Linked Lists

By: Anita Rathi

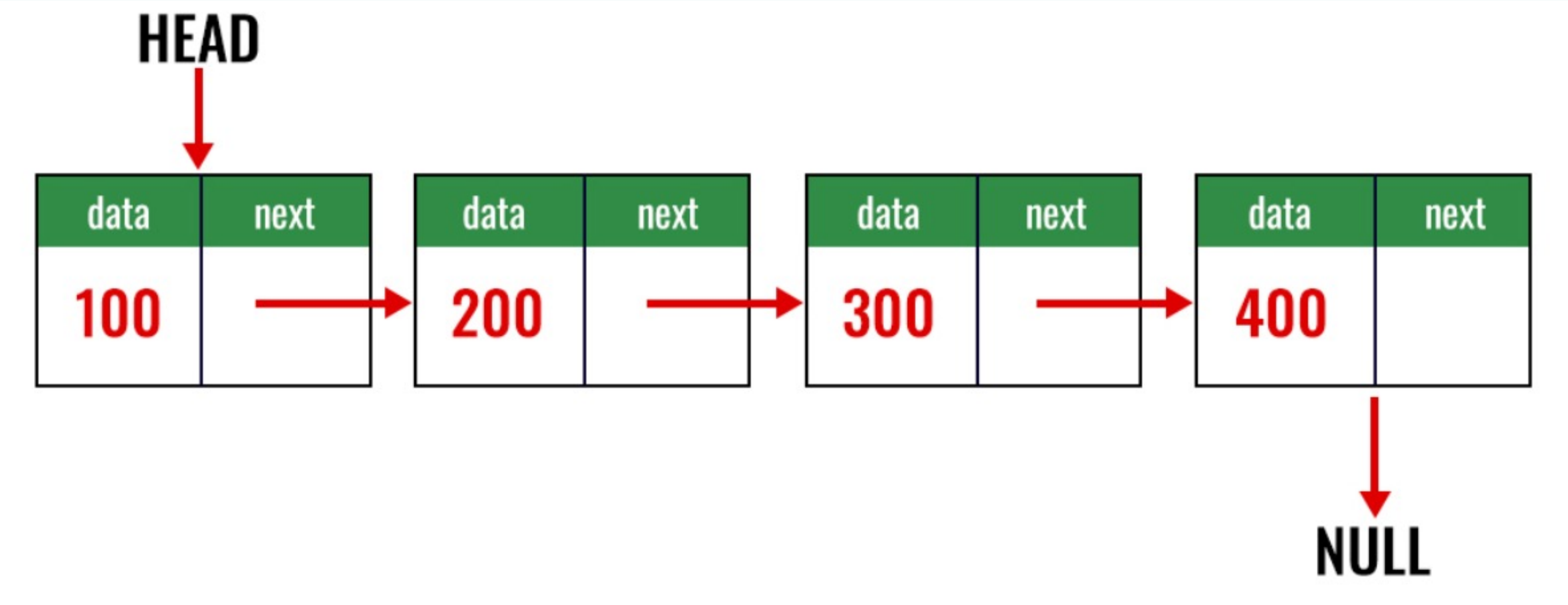
Linked List

- A linked list is a linear collection of data elements whose order is not given by their physical placement in memory.
- Each element points to the next.
- It is a data structure consisting of a collection of nodes which together represent a sequence.
- A node has two parts
    - Data part – contains the data
    - Address part – contains the address of the next node

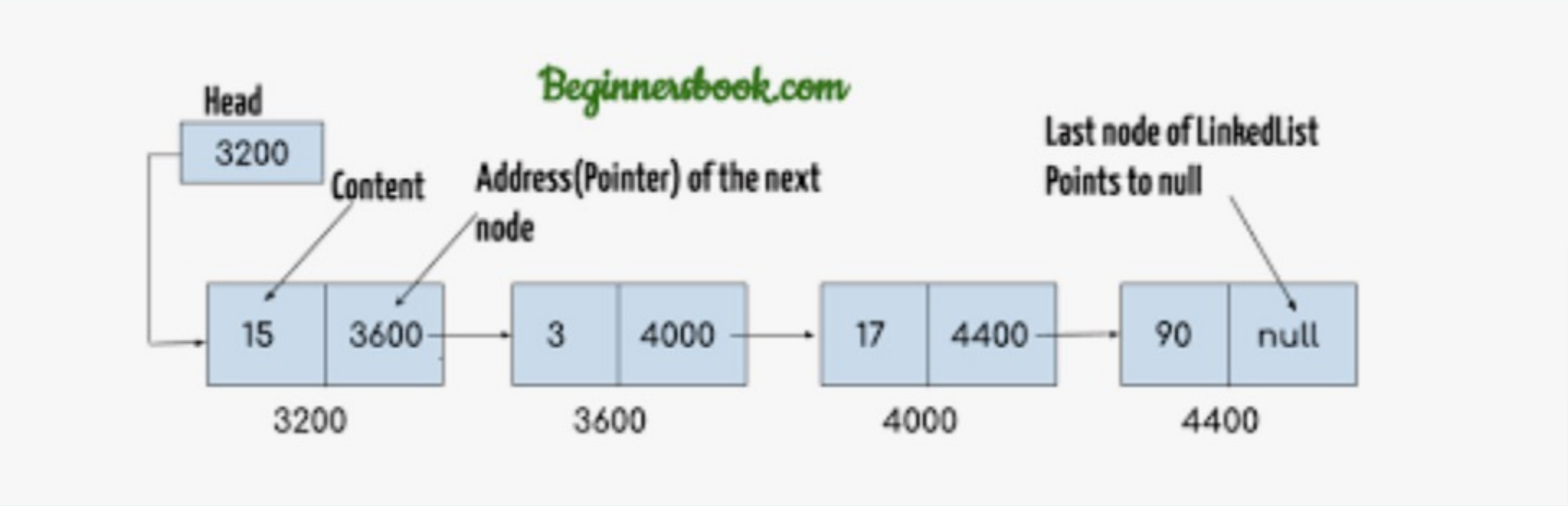
Linked List

- Each element/node in the linked list points to the next node.
- It is a data structure consisting of a collection of nodes which together represent a sequence.

Linked List

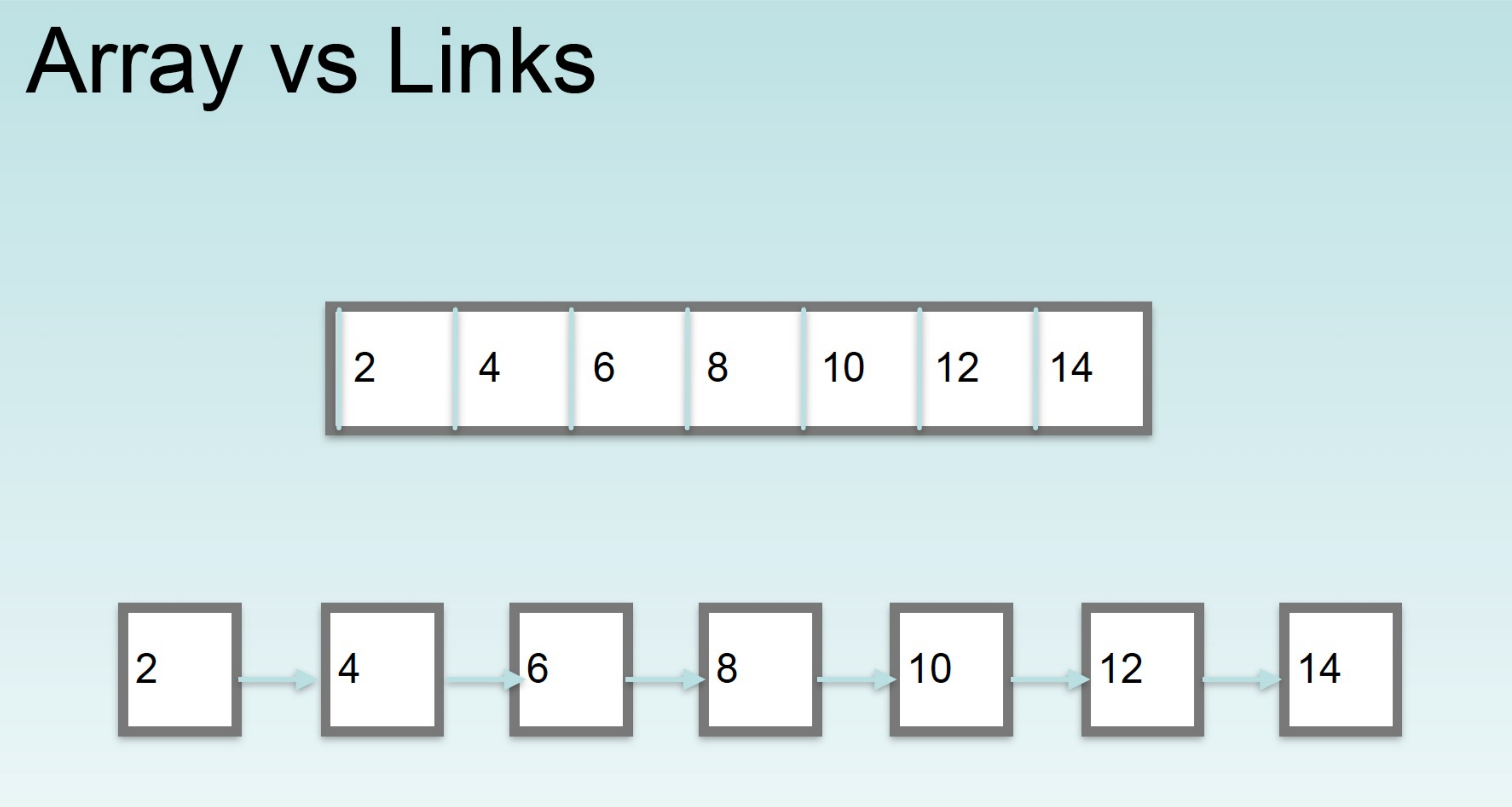

HEAD
data
next
100
data
next
200
data
next
300
data
next
400
NULL

Linked List


Beginnersbook.com
Head
3200
Content
Address(Pointer) of the next node
Last node of LinkedList Points to null
15
3600
3
4000
17
4400
90
null
3200
3600
4000
4400

- Sources – beginnersbook.com

Linked List



Array vs Links

2	4	6	8	10	12	14

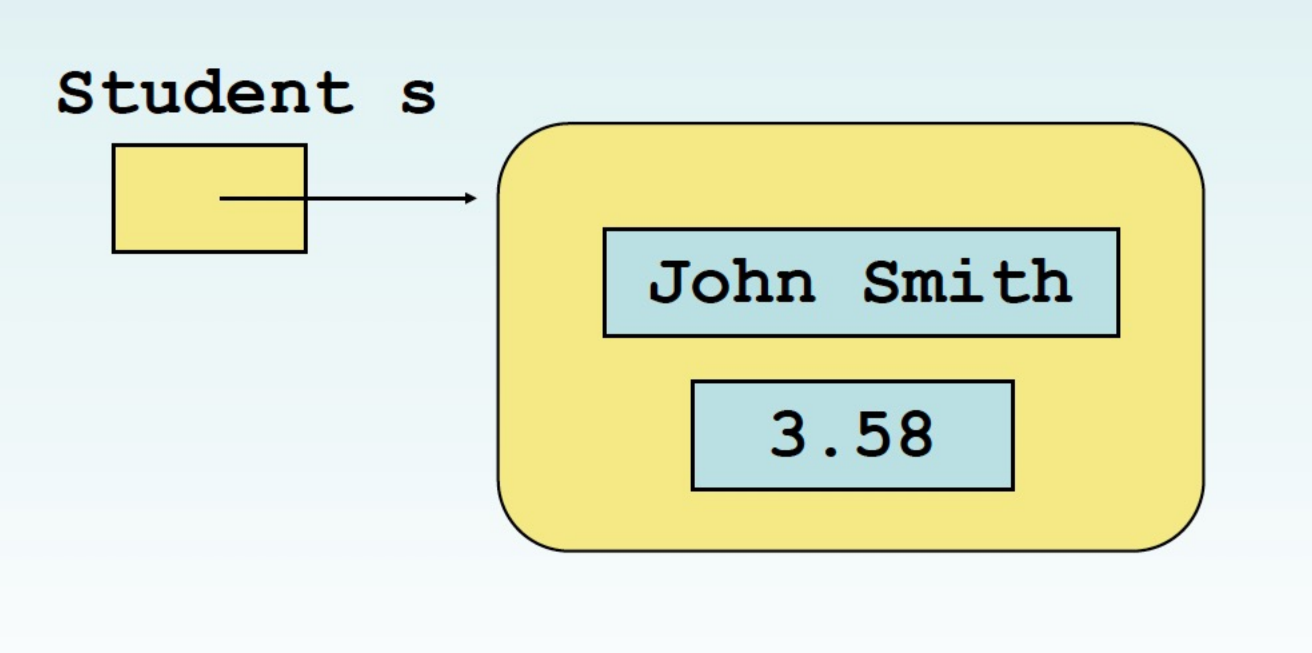
Linked List

- How do we create these nodes and link them together?
- The first node is typically called the head node or headptr. You may name it as First as well.
- Note: The data part could be a integer, String, float, or even an Object of a class type.

Linked List

Recall : Object References

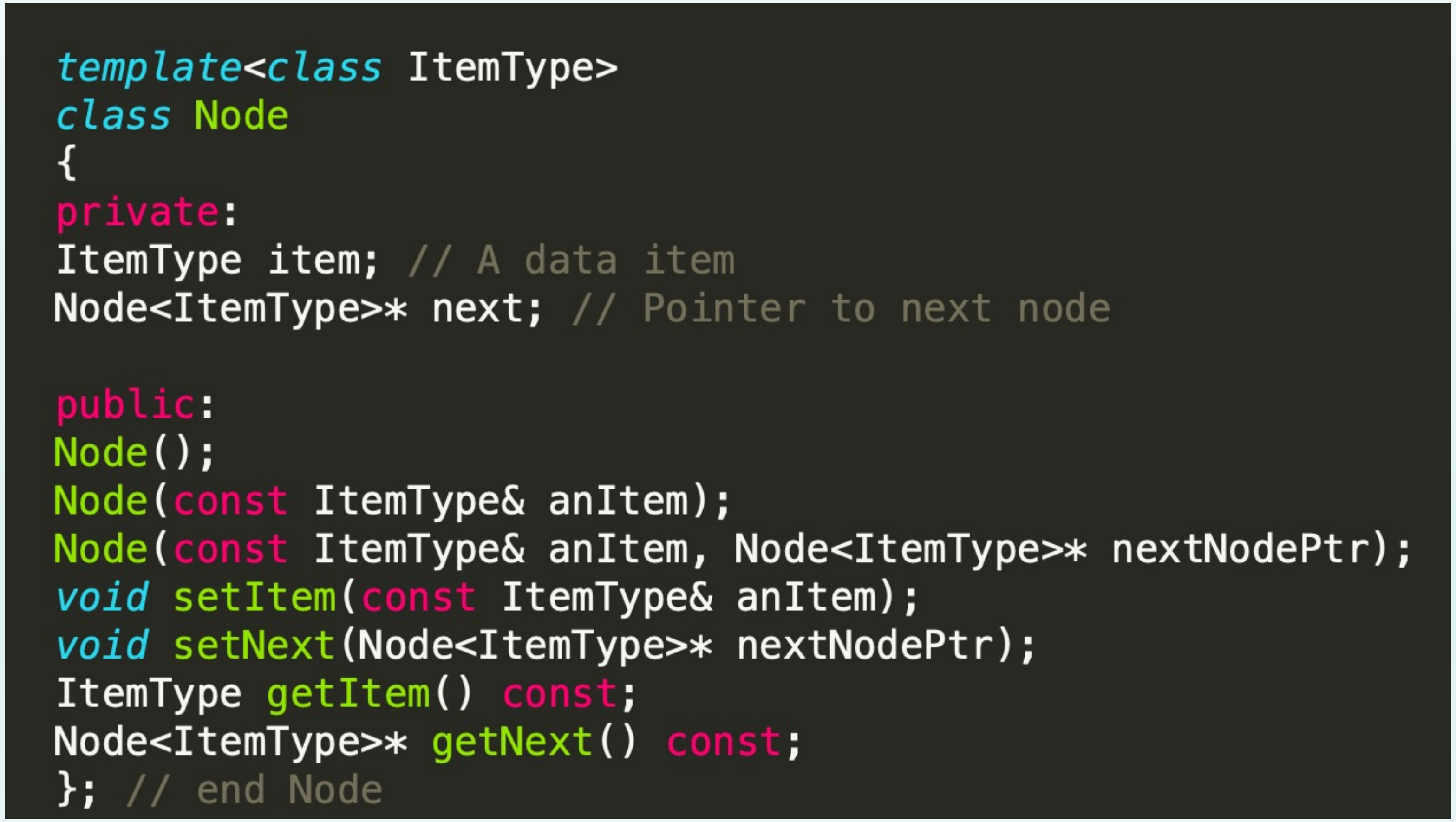
- An object reference is a variable that stores an address of the object.
- A reference can also be called a pointer.
- References often are depicted graphically:



Linked List

Node of Generic type

- The data type of data is represented here by the generic type ItemType. **Sample Code – Node.h**



```
template<class ItemType>
class Node
{
private:
ItemType item; // A data item
Node<ItemType>* next; // Pointer to next node

public:
Node();
Node(const ItemType& anItem);
Node(const ItemType& anItem, Node<ItemType>* nextNodePtr);
void setItem(const ItemType& anItem);
void setNext(Node<ItemType>* nextNodePtr);
ItemType getItem() const;
Node<ItemType>* getNext() const;
}; // end Node
```

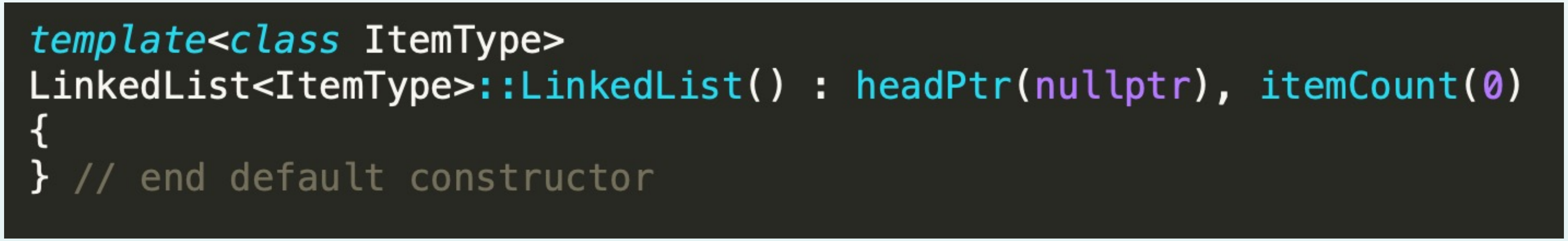
Accessing Nodes in Linked List

- Although you have direct access to any of an array’s elements, you must traverse a chain of linked nodes to locate a specified node.
- A list’s retrieval, insertion, and removal operations require such a traversal.
- Implementation of **Node.h** in **Node.cpp**.

List Interface

- The List interface will now be implemented using Linked List instead of Array List.
- Sample Code **ListInterface.h**
- A copy constructor and a destructor are necessary for a link-based implementation

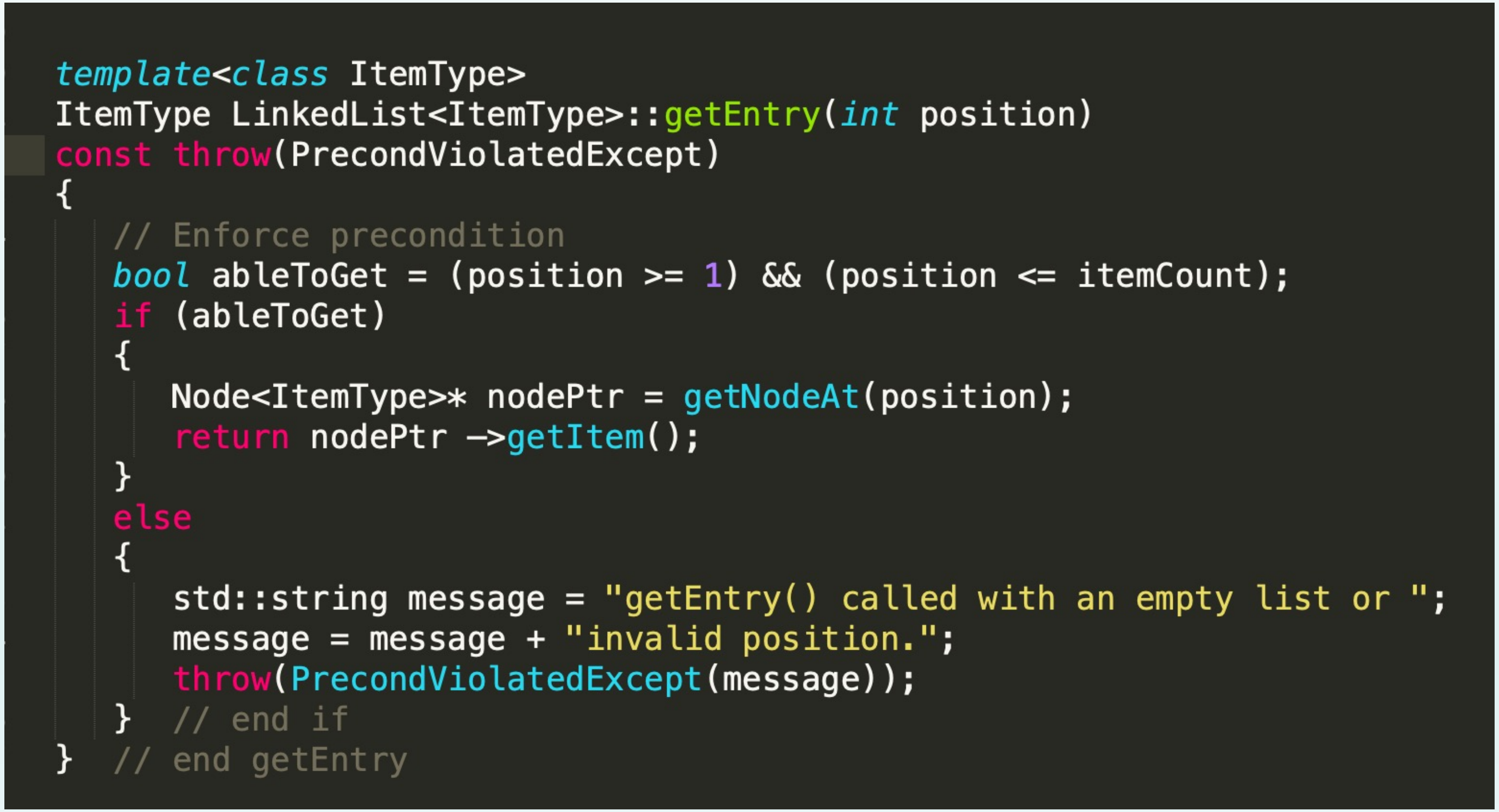
Definition of Constructor Linked List



```
template<class ItemType>
LinkedList<ItemType>::LinkedList() : headPtr(nullptr), itemCount(0)
{
} // end default constructor
```

Definition of getEntry()

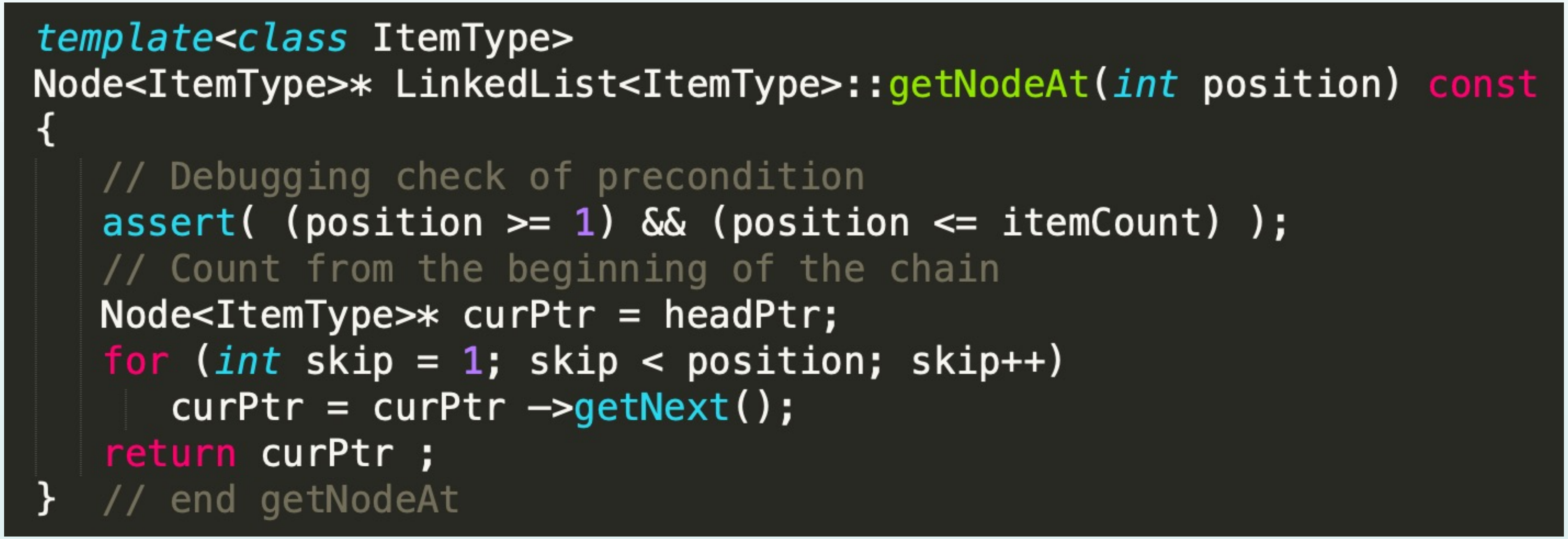
- getEntry() in LinkedList, will enforce its precondition by throwing an exception if position is out of bounds.



```cpp
template<class ItemType>
ItemType LinkedList<ItemType>::getEntry(int position)
const throw(PrecondViolatedExcept)
{
   // Enforce precondition
   bool ableToGet = (position >= 1) && (position <= itemCount);
   if (ableToGet)
   {
      Node<ItemType>* nodePtr = getNodeAt(position);
      return nodePtr ->getItem();
   }
   else
   {
      std::string message = "getEntry() called with an empty list or ";
      message = message + "invalid position.";
      throw(PrecondViolatedExcept(message));
   }  // end if
}  // end getEntry
```

Definition of getNodeAt()

- getNodeAt() locates the node at a given position by traversing the chain. It then returns a pointer to the located node.

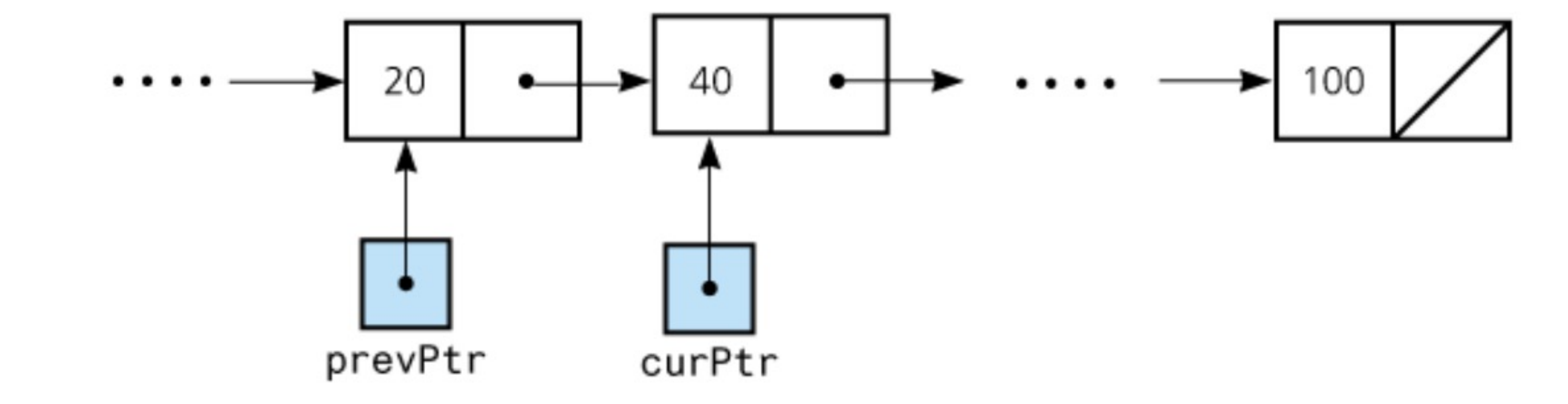


```
template<class ItemType>
Node<ItemType>* LinkedList<ItemType>::getNodeAt(int position) const
{
   // Debugging check of precondition
   assert( (position >= 1) && (position <= itemCount) );
   // Count from the beginning of the chain
   Node<ItemType>* curPtr = headPtr;
   for (int skip = 1; skip < position; skip++)
      curPtr = curPtr ->getNext();
   return curPtr ;
}  // end getNodeAt
```

Insertion of new node in between two nodes

- To insert a new node in between two nodes use the following steps-
  1. Create a new Node(newNodePtr).
  2. Traverse to a node (prevPtr) previous to the node (curPtr) where insertion needs to be done.

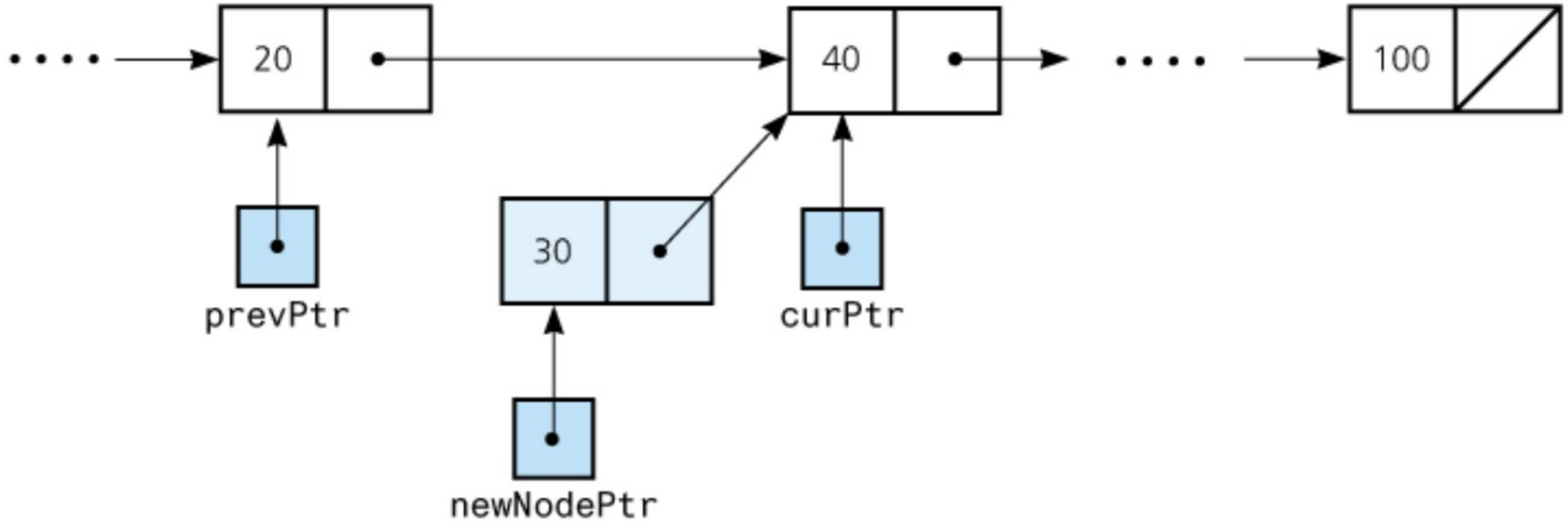
(a) Before the insertion of a new node



Insertion of new node in between two nodes

2. Connect the next of newNodePtr to the curPtr.

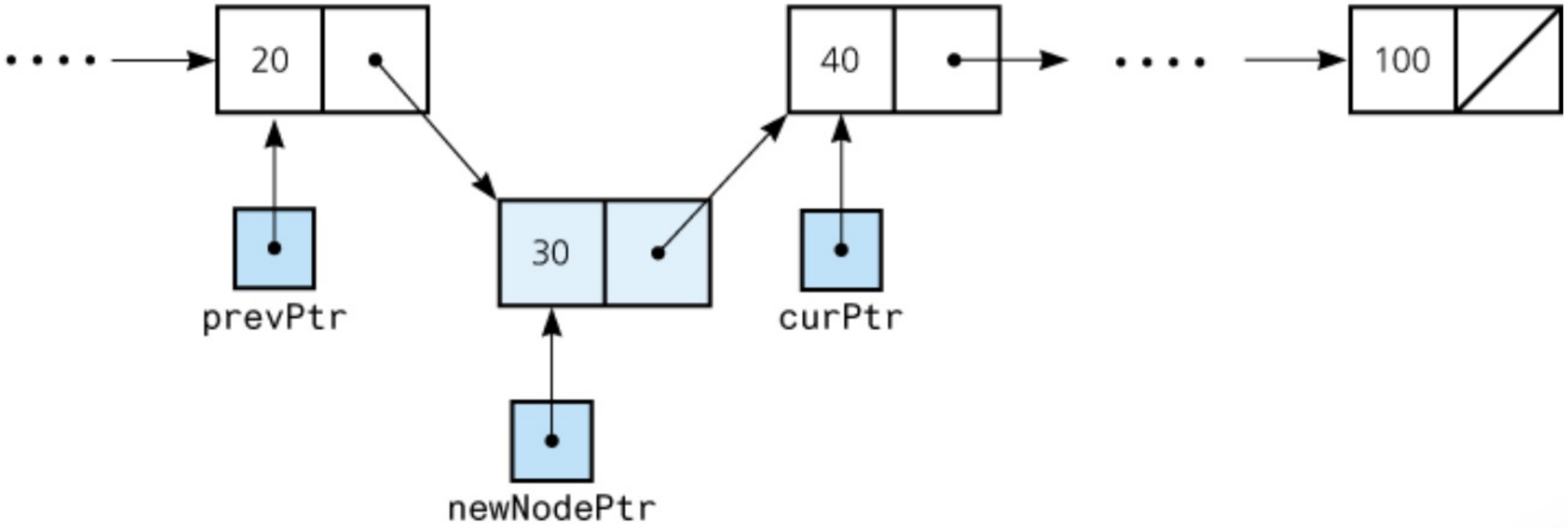
(b) After newNodePtr->setNext(curPtr) executes



Insertion of new node in between two nodes

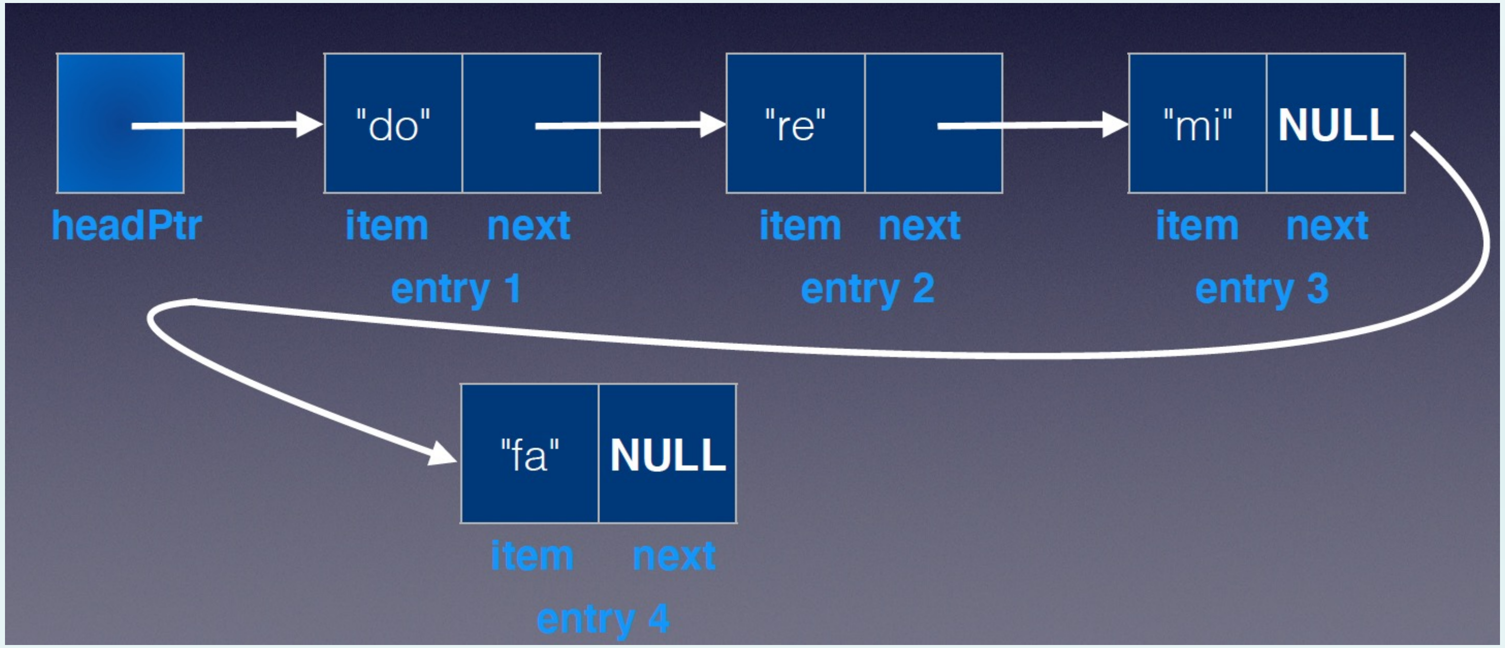
2. Connect the next of prevPtr to the newNodePtr.

(c) After prevPtr->setNext(newNodePtr) executes



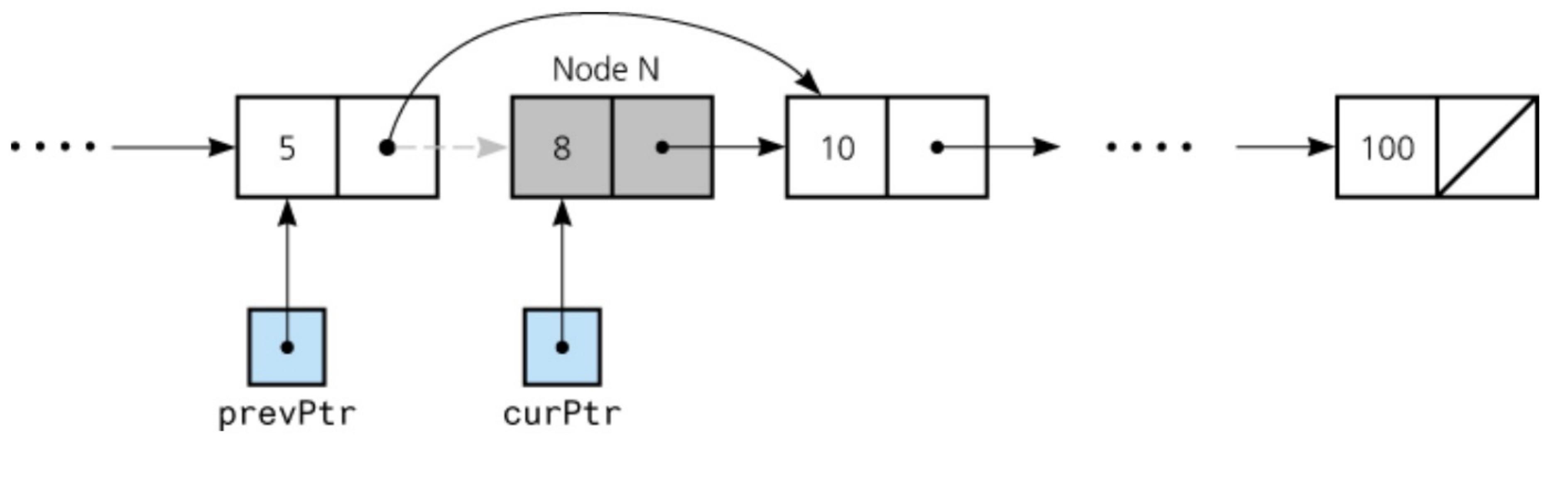
Insertion of new node at the end of Linked List

- Inserting a new node at the end of the list is simpler. To insert at the end of the list use the following steps:
  1. Create a new node
  2. Traverse to the end of the Linked List.
  3. Connect the nextPtr of last node to the new node.



Deletion of a node in Linked List

- To delete a node in linked list use the following steps:
  1. Traverse to the previous node(prevPtr) of the node(curPtr) to be deleted in the Linked List.
  2. Connect the nextPtr of prevPtr to the nextPtr of curPtr.
  3. Delete curPtr.

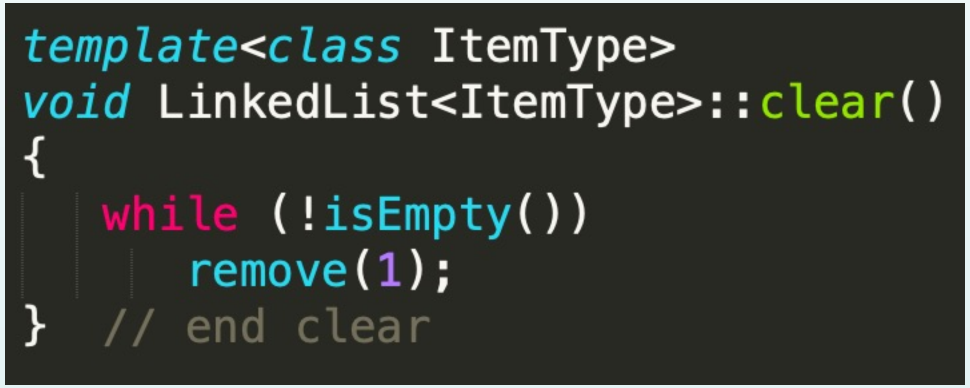


Deletion of last node in Linked List

- To delete the last node in linked list use the following steps:
  1. Traverse to the previous node(prevPtr) of the last node(curPtr) in the Linked List.
  2. Make the nextPtr of prevPtr as nullptr.
  3. Delete curPtr.

clear() in Linked List

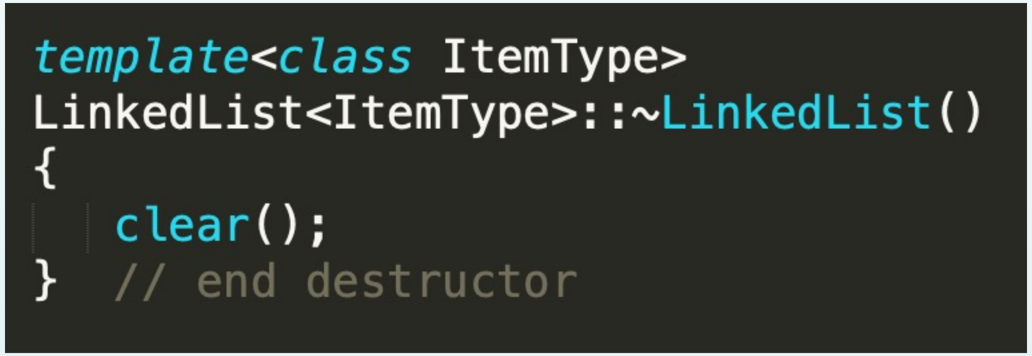
- To clear the linked list use remove() till the Linked List has no nodes.



```
template<class ItemType>
void LinkedList<ItemType>::clear()
{
    while (!isEmpty())
        remove(1);
}  // end clear
```

Destructor in Linked List

- The destructor can simply call clear() as clear() invokes remove repeatedly until the list is empty, and remove deallocates the nodes it removes.



```
template<class ItemType>
LinkedList<ItemType>::~LinkedList()
{
    clear();
}  // end destructor
```