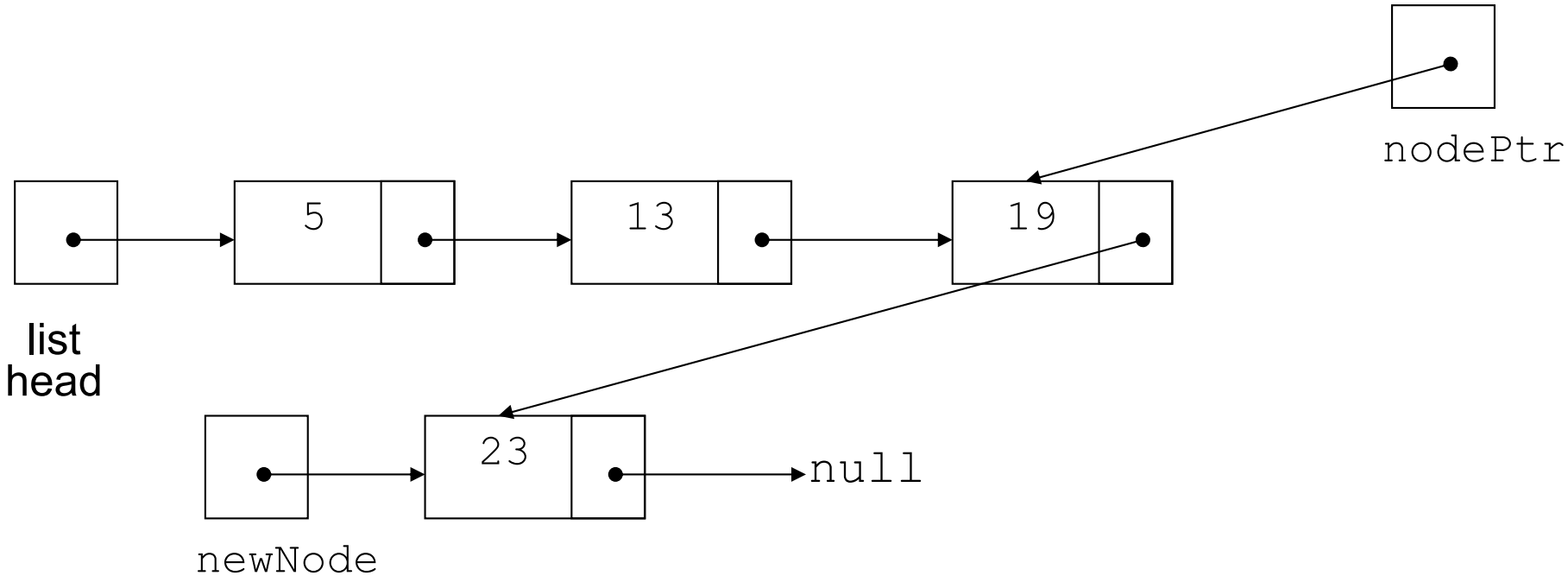
- Add a node to the end of the list
- Basic process:
  - Create the new node (as already described)
  - Add node to the end of the list:
    - If list is empty, set head pointer to this node
    - Else,
      - traverse the list to the end
      - set pointer of last node to point to new node

Appending a Node



New node created, end of list located

Appending a Node



New node added to end of list

C++ code for Appending a Node

```
11  void appendNode(ListNode * head, double num)
12  {
13     ListNode *newNode;  // To point to a new node
14     ListNode *nodePtr;  // To move through the list
15
16     // Allocate a new node and store num there.
17     newNode = new ListNode;
18     newNode->value = num;
19     newNode->next = nullptr;
20
21     // If there are no nodes in the list
22     // make newNode the first node.
23     if (!head)
```

C++ code for Appending a Node (Continued)

```
24          head = newNode;
25       else  // Otherwise, insert newNode at end.
26       {
27          // Initialize nodePtr to head of list.
28          nodePtr = head;
29
30          // Find the last node in the list.
31          while (nodePtr->next)
32             nodePtr = nodePtr->next;
33
34          // Insert newNode as the last node.
35          nodePtr->next = newNode;
36       }
37    }
```

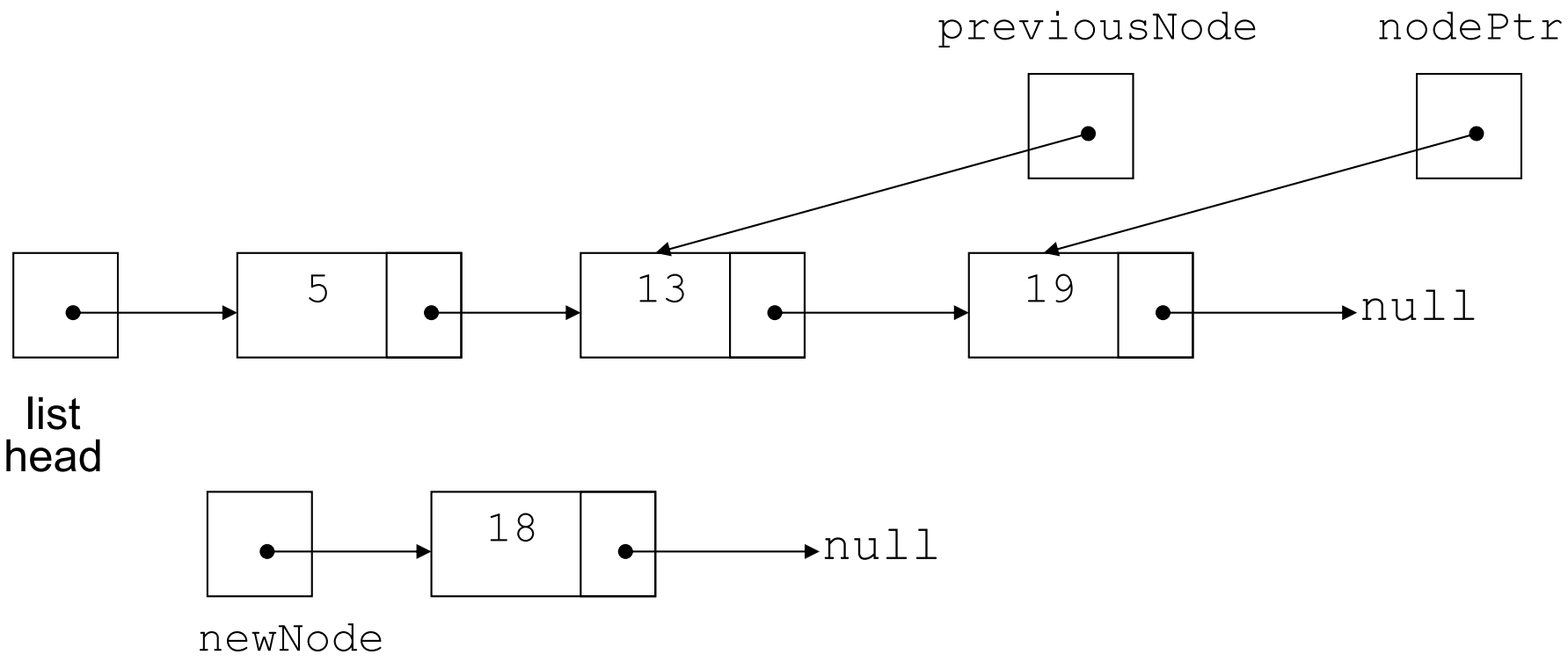
The main function:

```
Int main()
{
        ListNode * head=nullptr;
        appendNode(head, 4.5);
        appendNode(head, 8.2);
        appendNode(head, 6.7);

        return 0;
}
```

Inserting a Node into a Linked List

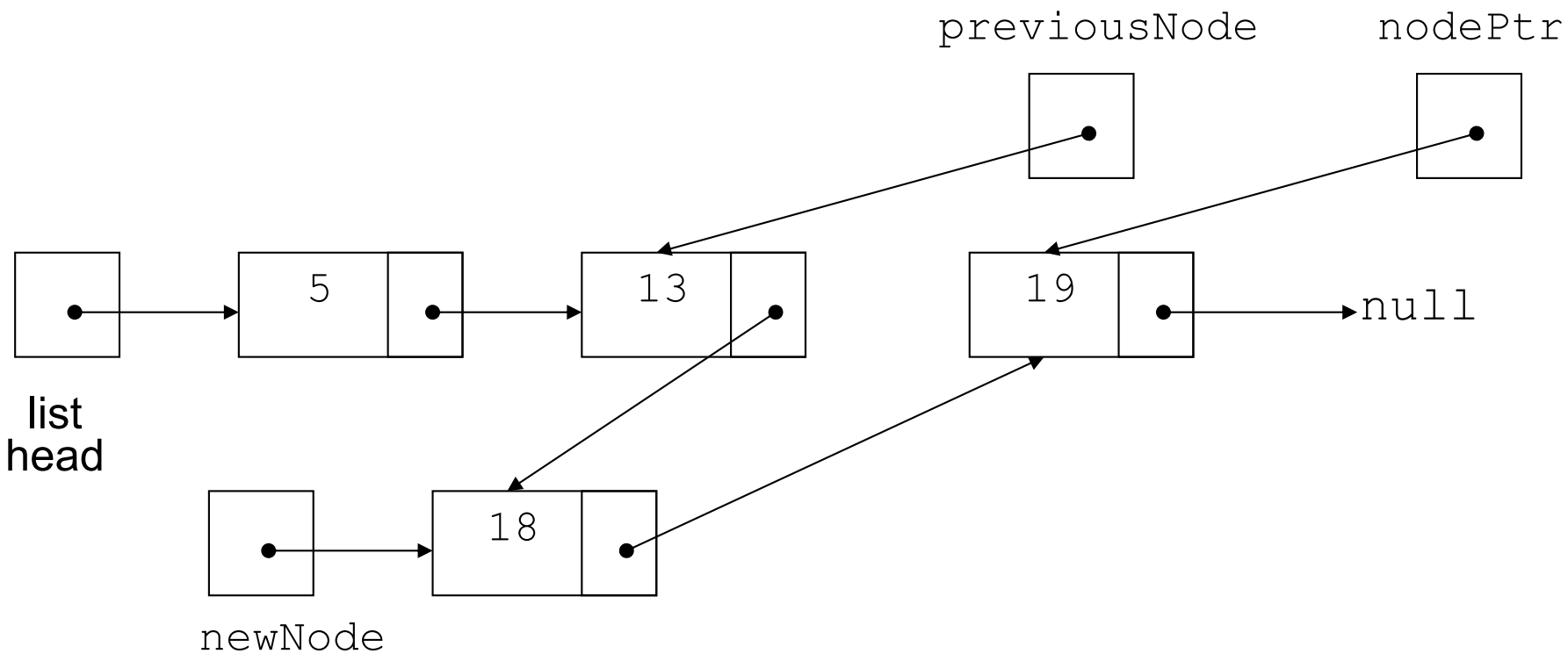
- Used to maintain a linked list in order
- Requires two pointers to traverse the list:
  - pointer to locate the node with data value greater than that of node to be inserted
  - pointer to 'trail behind' one node, to point to node before point of insertion
- New node is inserted between the nodes pointed at by these pointers

Inserting a Node into a Linked List



New node created, correct position located

Inserting a Node into a Linked List



New node inserted in order in the linked list

```
Void insertNode(ListNode &head, double num)
70  {
71      ListNode *newNode;                // A new node
72      ListNode *nodePtr;                // To traverse the list
73      ListNode *previousNode = nullptr; // The previous node
74
75      // Allocate a new node and store num there.
76      newNode = new ListNode;
77      newNode->value = num;
78
79      // If there are no nodes in the list
80      // make newNode the first node
81      if (!head)
82      {
83          head = newNode;
84          newNode->next = nullptr;
85      }
86      else // Otherwise, insert newNode
87      {
88          // Position nodePtr at the head of list.
89          nodePtr = head;
90
```

```
91          // Initialize previousNode to nullptr.
92          previousNode = nullptr;
93
94          // Skip all nodes whose value is less than num.
95          while (nodePtr != nullptr && nodePtr->value < num)
96          {
97               previousNode = nodePtr;
98               nodePtr = nodePtr->next;
99          }
100
101         // If the new node is to be the 1st in the list,
102         // insert it before all other nodes.
103         if (previousNode == nullptr)
104         {
105              head = newNode;
106              newNode->next = nodePtr;
107         }
108         else // Otherwise insert after the previous node.
109         {
110              previousNode->next = newNode;
111              newNode->next = nodePtr;
112         }
113     }
114 }
```

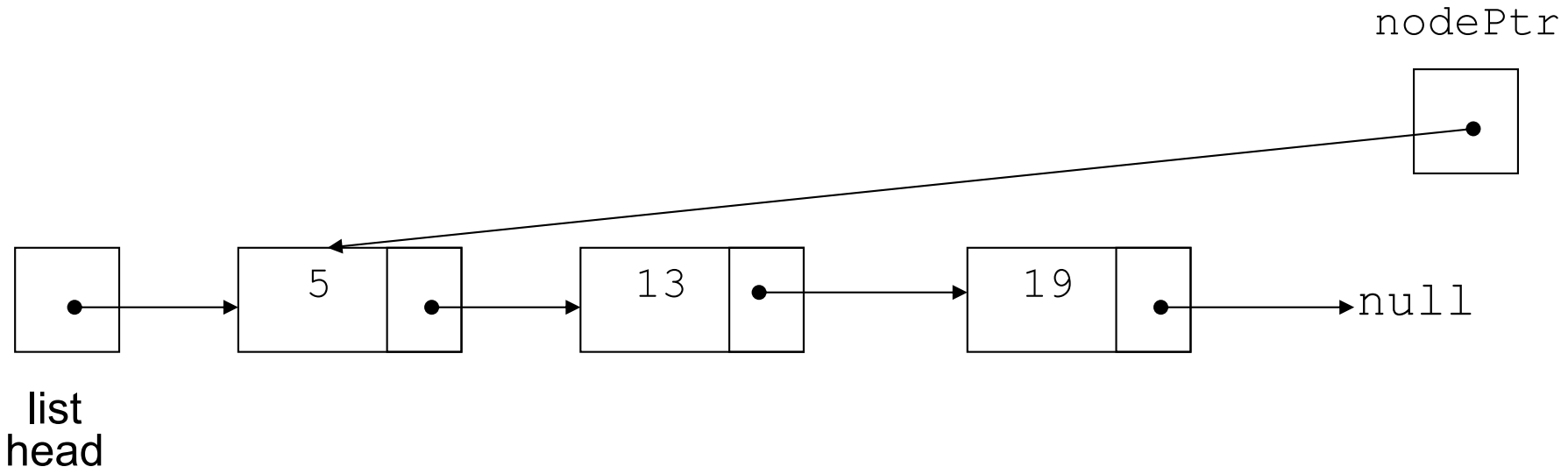
The main function:

```
Int main()
{
        ListNode * head=nullptr;

        insertNode(head, 4.5);
        insertNode(head, 8.2);
        insertNode(head, 6.7);

        return 0;
}
```

Traversing a Linked List

- Visit each node in a linked list: display contents, validate data, etc.
- Basic process:
  - set a pointer to the contents of the head pointer
  - while pointer is not a null pointer
    - process data
    - go to the next node by setting the pointer to the pointer field of the current node in the list
  - end while

Traversing a Linked List

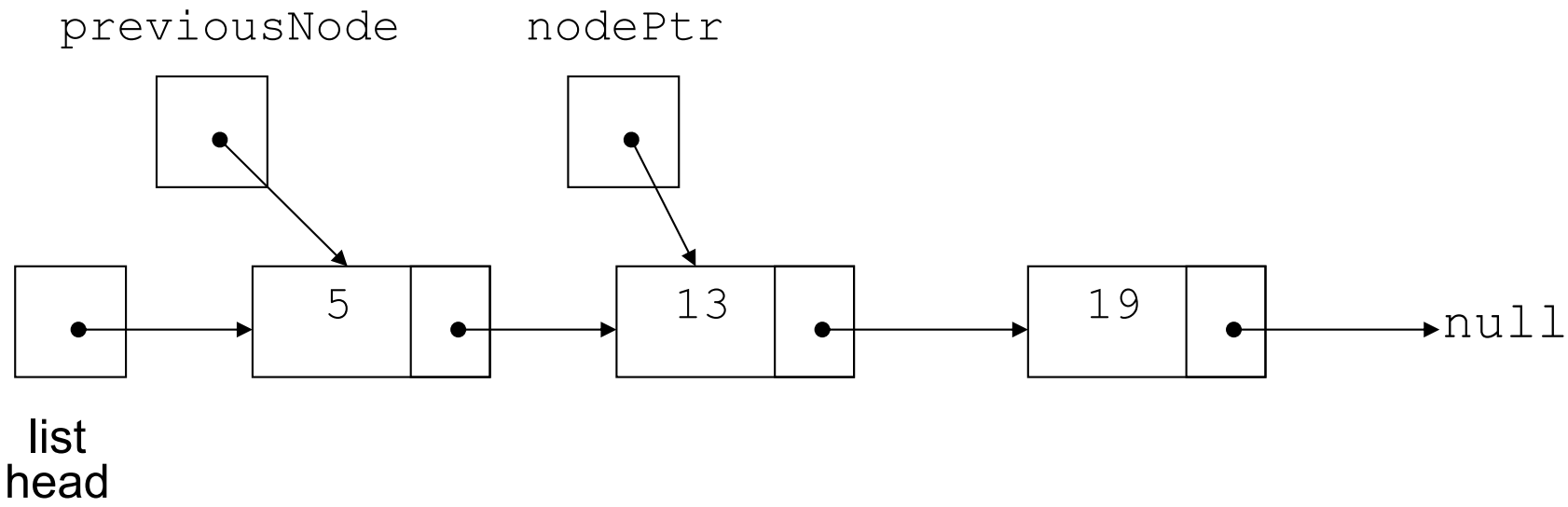


`nodePtr` points to the node containing `5`, then the node containing `13`, then the node containing `19`, then points to the null pointer, and the list traversal stops

Deleting a Node

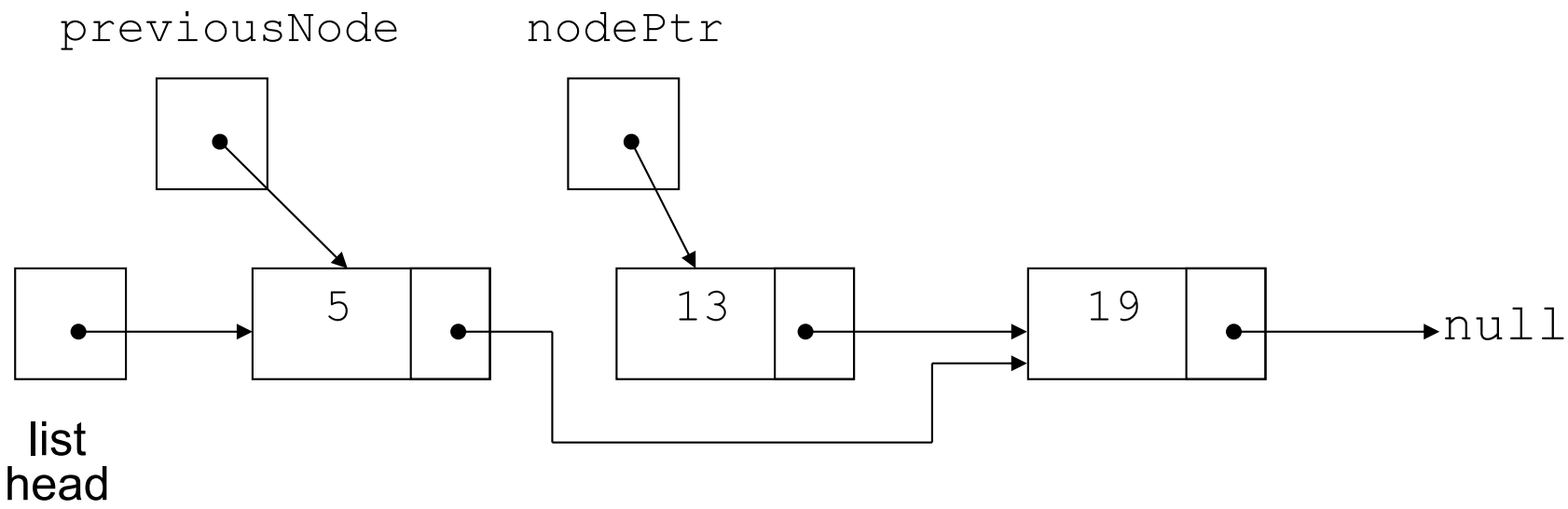
- Used to remove a node from a linked list
- If list uses dynamic memory, then delete node from memory
- Requires two pointers: one to locate the node to be deleted, one to point to the node before the node to be deleted

Deleting a Node



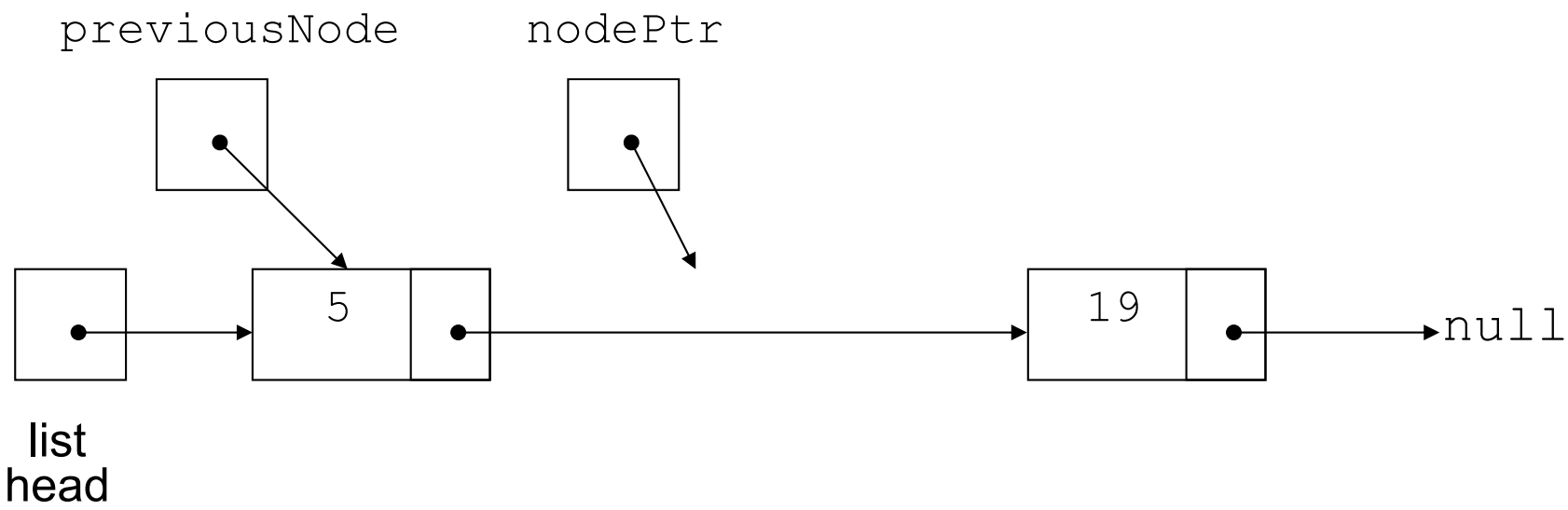
Locating the node containing `13`

Deleting a Node



Adjusting pointer around the node to be deleted

Deleting a Node



Linked list after deleting the node containing `13`

```
    void deleteNode(ListNode * head, double num)
123 {
124     ListNode *nodePtr;      // To traverse the list
125     ListNode *previousNode; // To point to the previous node
126
127     // If the list is empty, do nothing.
128     if (!head)
129         return;
130
```

```
131         // Determine if the first node is the one.
132         if (head->value == num)
133         {
134             nodePtr = head->next;
135             delete head;
136             head = nodePtr;
137         }
138         else
139         {
140             // Initialize nodePtr to head of list
141             nodePtr = head;
142
143             // Skip all nodes whose value member is
144             // not equal to num.
145             while (nodePtr != nullptr && nodePtr->value != num)
146             {
147                 previousNode = nodePtr;
148                 nodePtr = nodePtr->next;
149             }
150
```

```
151             // If nodePtr is not at the end of the list,
152             // link the previous node to the node after
153             // nodePtr, then delete nodePtr.
154             if (nodePtr)
155             {
156                 previousNode->next = nodePtr->next;
157                 delete nodePtr;
158             }
159         }
160 }
```

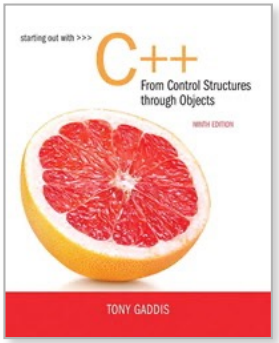
The main function:

```
Int main()
{
        ListNode * head=nullptr;

        insertNode(head, 4.5);
        insertNode(head, 8.2);
        insertNode(head, 6.7);

        deleteNode(head, 8.2);

        return 0;
}
```

Destroying a Linked List

- Must remove all nodes used in the list
- To do this, use list traversal to visit each node
- For each node,
  - Unlink the node from the list
  - If the list uses dynamic memory, then free the node's memory
- Set the list head to `nullptr`

```
void DestroyList(ListNode * head)
{
    ListNode *nodePtr;   // To traverse the list
    ListNode *nextNode;  // To point to the next node

    // Position nodePtr at the head of the list.
    nodePtr = head;

    // While nodePtr is not at the end of the list...
    while (nodePtr != nullptr)
    {
        // Save a pointer to the next node.
        nextNode = nodePtr->next;

        // Delete the current node.
        delete nodePtr;

        // Position nodePtr at the next node.
        nodePtr = nextNode;
    }
}
```

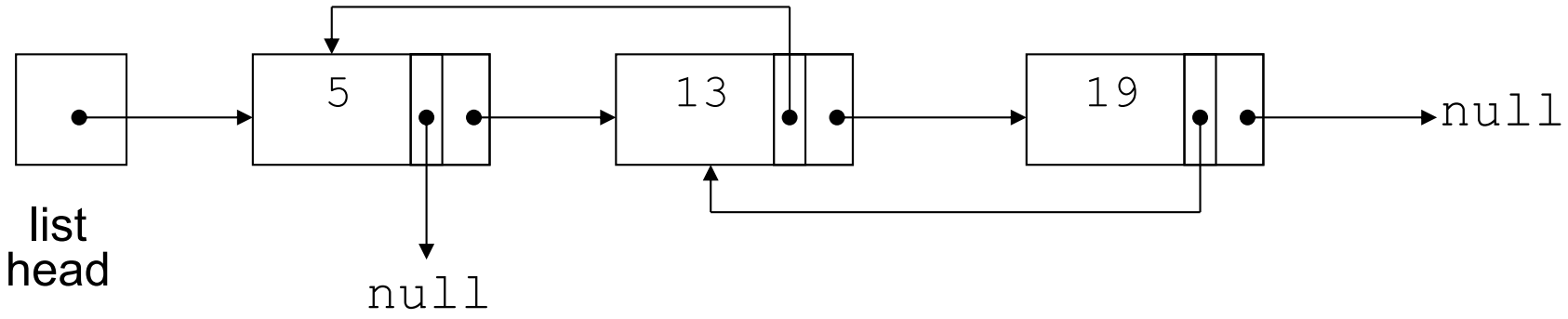


18.4

Variations of the Linked List

Variations of the Linked List

- Other linked list organizations:
  - doubly-linked list: each node contains two pointers: one to the next node in the list, one to the previous node in the list



Variations of the Linked List

- Other linked list organizations:
  - circular linked list: the last node in the list points back to the first node in the list, not to the null pointer