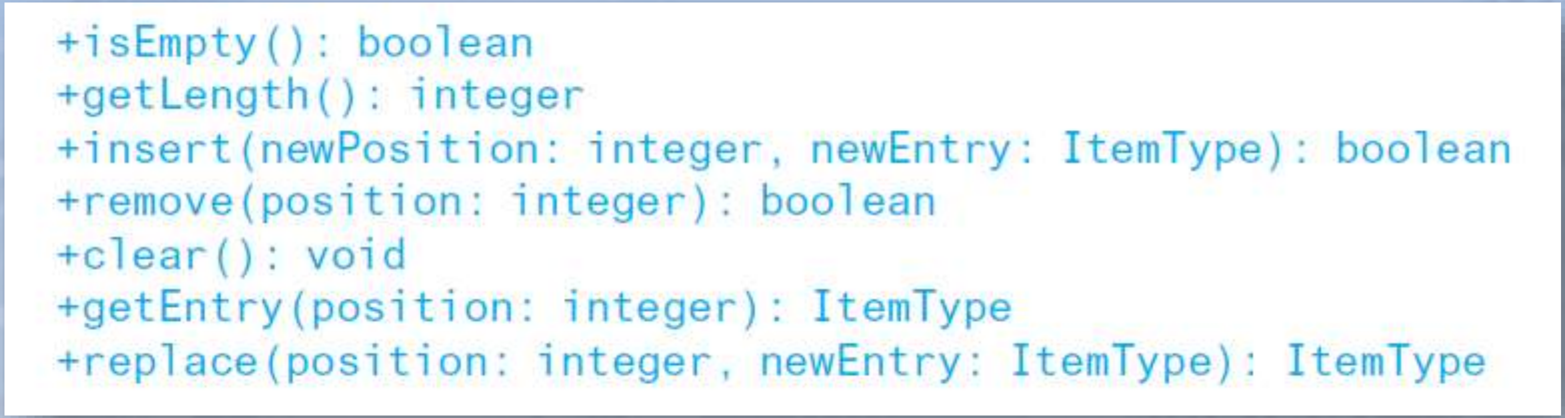
List Implementations

Chapter 9

Array-Based Implementation of the ADT List



```
+isEmpty(): boolean
+getLength(): integer
+insert(newPosition: integer, newEntry: ItemType): boolean
+remove(position: integer): boolean
+clear(): void
+getEntry(position: integer): ItemType
+replace(position: integer, newEntry: ItemType): ItemType
```

List operations in their UML form

Array-Based Implementation of the ADT List

- Array-based implementation is a natural choice
  - Both an array and a list identify their items by number
- However
  - ADT list has operations such as getLength that an array does not
  - Must keep track of number of entries

The Header File

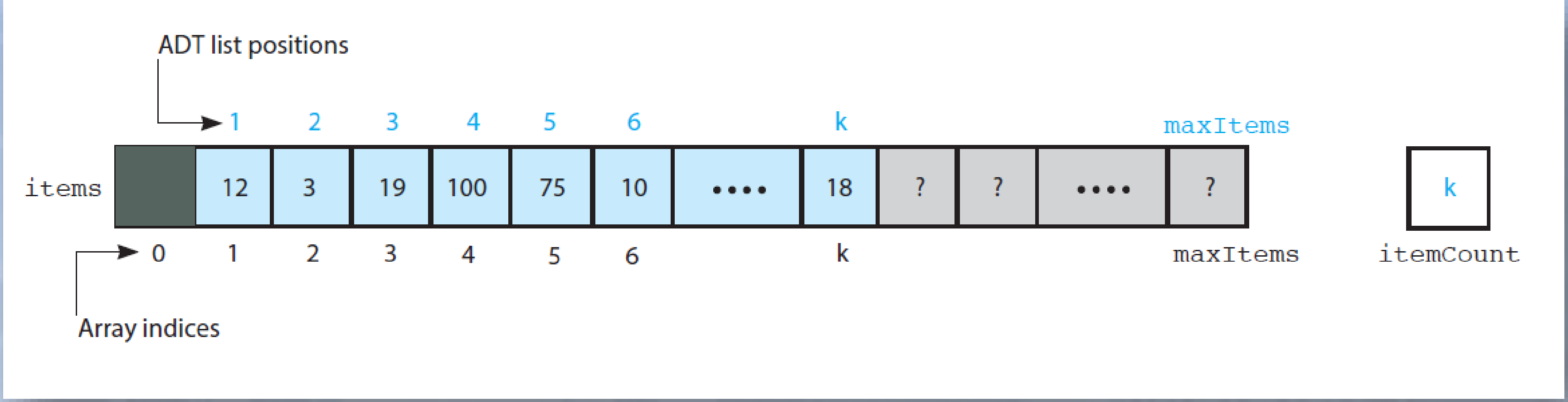
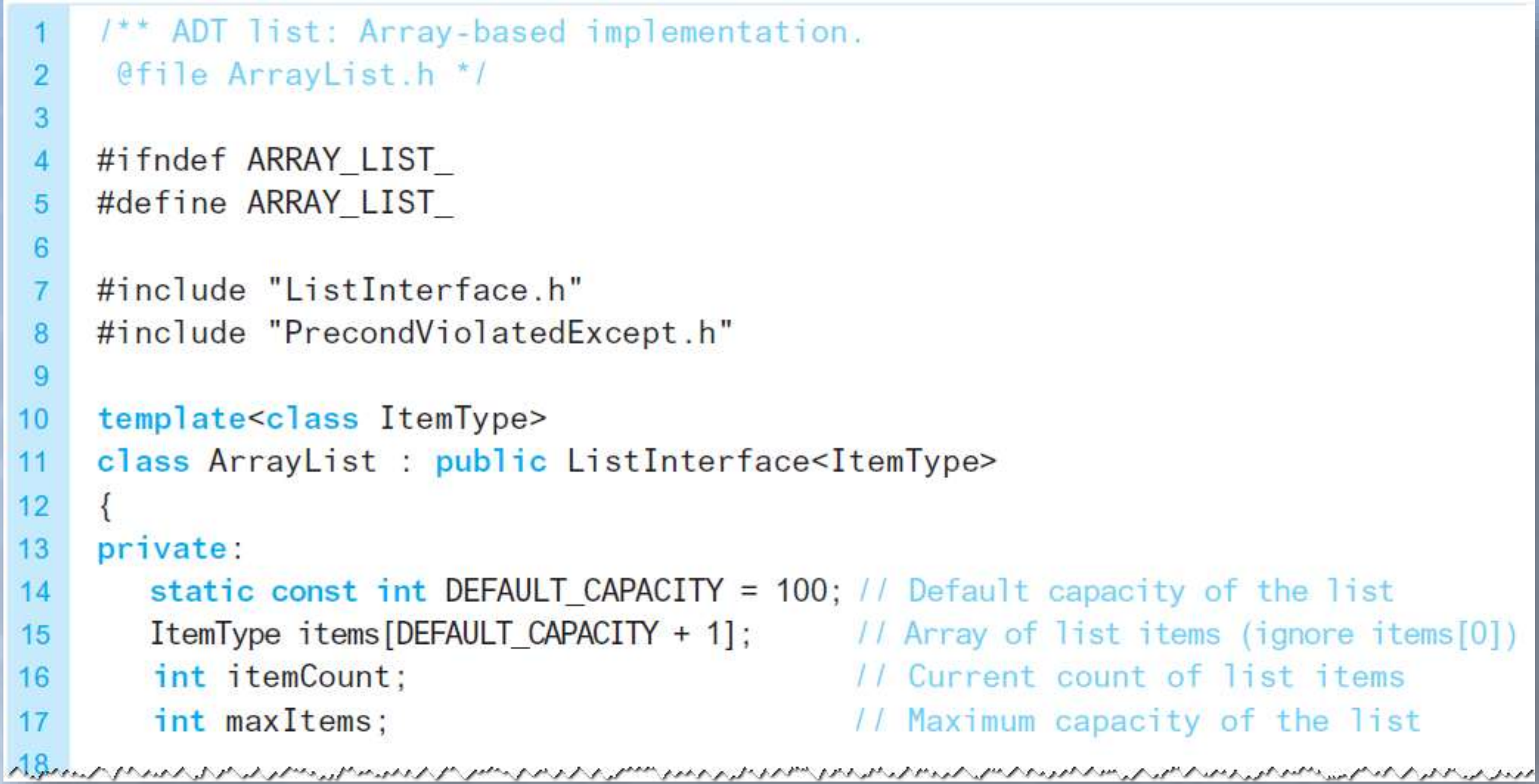


FIGURE 9-1 An array-based implementation of the ADT list

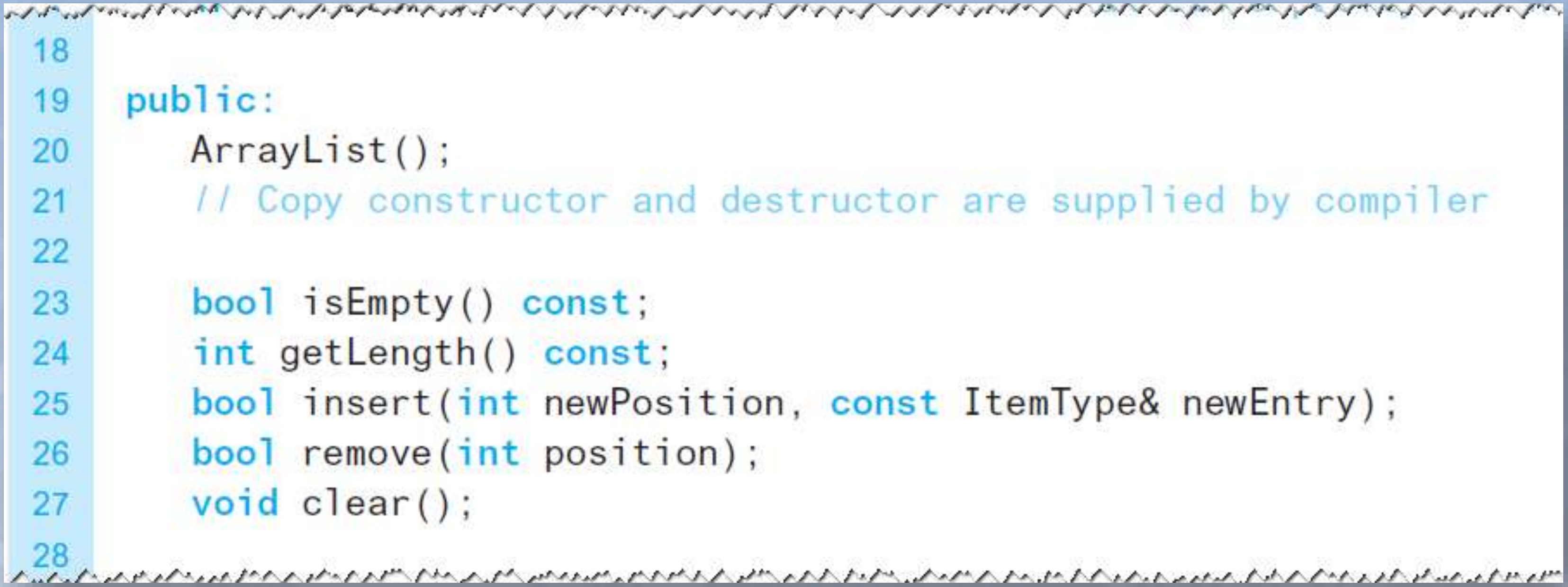
The Header File



```
/** ADT list: Array-based implementation.
 @file ArrayList.h */

#ifndef ARRAY_LIST_
#define ARRAY_LIST_

#include "ListInterface.h"
#include "PrecondViolatedExcept.h"

template<class ItemType>
class ArrayList : public ListInterface<ItemType>
{
private:
   static const int DEFAULT_CAPACITY = 100; // Default capacity of the list
   ItemType items[DEFAULT_CAPACITY + 1];    // Array of list items (ignore items[0])
   int itemCount;                           // Current count of list items
   int maxItems;                            // Maximum capacity of the list
```

LISTING 9-1 The header file for the class ArrayList

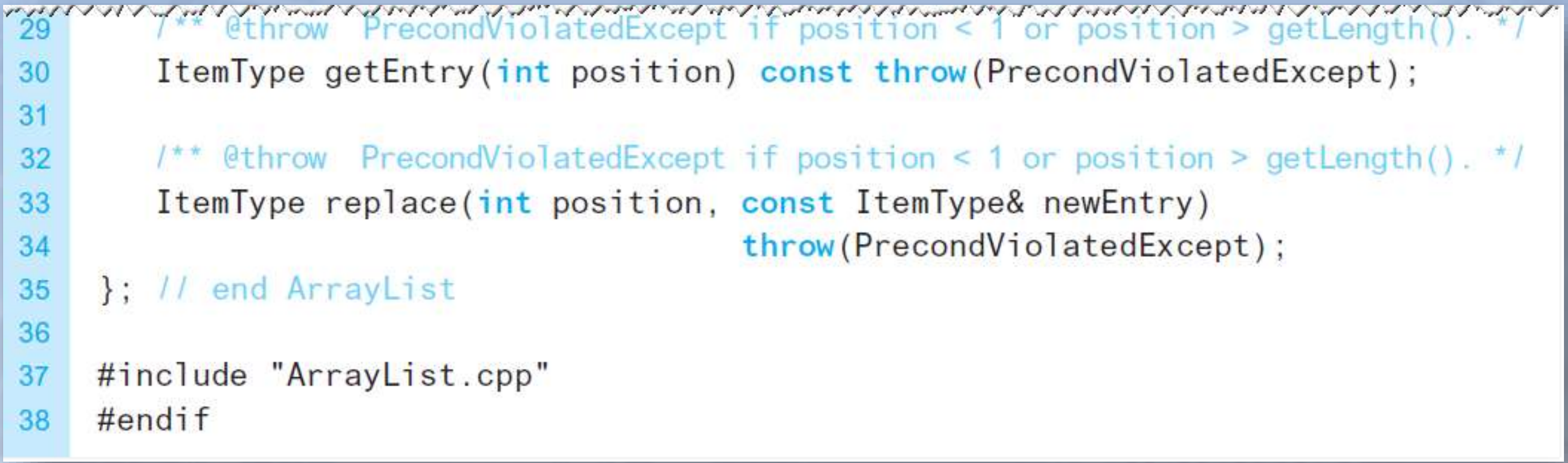
The Header File



```

public:
   ArrayList();
   // Copy constructor and destructor are supplied by compiler

   bool isEmpty() const;
   int getLength() const;
   bool insert(int newPosition, const ItemType& newEntry);
   bool remove(int position);
   void clear();

```

LISTING 9-1 The header file for the class ArrayList

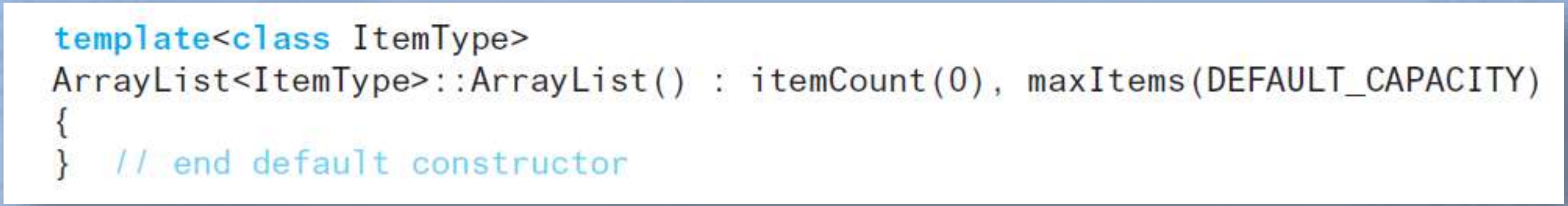
The Header File



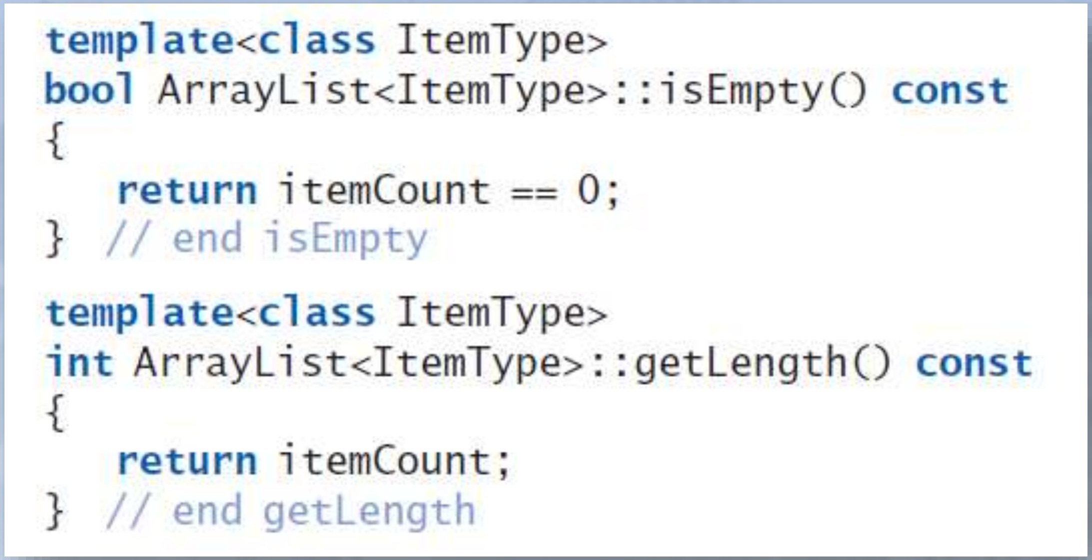
```
   /** @throw  PrecondViolatedExcept if position < 1 or position > getLength(). */
   ItemType getEntry(int position) const throw(PrecondViolatedExcept);

   /** @throw  PrecondViolatedExcept if position < 1 or position > getLength(). */
   ItemType replace(int position, const ItemType& newEntry)
                                  throw(PrecondViolatedExcept);
}; // end ArrayList

#include "ArrayList.cpp"
#endif
```

LISTING 9-1 The header file for the class ArrayList

The Implementation File



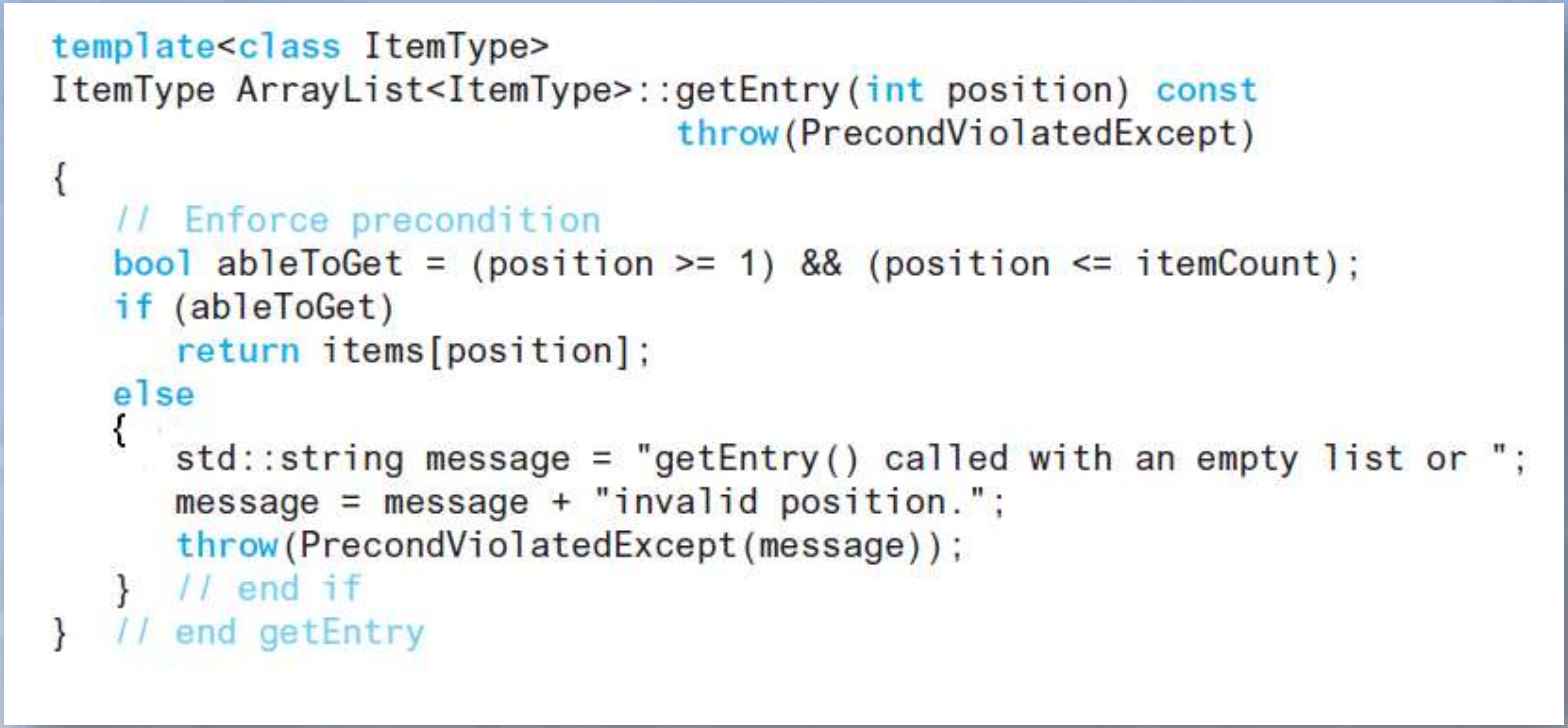
```
template<class ItemType>
ArrayList<ItemType>::ArrayList() : itemCount(0), maxItems(DEFAULT_CAPACITY)
{
}  // end default constructor
```



```
template<class ItemType>
bool ArrayList<ItemType>::isEmpty() const
{
   return itemCount == 0;
}  // end isEmpty

template<class ItemType>
int ArrayList<ItemType>::getLength() const
{
   return itemCount;
}  // end getLength
```

Constructor, methods isEmpty and getLength

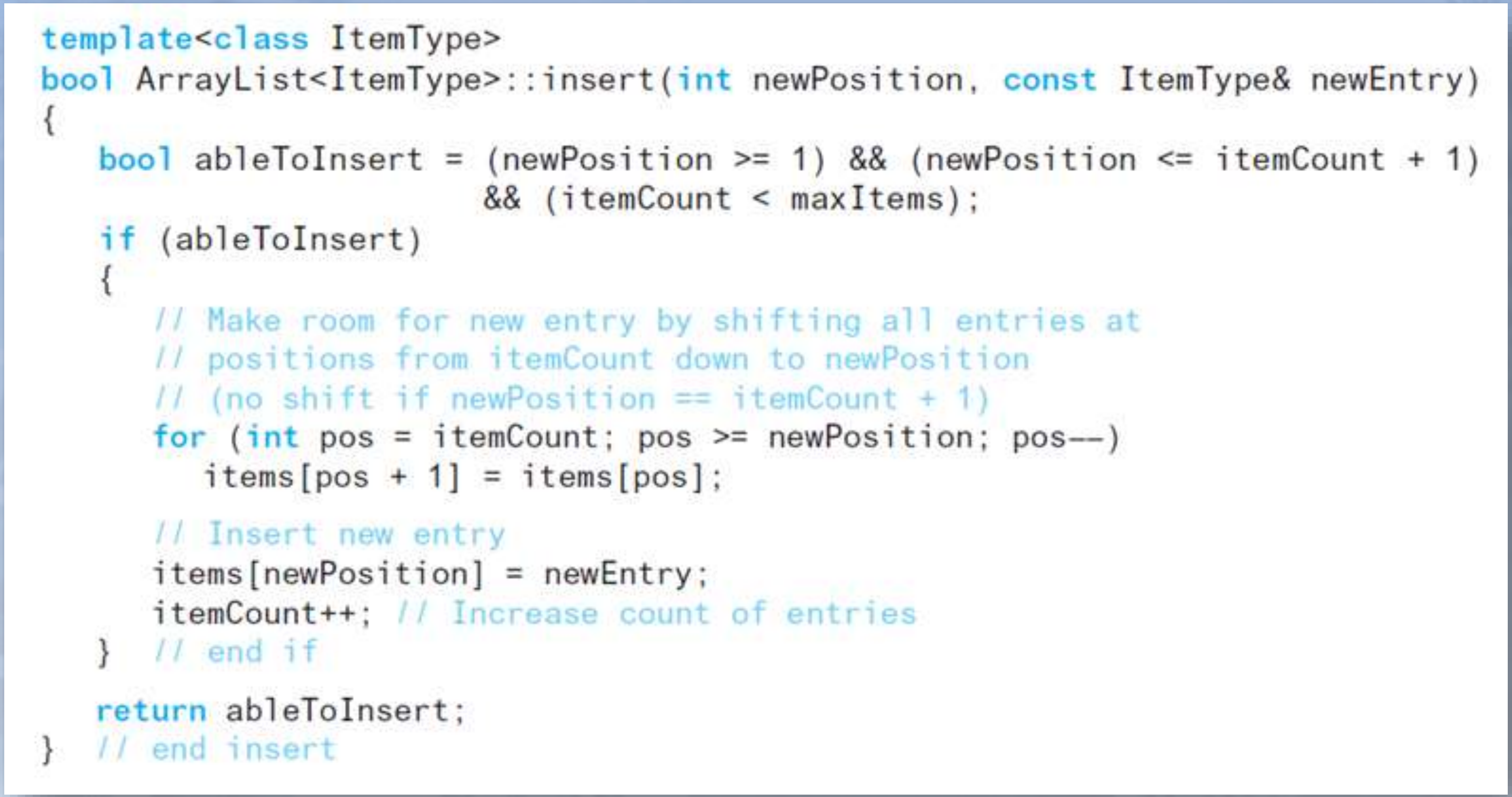
The Implementation File



```
template<class ItemType>
ItemType ArrayList<ItemType>::getEntry(int position) const
                              throw(PrecondViolatedExcept)
{
   // Enforce precondition
   bool ableToGet = (position >= 1) && (position <= itemCount);
   if (ableToGet)
      return items[position];
   else
   {
      std::string message = "getEntry() called with an empty list or ";
      message = message + "invalid position.";
      throw(PrecondViolatedExcept(message));
   }  // end if
}  // end getEntry
```

Method getEntry

The Implementation File



```
template<class ItemType>
bool ArrayList<ItemType>::insert(int newPosition, const ItemType& newEntry)
{
   bool ableToInsert = (newPosition >= 1) && (newPosition <= itemCount + 1)
                       && (itemCount < maxItems);
   if (ableToInsert)
   {
      // Make room for new entry by shifting all entries at
      // positions from itemCount down to newPosition
      // (no shift if newPosition == itemCount + 1)
      for (int pos = itemCount; pos >= newPosition; pos--)
         items[pos + 1] = items[pos];

      // Insert new entry
      items[newPosition] = newEntry;
      itemCount++; // Increase count of entries
   }  // end if

   return ableToInsert;
}  // end insert
```

Method insert

The Implementation File

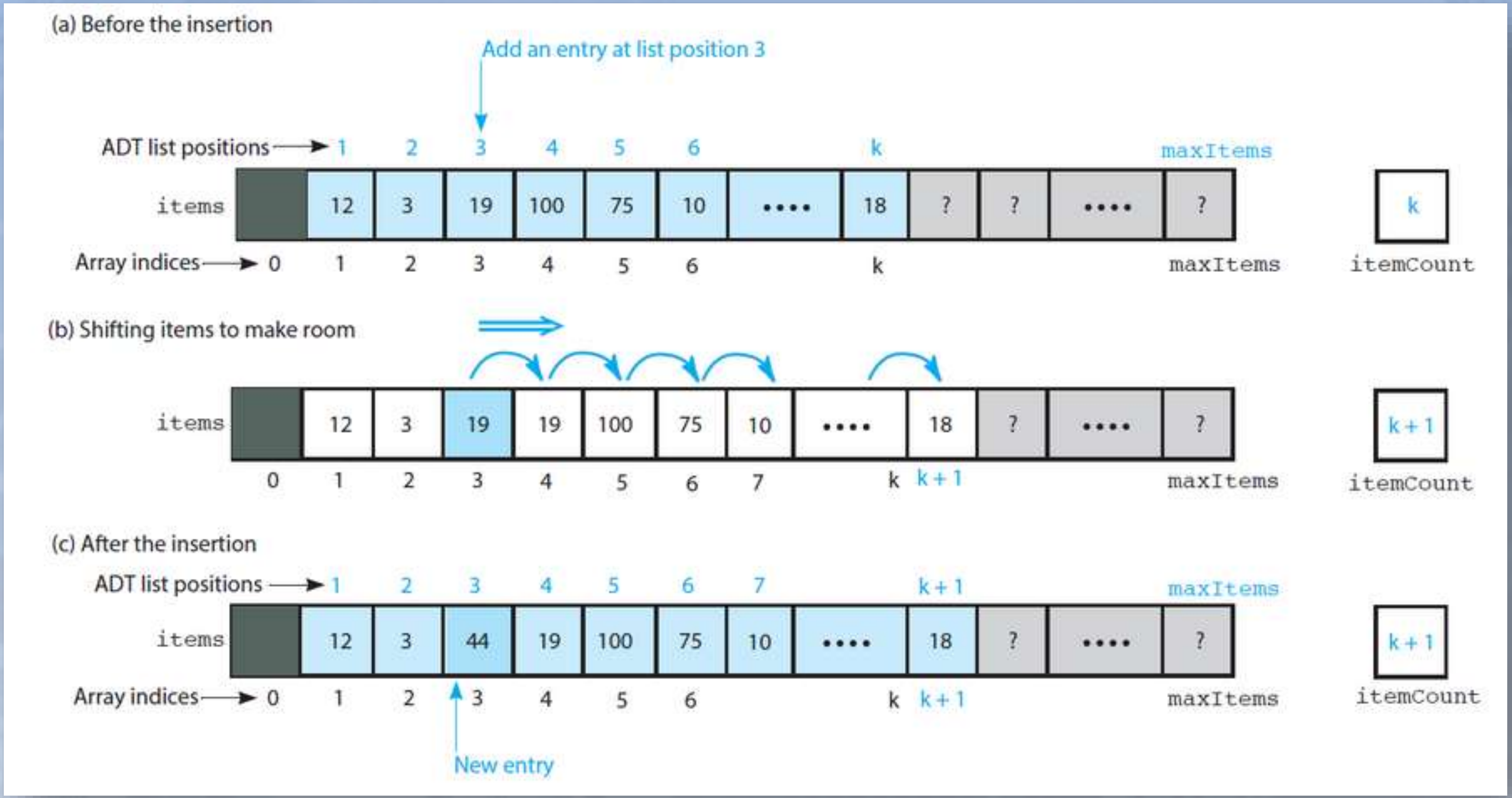
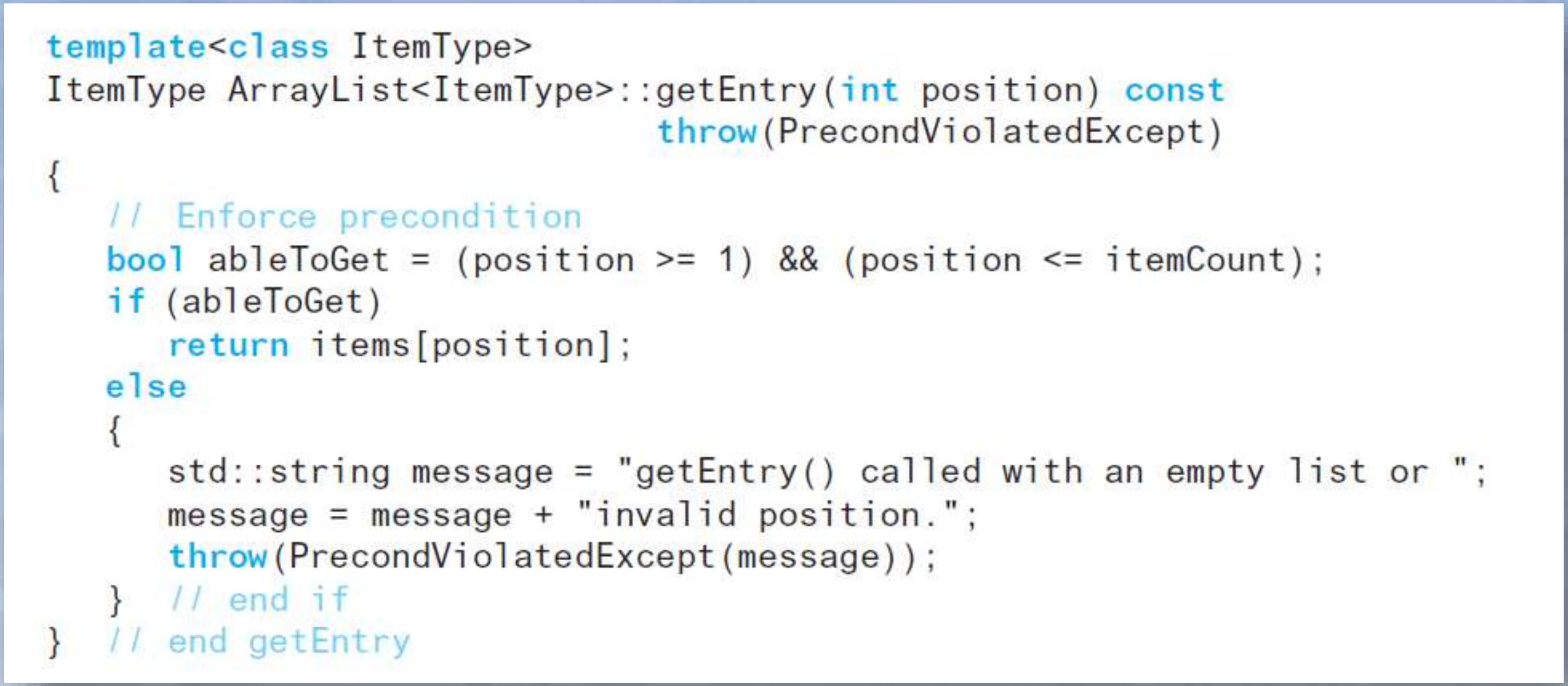

(a) Before the insertion
Add an entry at list position 3
ADT list positions → 1 2 3 4 5 6 k maxItems
items 12 3 19 100 75 10 •••• 18 ? ? •••• ?
Array indices → 0 1 2 3 4 5 6 k maxItems
k
itemCount
(b) Shifting items to make room
items 12 3 19 19 100 75 10 •••• 18 ? •••• ?
0 1 2 3 4 5 6 7 k k + 1 maxItems
k + 1
itemCount
(c) After the insertion
ADT list positions → 1 2 3 4 5 6 7 k + 1 maxItems
items 12 3 44 19 100 75 10 •••• 18 ? •••• ?
Array indices → 0 1 2 3 4 5 6 k k + 1 maxItems
k + 1
itemCount
New entry

FIGURE 9-2 Shifting items for insertion

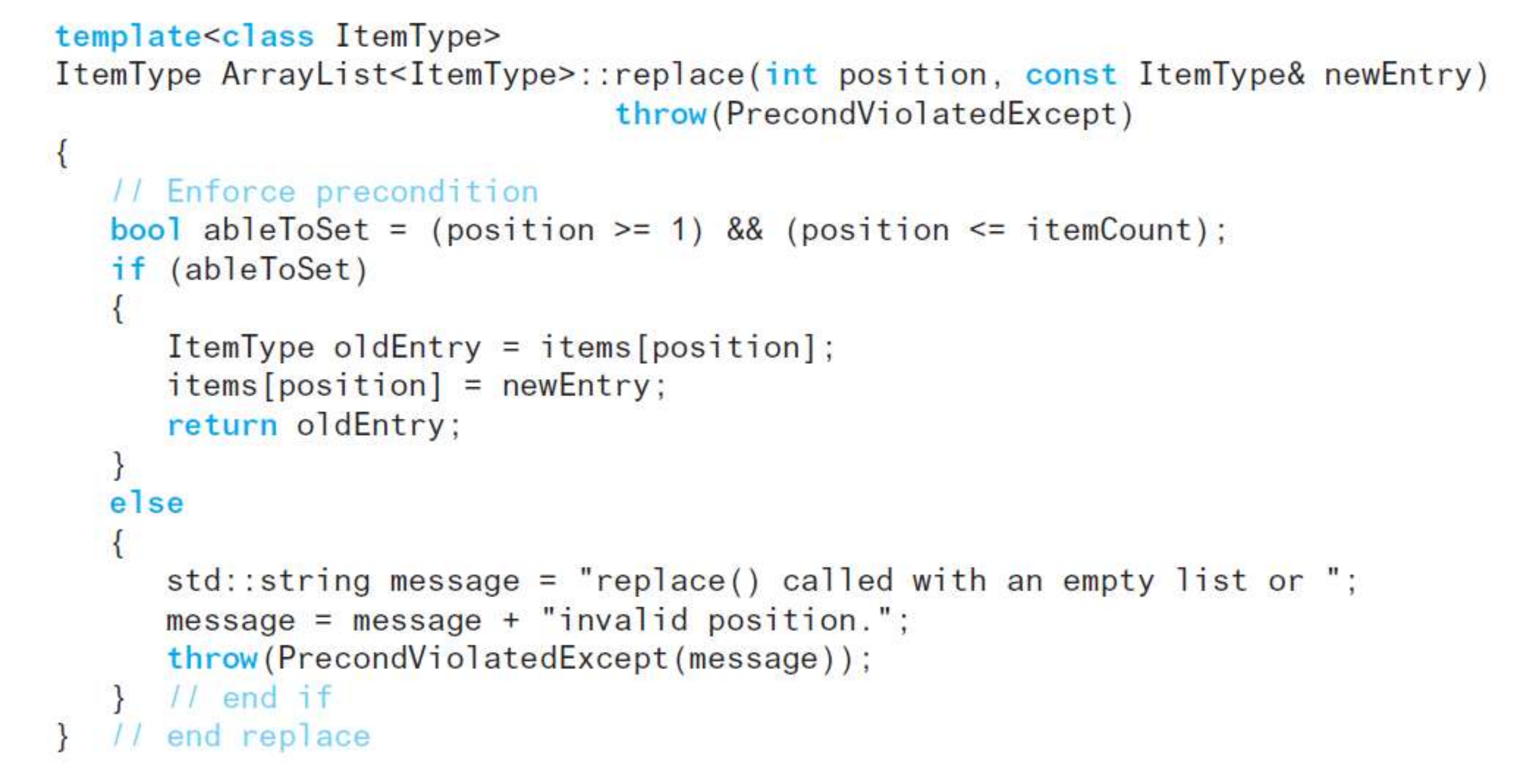
The Implementation File



```
template<class ItemType>
ItemType ArrayList<ItemType>::getEntry(int position) const
                                       throw(PrecondViolatedExcept)
{
   //  Enforce precondition
   bool ableToGet = (position >= 1) && (position <= itemCount);
   if (ableToGet)
      return items[position];
   else
   {
      std::string message = "getEntry() called with an empty list or ";
      message = message + "invalid position.";
      throw(PrecondViolatedExcept(message));
   }  // end if
}  // end getEntry
```

Method getEntry

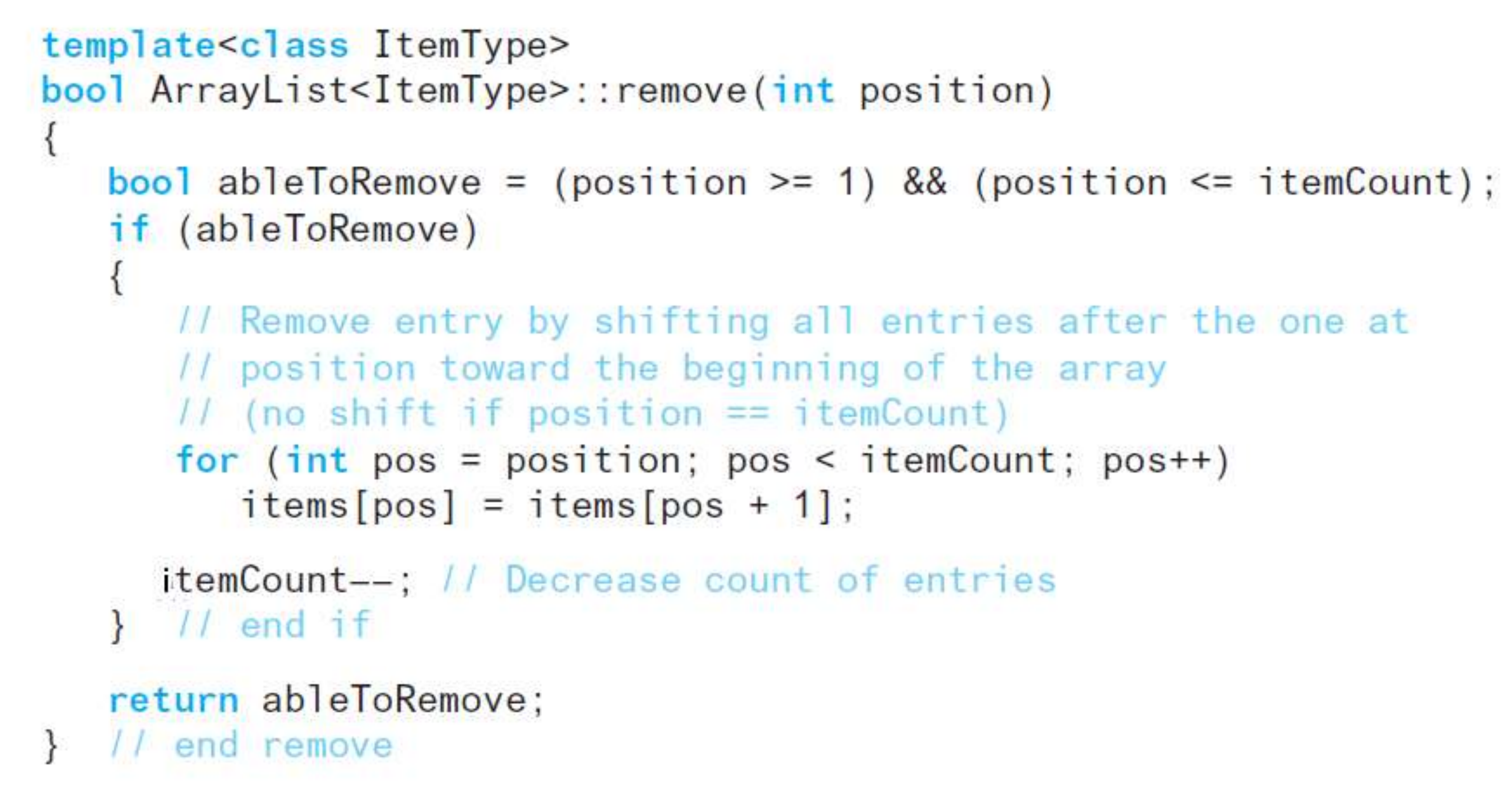
The Implementation File



```
template<class ItemType>
ItemType ArrayList<ItemType>::replace(int position, const ItemType& newEntry)
                              throw(PrecondViolatedExcept)
{
   // Enforce precondition
   bool ableToSet = (position >= 1) && (position <= itemCount);
   if (ableToSet)
   {
      ItemType oldEntry = items[position];
      items[position] = newEntry;
      return oldEntry;
   }
   else
   {
      std::string message = "replace() called with an empty list or ";
      message = message + "invalid position.";
      throw(PrecondViolatedExcept(message));
   }  // end if
}  // end replace
```

Method replace

The Implementation File



```
template<class ItemType>
bool ArrayList<ItemType>::remove(int position)
{
   bool ableToRemove = (position >= 1) && (position <= itemCount);
   if (ableToRemove)
   {
      // Remove entry by shifting all entries after the one at
      // position toward the beginning of the array
      // (no shift if position == itemCount)
      for (int pos = position; pos < itemCount; pos++)
         items[pos] = items[pos + 1];

      itemCount--; // Decrease count of entries
   }  // end if

   return ableToRemove;
}  // end remove
```

Method remove

The Implementation File

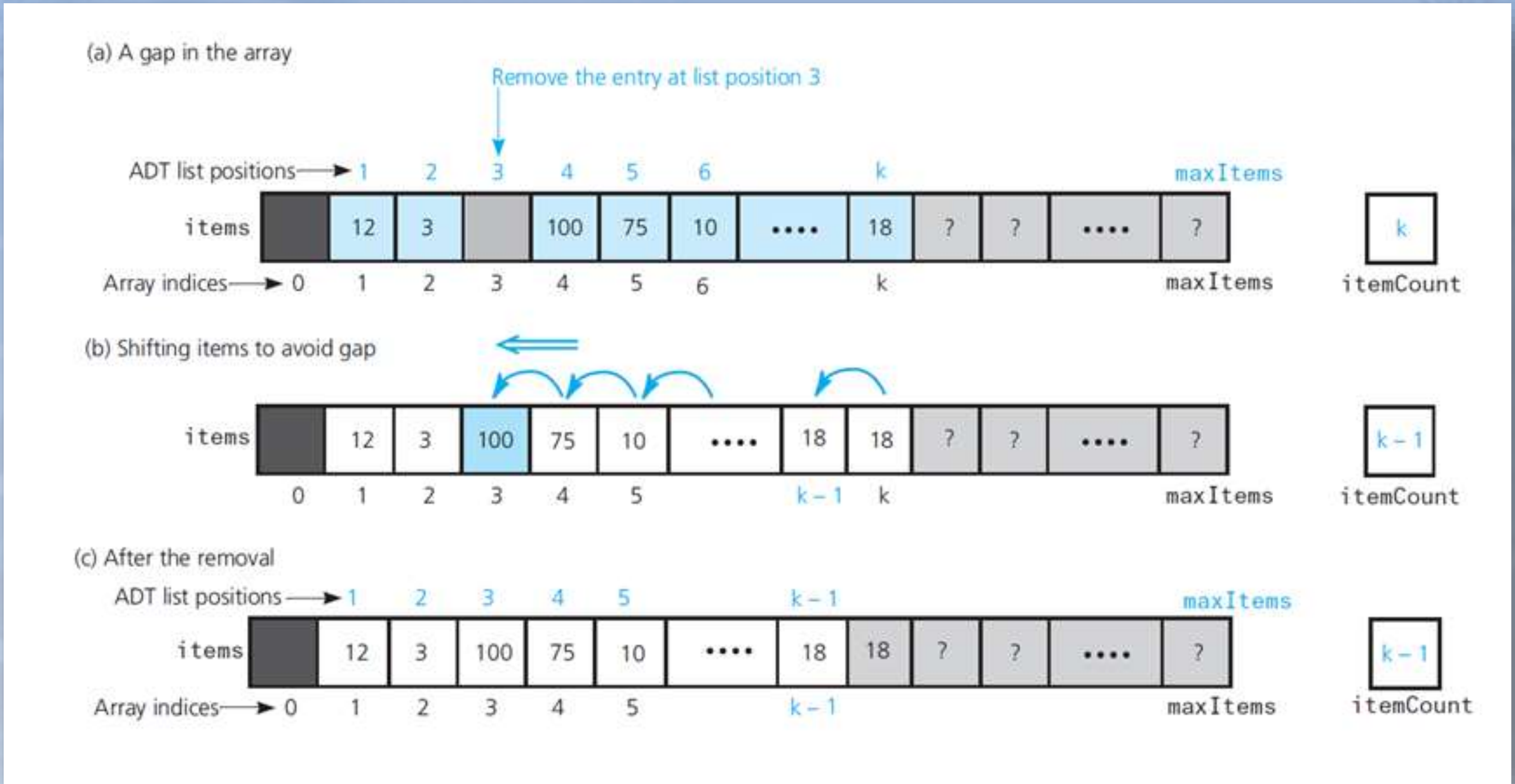
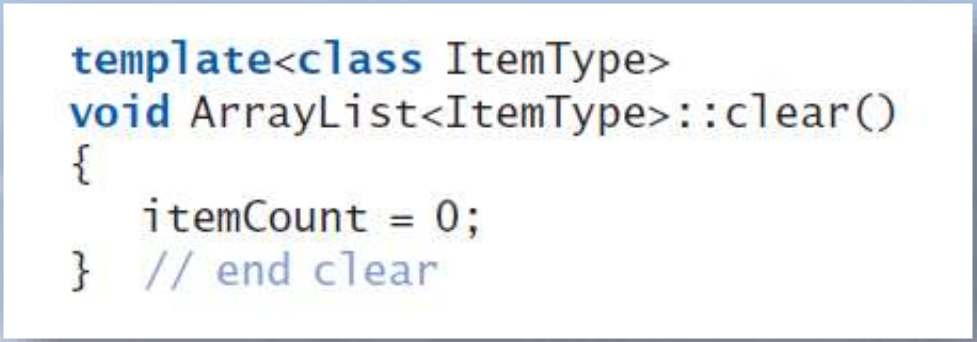


FIGURE 9-3 Shifting items to remove an entry

The Implementation File



```
template<class ItemType>
void ArrayList<ItemType>::clear()
{
   itemCount = 0;
}  // end clear
```

Method clear

Link-Based Implementation of the ADT List

- We can use C++ pointers instead of an array to implement ADT list
  - Link-based implementation does not shift items during insertion and removal operations
  - We need to represent items in the list and its length

Link-Based Implementation of the ADT List

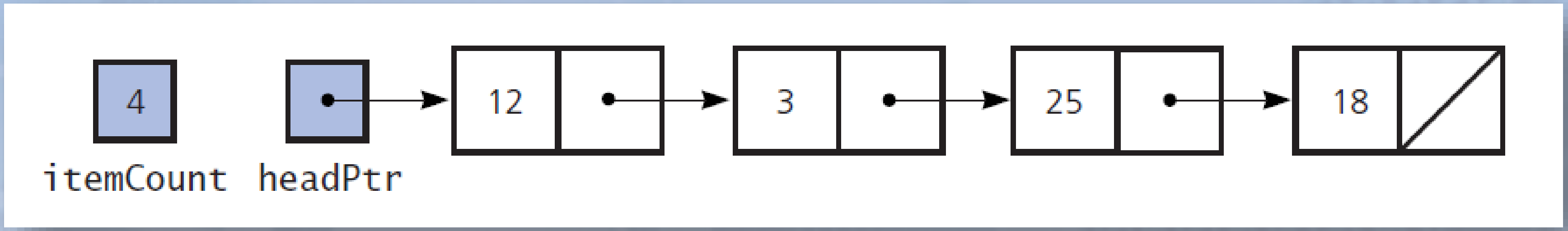
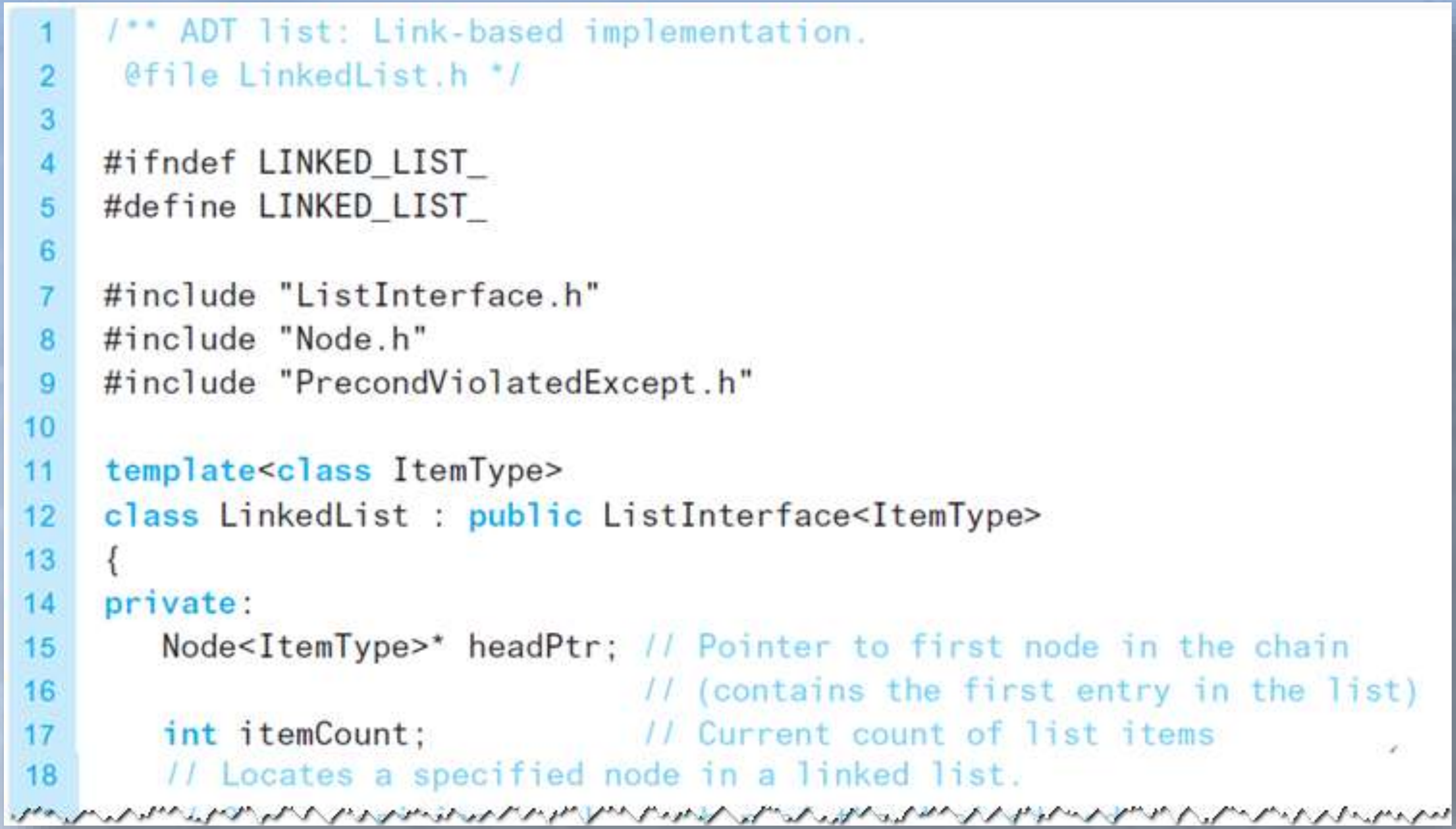


FIGURE 9-4 A link-based implementation of the ADT list

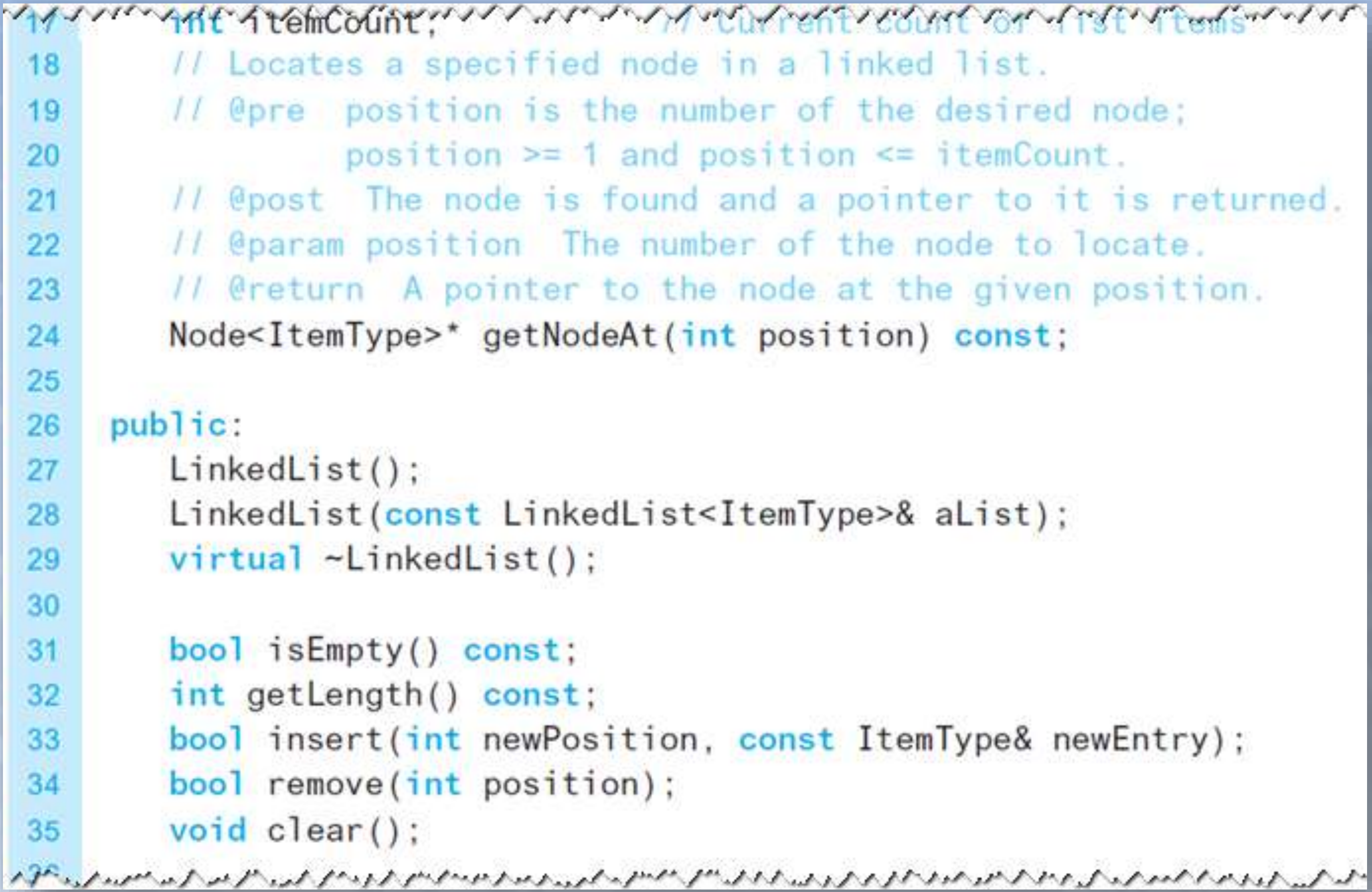
The Header File



```
/** ADT list: Link-based implementation.
 @file LinkedList.h */

#ifndef LINKED_LIST_
#define LINKED_LIST_

#include "ListInterface.h"
#include "Node.h"
#include "PrecondViolatedExcept.h"

template<class ItemType>
class LinkedList : public ListInterface<ItemType>
{
private:
   Node<ItemType>* headPtr; // Pointer to first node in the chain
                            // (contains the first entry in the list)
   int itemCount;           // Current count of list items
   // Locates a specified node in a linked list.
```

LISTING 9-2 The header file for the class LinkedList

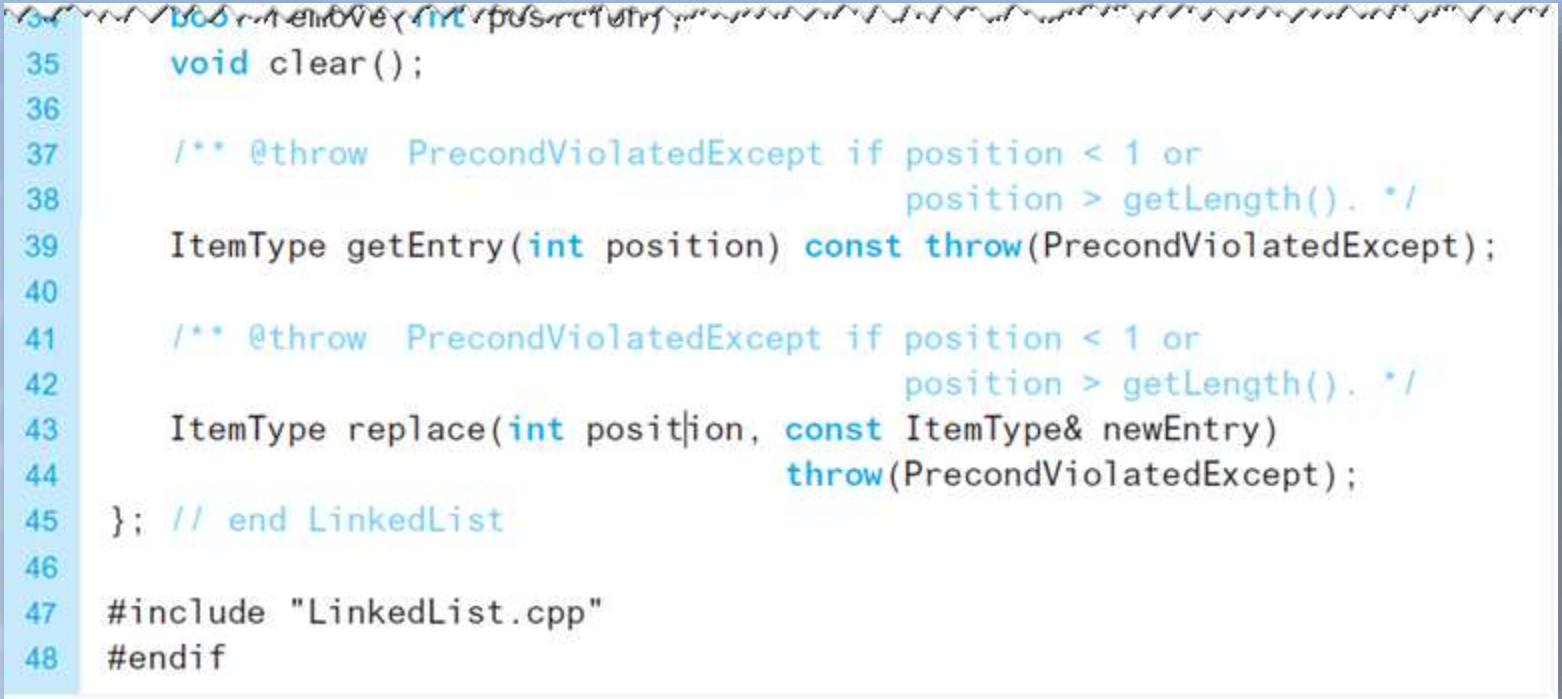
The Header File



```
   int itemCount;             // Current count of list items
   // Locates a specified node in a linked list.
   // @pre  position is the number of the desired node;
            position >= 1 and position <= itemCount.
   // @post  The node is found and a pointer to it is returned.
   // @param position  The number of the node to locate.
   // @return  A pointer to the node at the given position.
   Node<ItemType>* getNodeAt(int position) const;

public:
   LinkedList();
   LinkedList(const LinkedList<ItemType>& aList);
   virtual ~LinkedList();

   bool isEmpty() const;
   int getLength() const;
   bool insert(int newPosition, const ItemType& newEntry);
   bool remove(int position);
   void clear();
```

LISTING 9-2 The header file for the class LinkedList

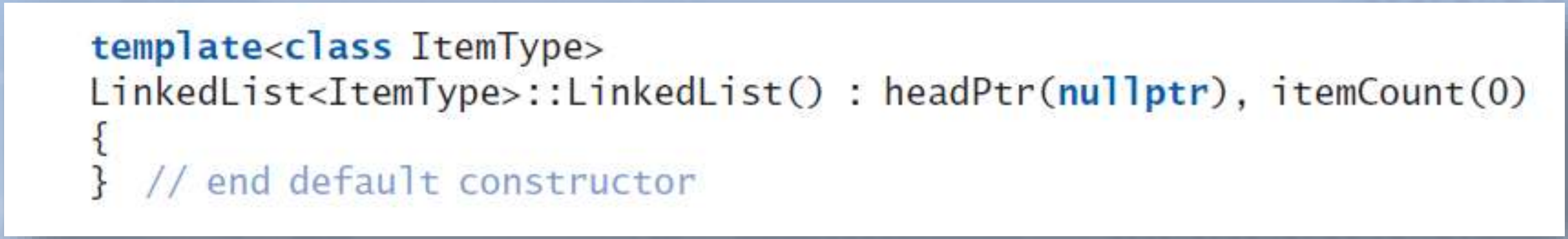
The Header File



```
   bool remove(int position);
   void clear();

   /** @throw  PrecondViolatedExcept if position < 1 or
                                        position > getLength(). */
   ItemType getEntry(int position) const throw(PrecondViolatedExcept);

   /** @throw  PrecondViolatedExcept if position < 1 or
                                        position > getLength(). */
   ItemType replace(int position, const ItemType& newEntry)
                                  throw(PrecondViolatedExcept);
}; // end LinkedList

#include "LinkedList.cpp"
#endif
```

LISTING 9-2 The header file for the class LinkedList

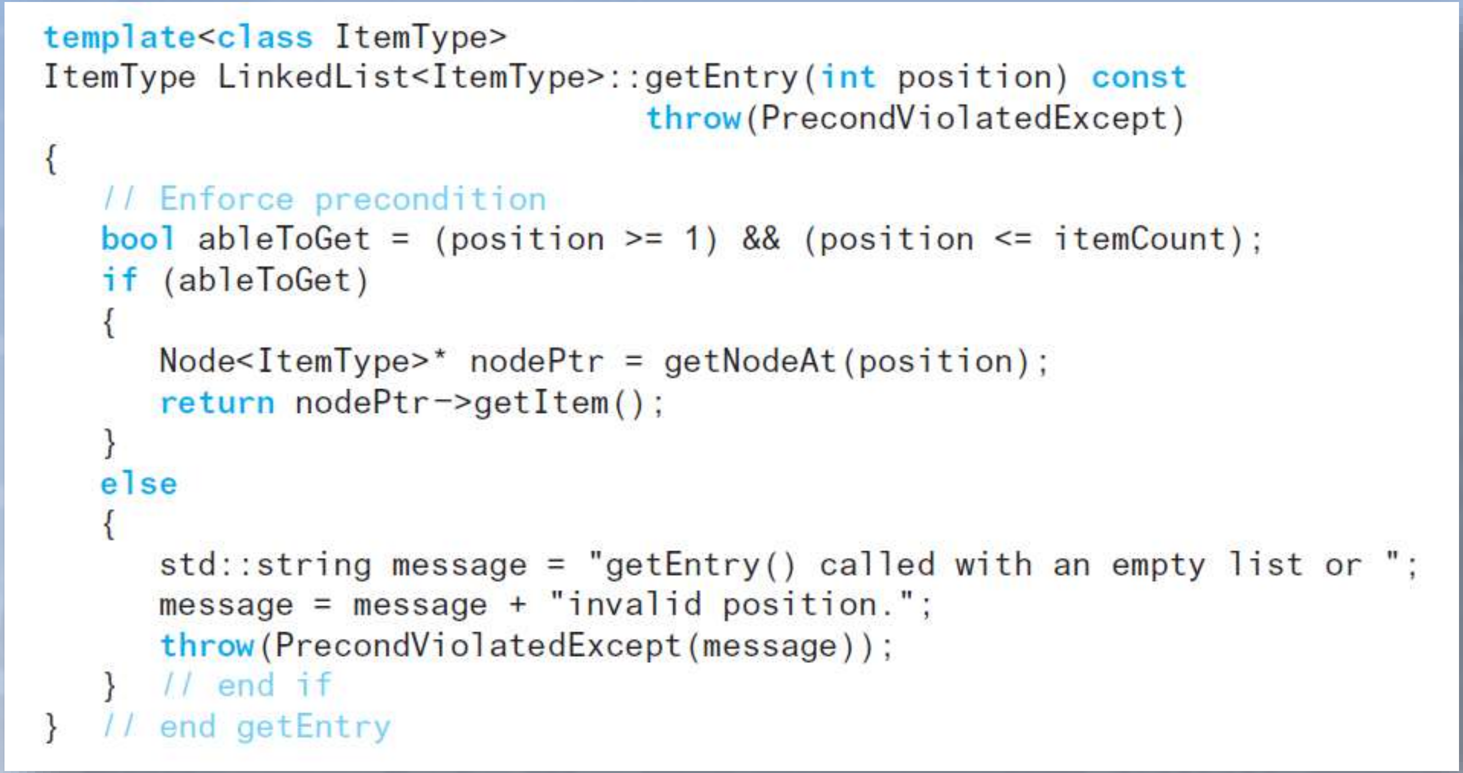
The Implementation File



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

Constructor

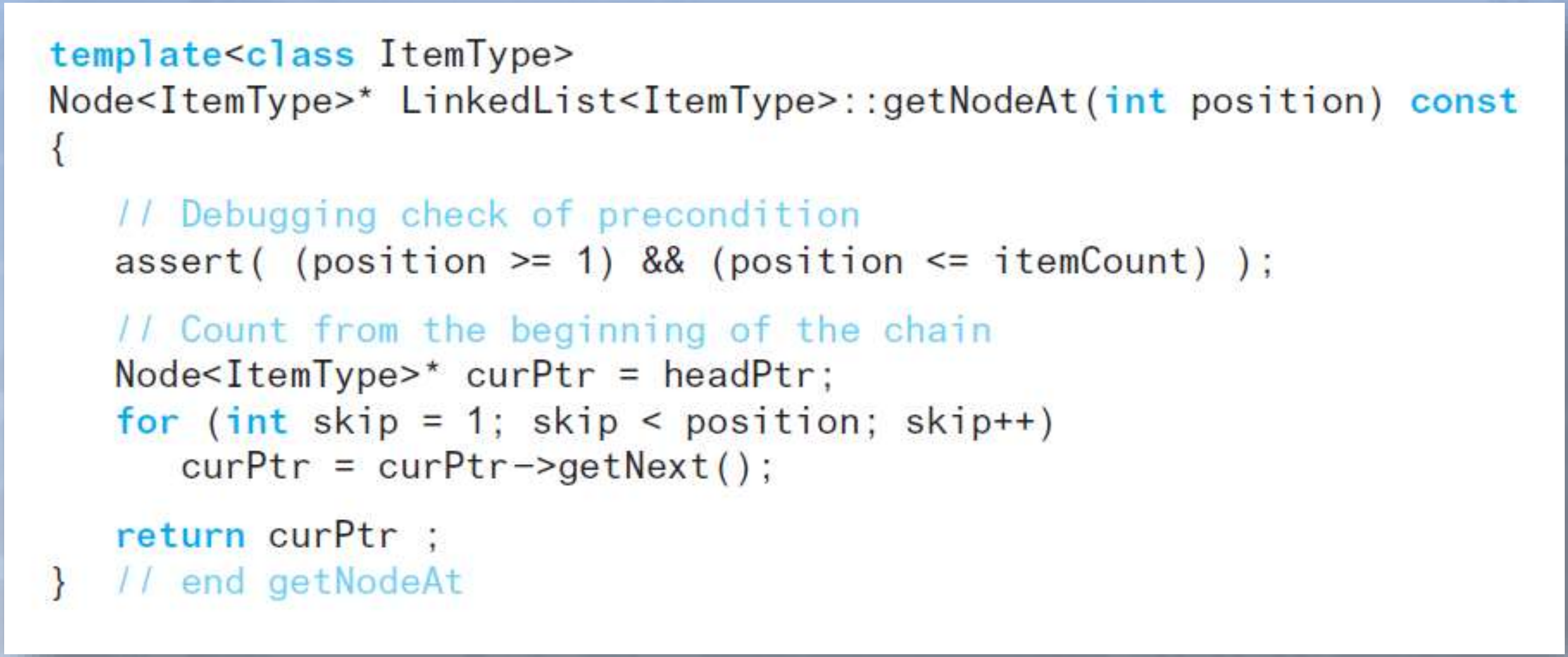
The Implementation File



```
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
```

Method getEntry

The Implementation File



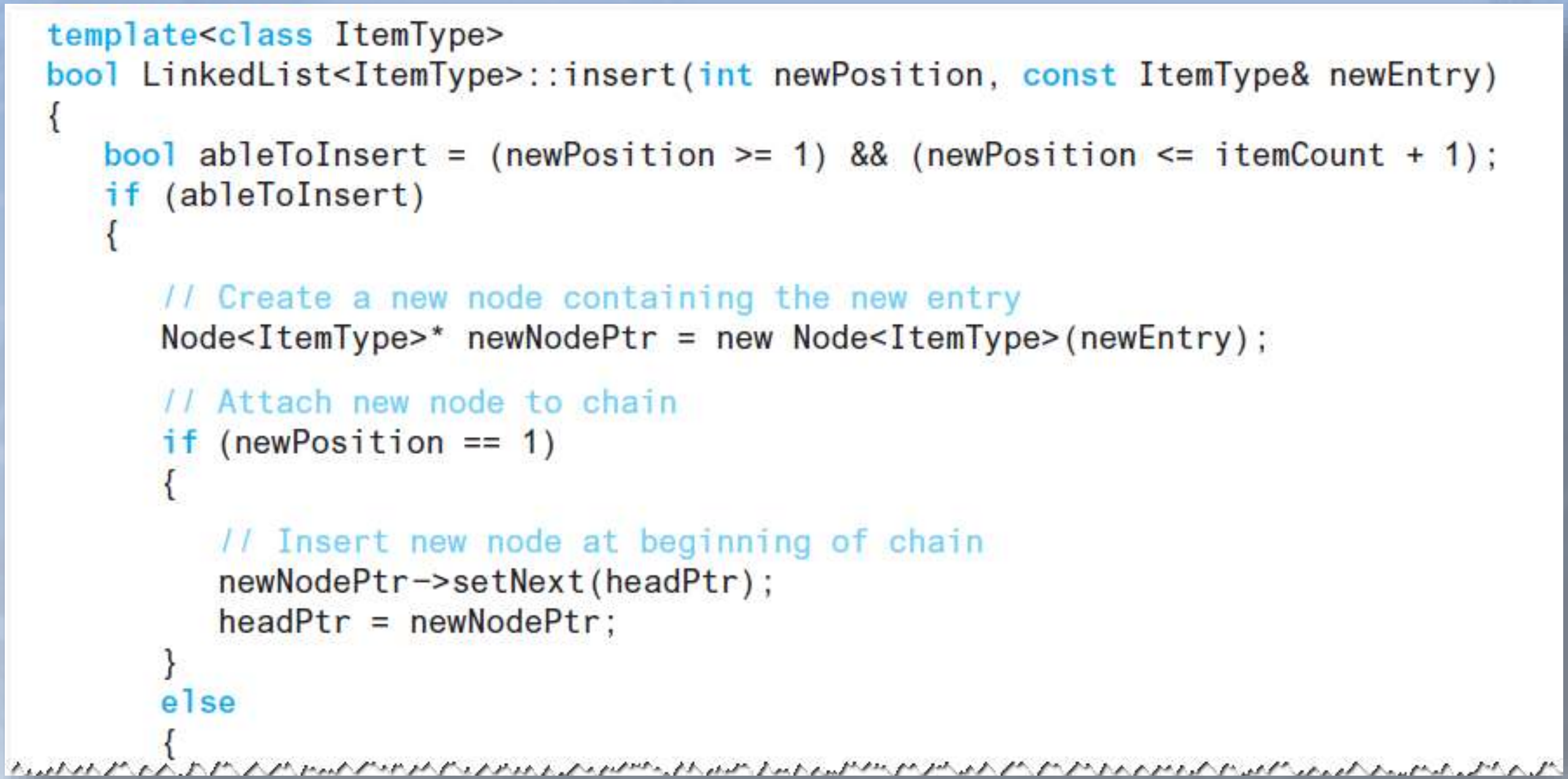
```
template<class ItemType>
Node<ItemType>* LinkedList<ItemType>::getNodeAt(int position) const
{
   // Debugging check of precondition
   assert( (position >= 1) && (position <= itemCount) );

   // 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
```

Method getNodeAt

The Implementation File

- Insertion process requires three high-level steps:
  1. Create a new node and store the new data in it.
  2. Determine the point of insertion.
  3. Connect the new node to the linked chain by changing pointers.

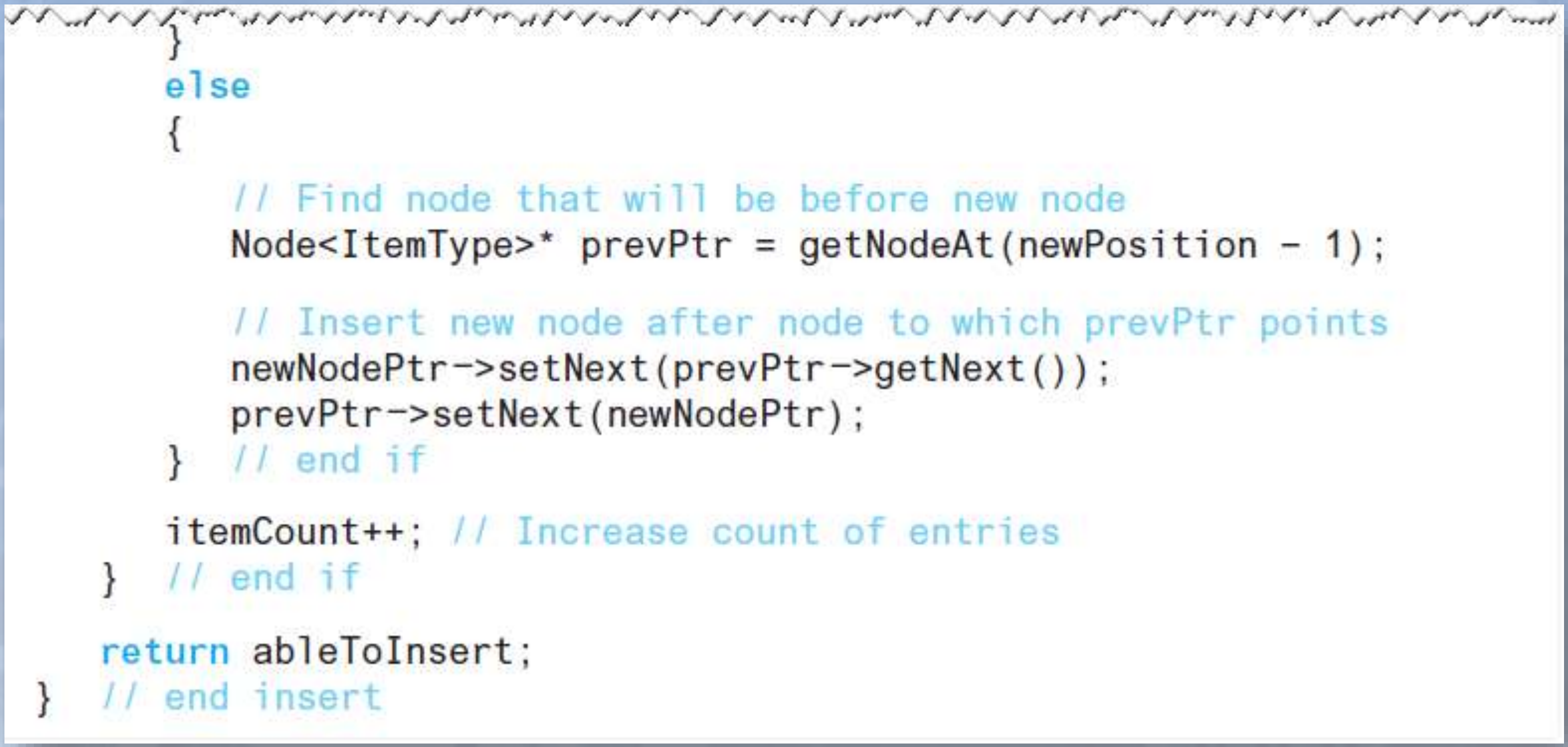
The Implementation File



```
template<class ItemType>
bool LinkedList<ItemType>::insert(int newPosition, const ItemType& newEntry)
{
   bool ableToInsert = (newPosition >= 1) && (newPosition <= itemCount + 1);
   if (ableToInsert)
   {
      // Create a new node containing the new entry
      Node<ItemType>* newNodePtr = new Node<ItemType>(newEntry);

      // Attach new node to chain
      if (newPosition == 1)
      {
         // Insert new node at beginning of chain
         newNodePtr->setNext(headPtr);
         headPtr = newNodePtr;
      }
      else
      {
```

Method insert

The Implementation File



```
      }
      else
      {
         // Find node that will be before new node
         Node<ItemType>* prevPtr = getNodeAt(newPosition - 1);

         // Insert new node after node to which prevPtr points
         newNodePtr->setNext(prevPtr->getNext());
         prevPtr->setNext(newNodePtr);
      }  // end if

      itemCount++; // Increase count of entries
   }  // end if

   return ableToInsert;
}  // end insert
```

Method insert

The Implementation File

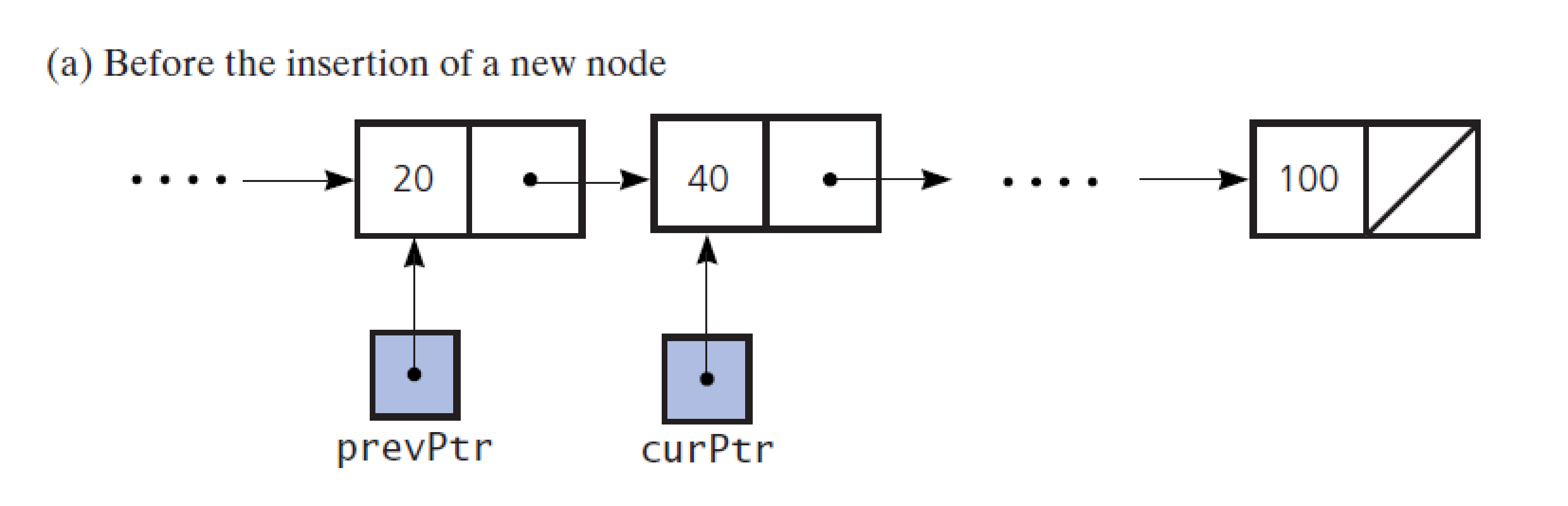


FIGURE 9-5 Inserting a new node between existing nodes of a linked chain

The Implementation File

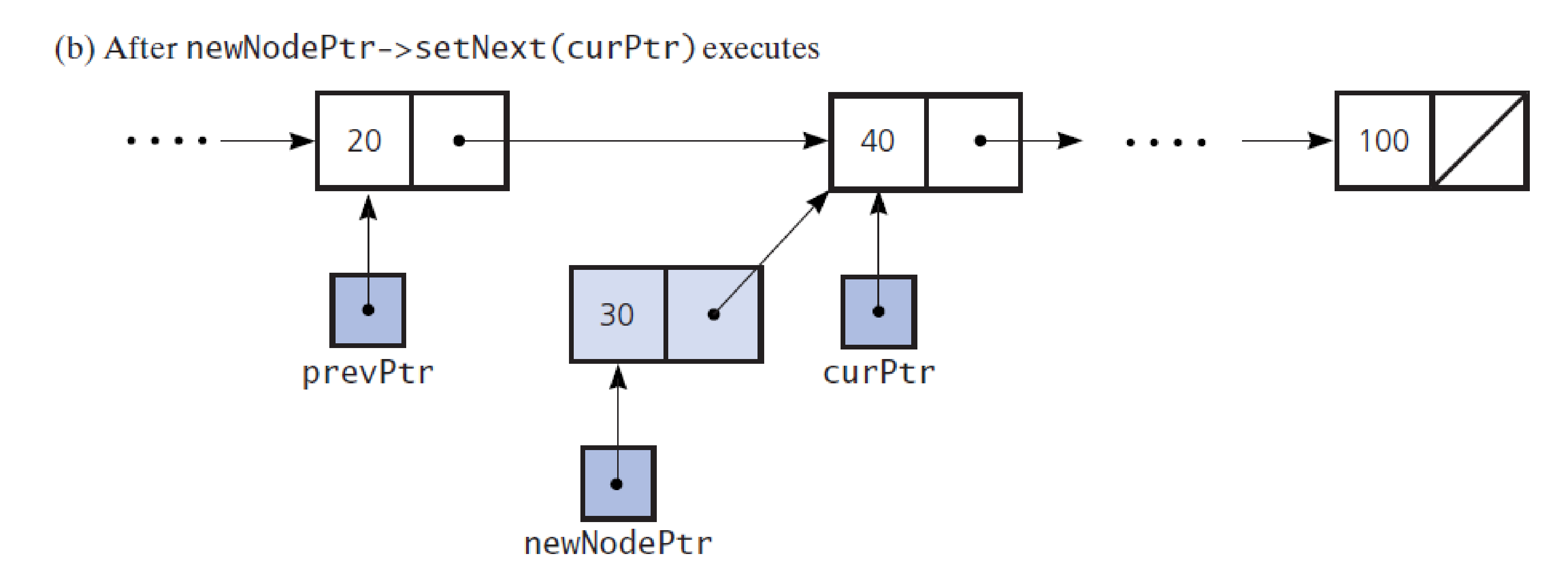


FIGURE 9-5 Inserting a new node between existing nodes of a linked chain

The Implementation File

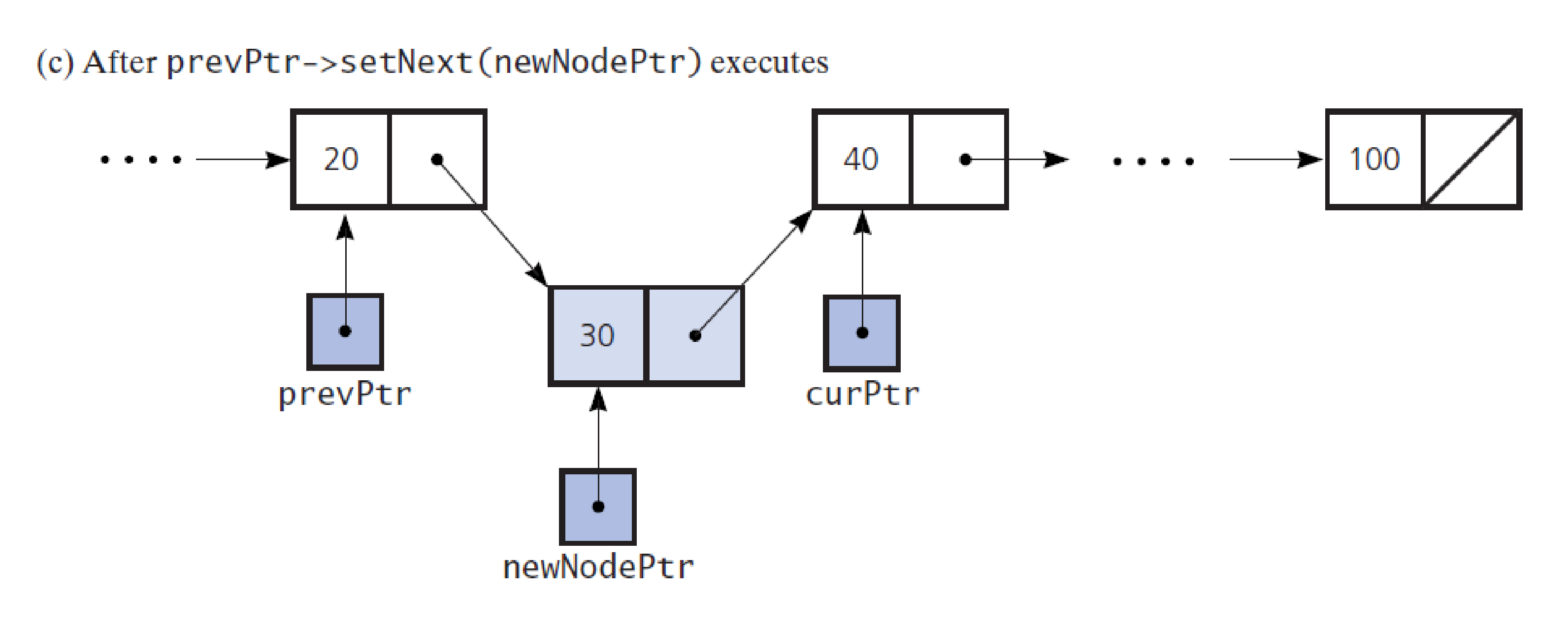


FIGURE 9-5 Inserting a new node between existing nodes of a linked chain

The Implementation File

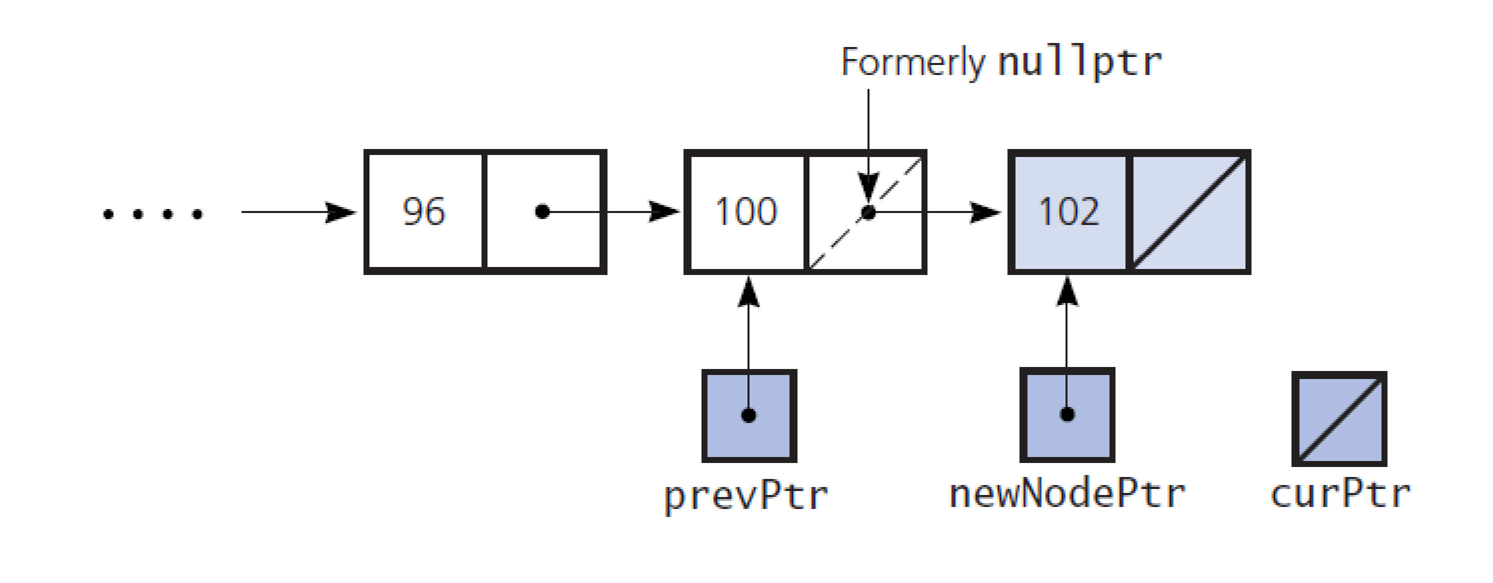


FIGURE 9-6 Inserting a new node at the end of a chain of linked nodes

The Implementation File

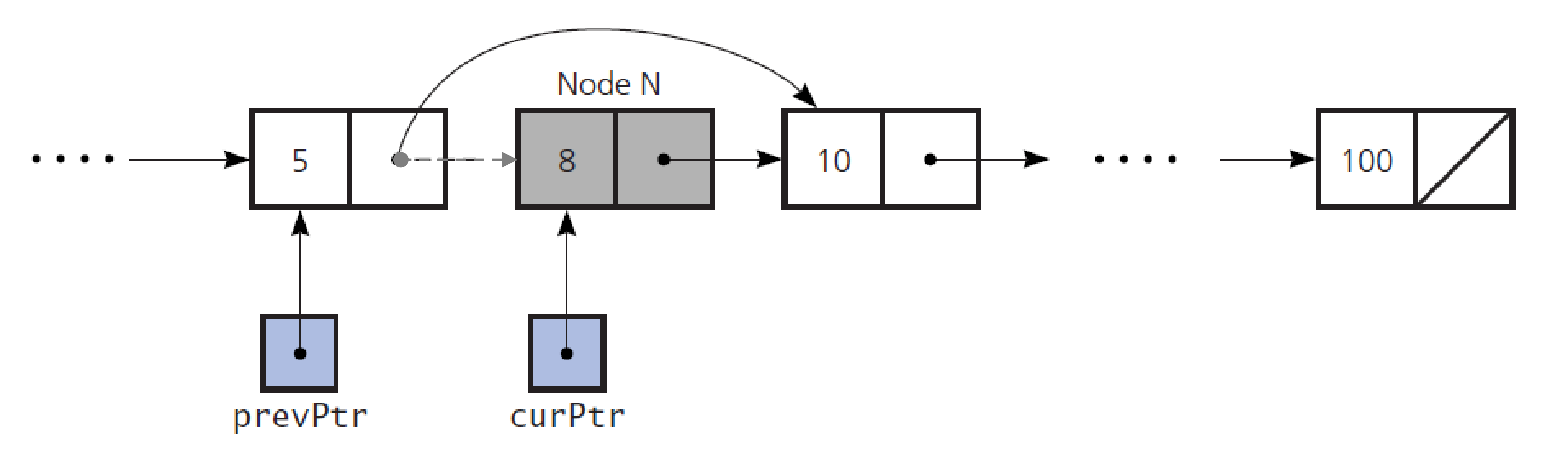


FIGURE 9-7 Removing a node from a chain

The Implementation File

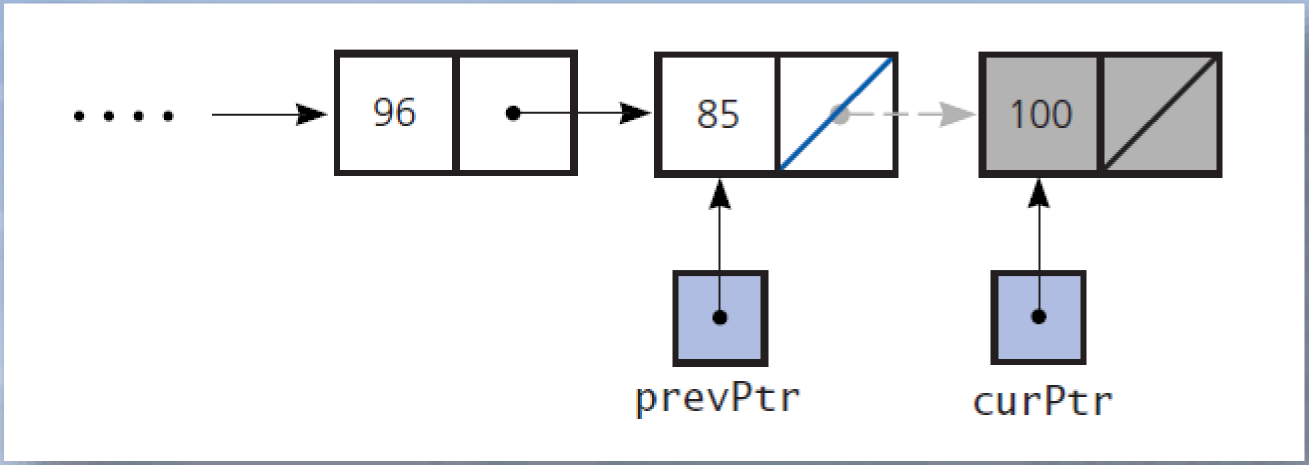
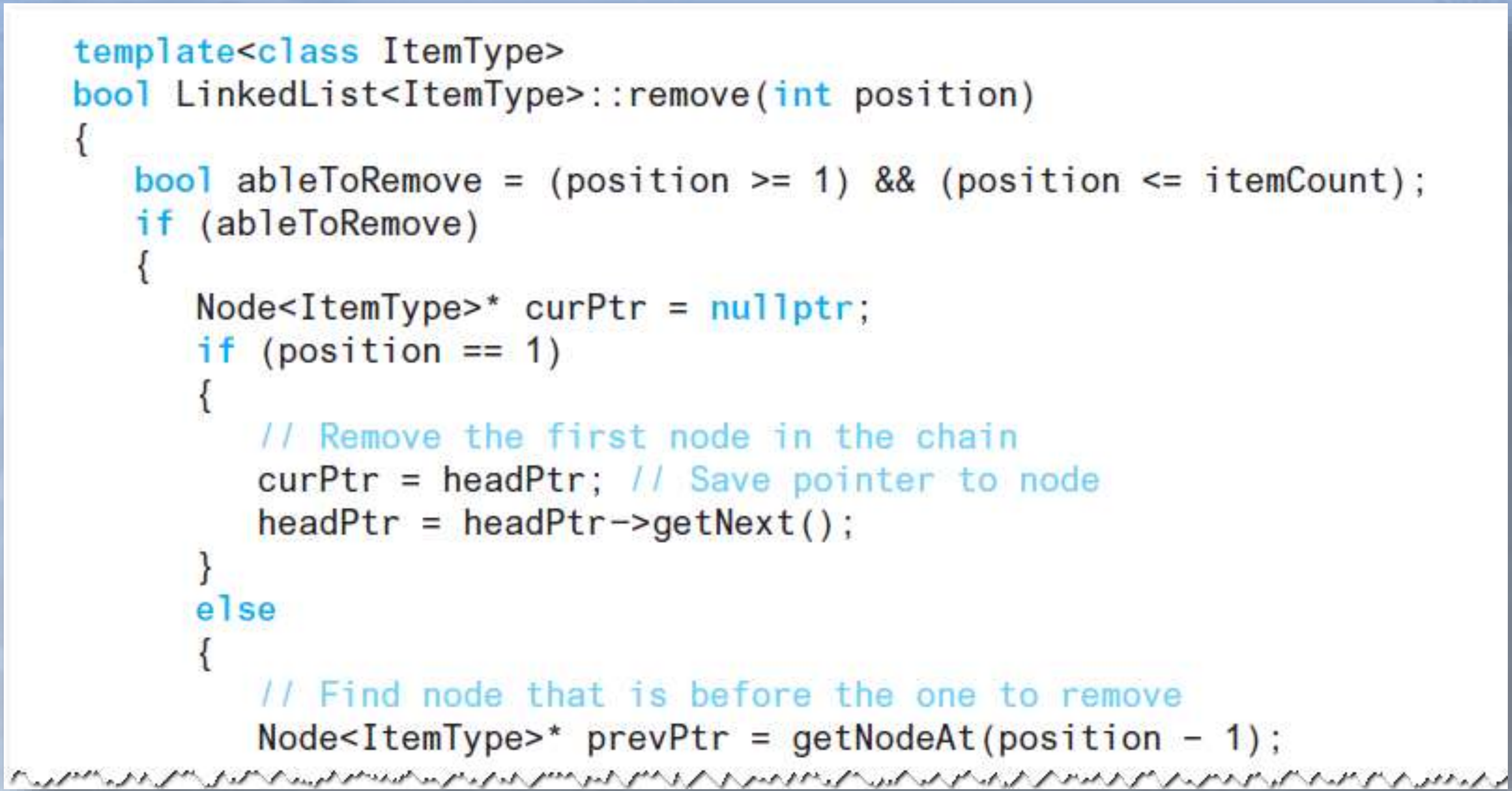


FIGURE 9-8 Removing the last node

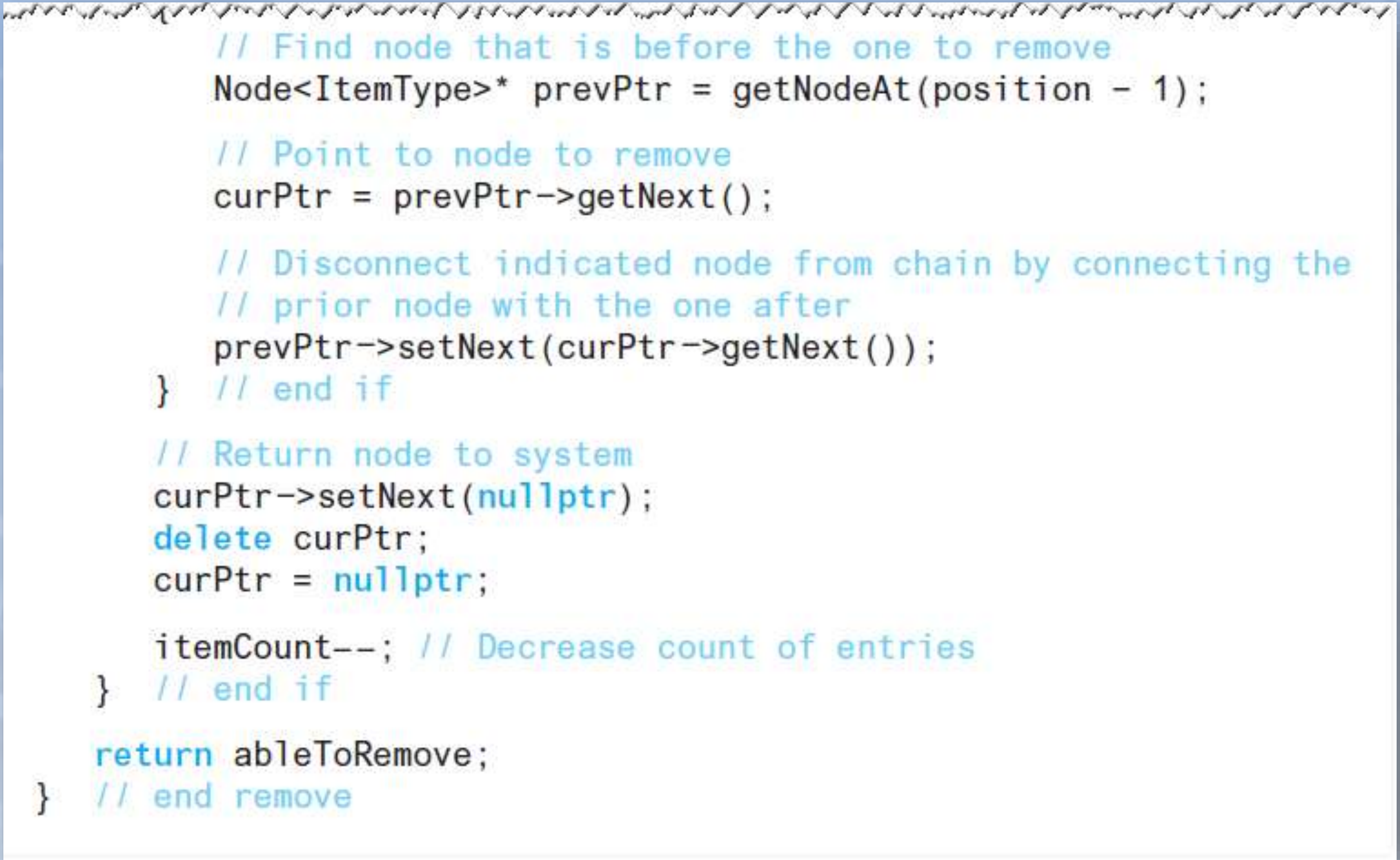
The Implementation File



```
template<class ItemType>
bool LinkedList<ItemType>::remove(int position)
{
   bool ableToRemove = (position >= 1) && (position <= itemCount);
   if (ableToRemove)
   {
      Node<ItemType>* curPtr = nullptr;
      if (position == 1)
      {
         // Remove the first node in the chain
         curPtr = headPtr; // Save pointer to node
         headPtr = headPtr->getNext();
      }
      else
      {
         // Find node that is before the one to remove
         Node<ItemType>* prevPtr = getNodeAt(position - 1);
```

Method remove

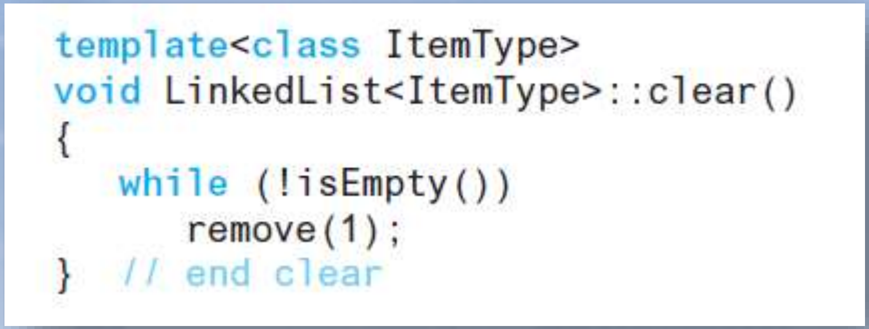
The Implementation File



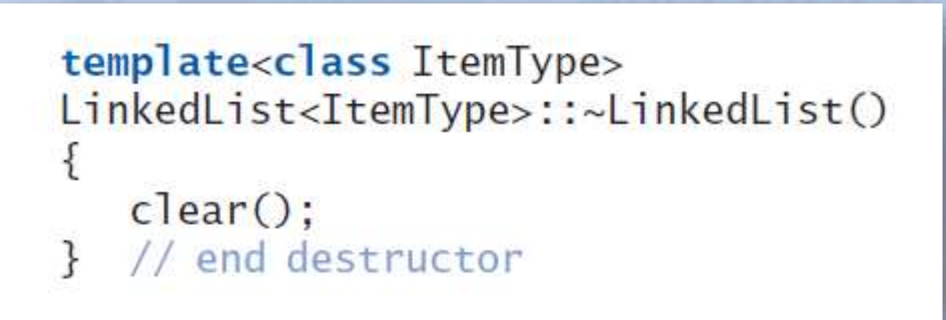
```
         // Find node that is before the one to remove
         Node<ItemType>* prevPtr = getNodeAt(position - 1);

         // Point to node to remove
         curPtr = prevPtr->getNext();

         // Disconnect indicated node from chain by connecting the
         // prior node with the one after
         prevPtr->setNext(curPtr->getNext());
      }  // end if

      // Return node to system
      curPtr->setNext(nullptr);
      delete curPtr;
      curPtr = nullptr;

      itemCount--; // Decrease count of entries
   }  // end if

   return ableToRemove;
}  // end remove
```

Method remove

The Implementation File



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

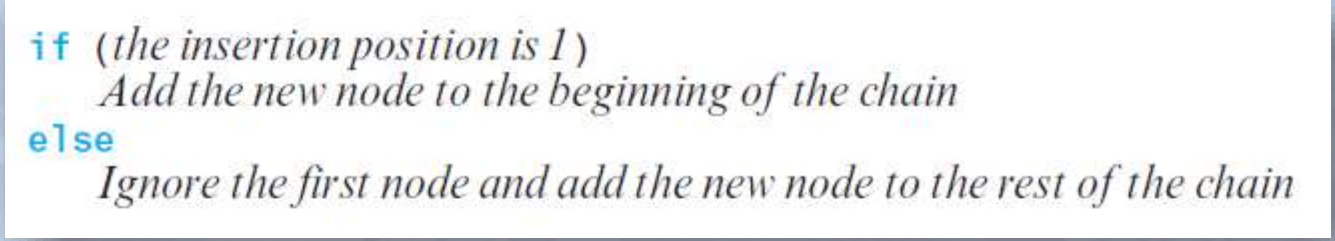


```
template<class ItemType>
LinkedList<ItemType>::~LinkedList()
{
   clear();
}  // end destructor
```

Method clear and the destructor

Using Recursion in LinkedList Methods

- Possible to process a linked chain by
  - Processing its first node and
  - Then the rest of the chain recursively
- Logic used to add a node



```
if (the insertion position is 1)
    Add the new node to the beginning of the chain
else
    Ignore the first node and add the new node to the rest of the chain
```

Using Recursion in LinkedList Methods

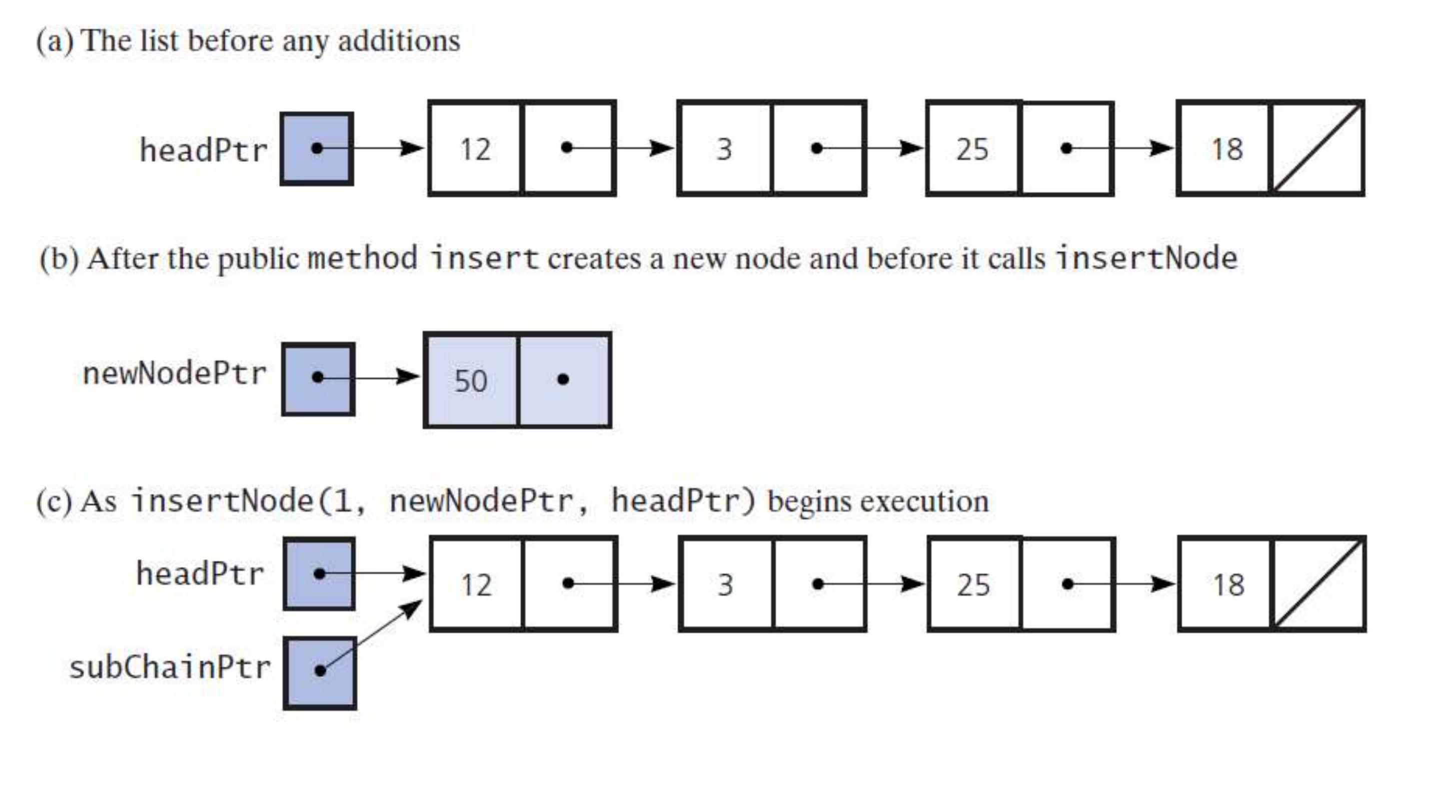

(a) The list before any additions
headPtr
12
3
25
18
(b) After the public method insert creates a new node and before it calls insertNode
newNodePtr
50
(c) As insertNode(1, newNodePtr, headPtr) begins execution
headPtr
subChainPtr
12
3
25
18

FIGURE 9-9 Recursively adding a node at the beginning of a chain

Using Recursion in LinkedList Methods

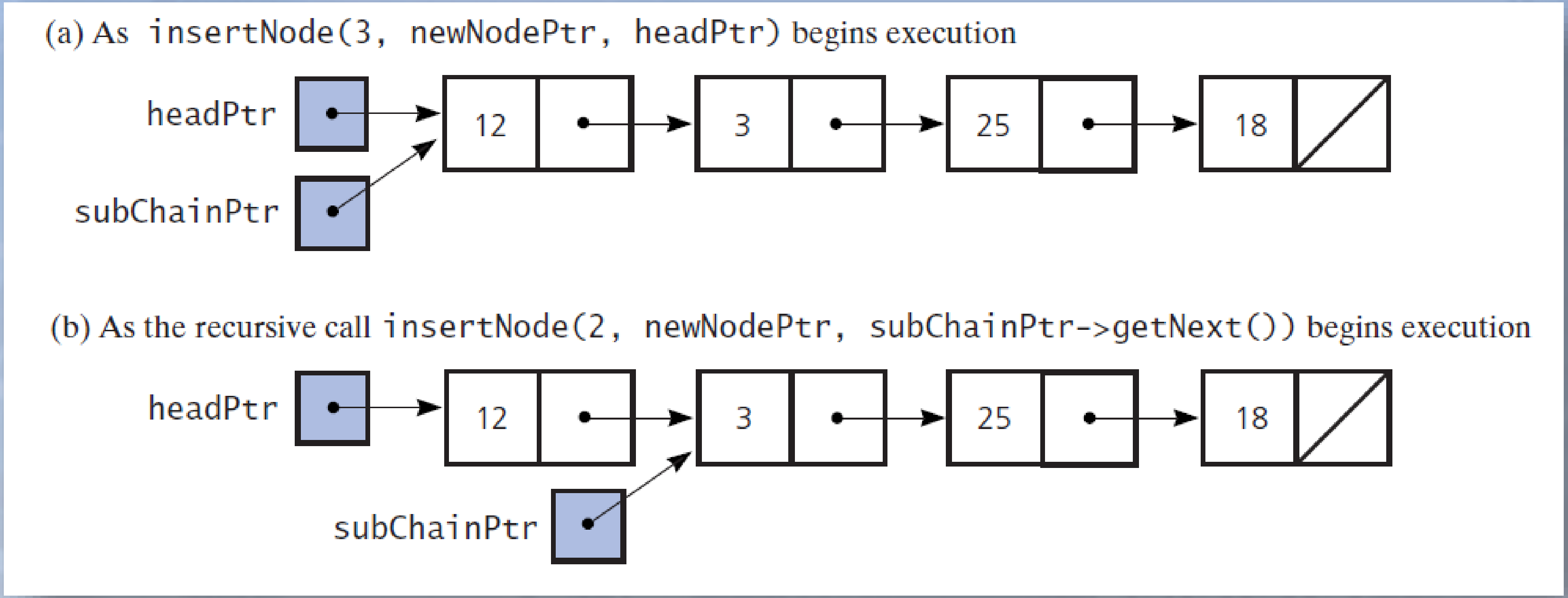

(a) As insertNode(3, newNodePtr, headPtr) begins execution
headPtr
subChainPtr
12
3
25
18
(b) As the recursive call insertNode(2, newNodePtr, subChainPtr->getNext()) begins execution
headPtr
12
3
25
18
subChainPtr

FIGURE 9-10 Recursively adding a node between existing nodes in a chain

Using Recursion in LinkedList Methods

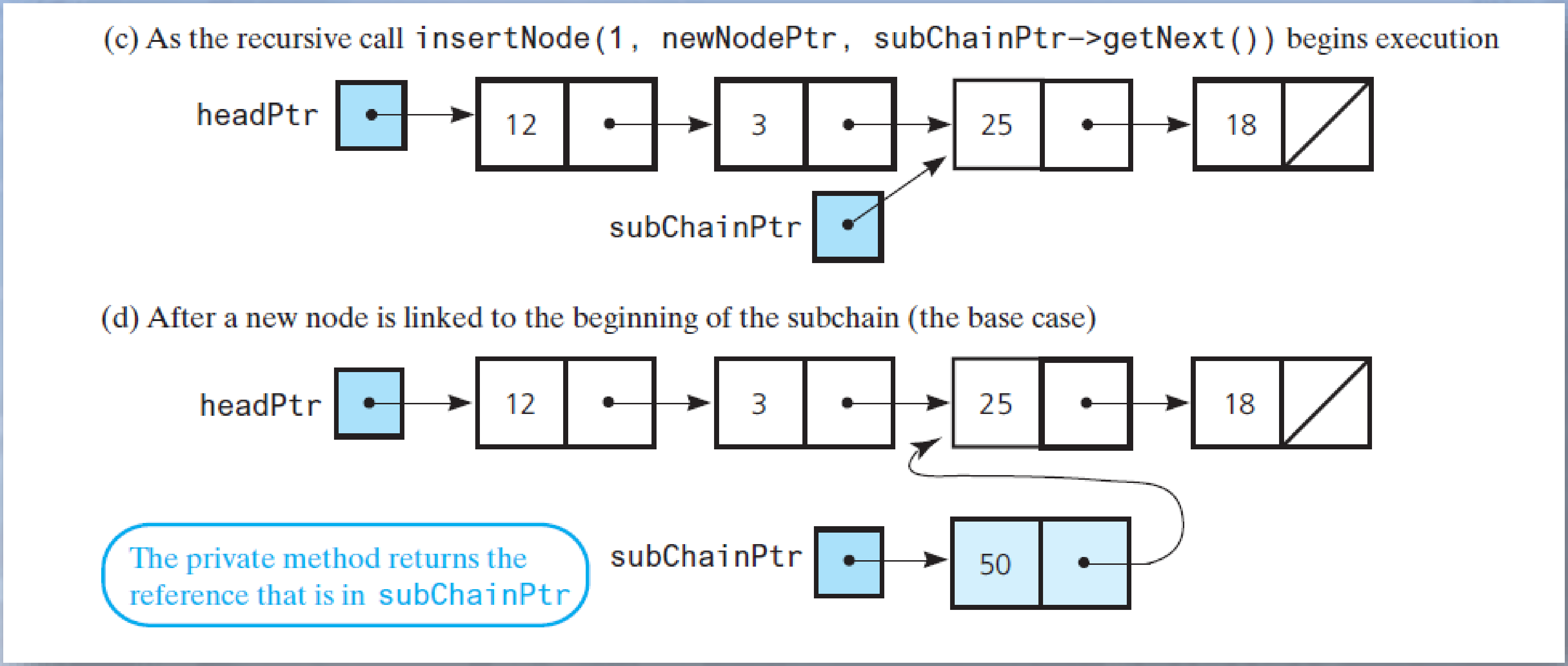
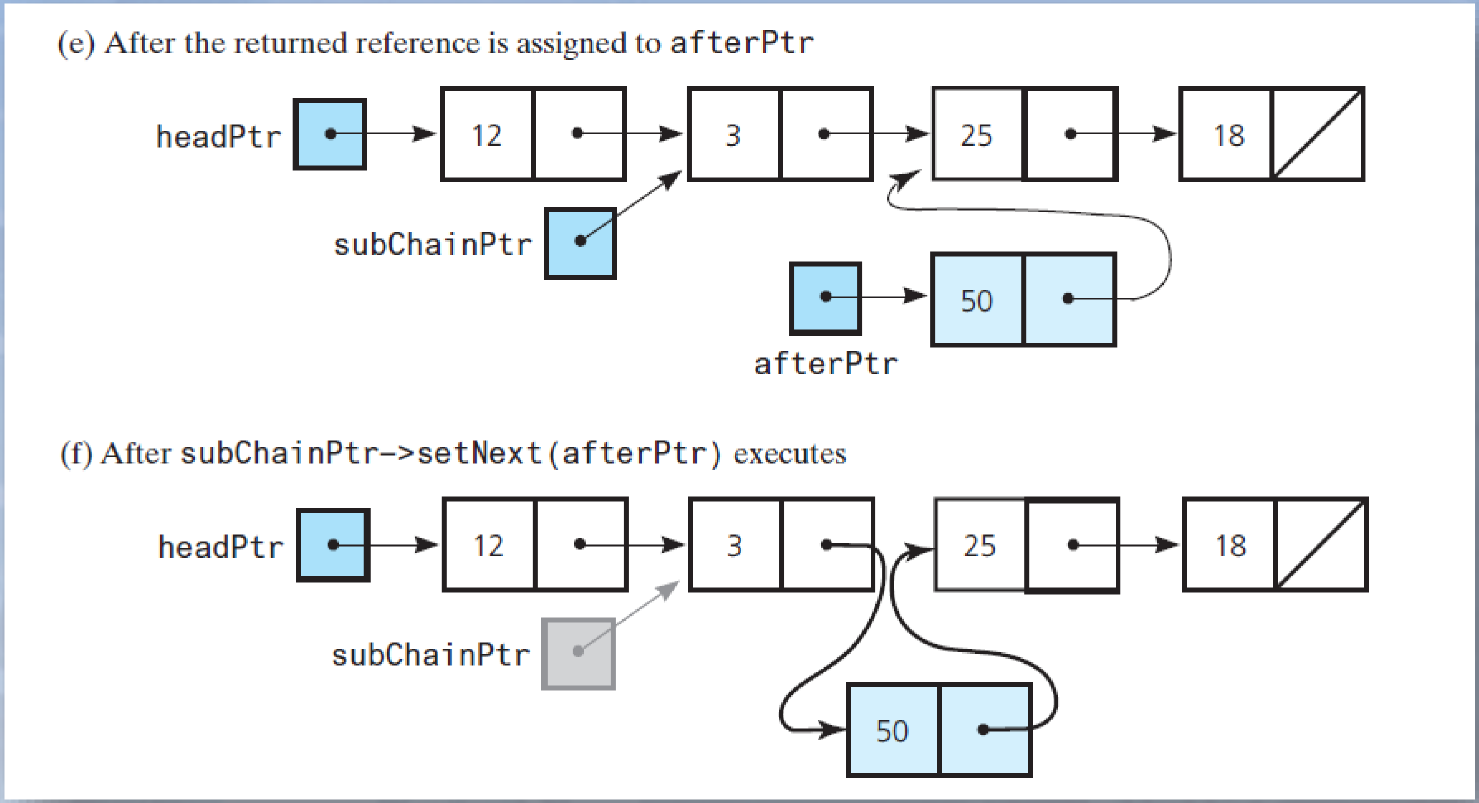

(c) As the recursive call insertNode(1, newNodePtr, subChainPtr->getNext()) begins execution
headPtr
12
3
25
18
subChainPtr
(d) After a new node is linked to the beginning of the subchain (the base case)
headPtr
12
3
25
18
subChainPtr
50
The private method returns the reference that is in subChainPtr

FIGURE 9-10 Recursively adding a node between existing nodes in a chain

Using Recursion in LinkedList Methods



(e) After the returned reference is assigned to afterPtr
headPtr
12
3
25
18
subChainPtr
50
afterPtr
(f) After subChainPtr->setNext(afterPtr) executes
headPtr
12
3
25
18
subChainPtr
50

FIGURE 9-10 Recursively adding a node between existing nodes in a chain

Comparing Implementations

- Time to access the *i*th node in a chain of linked nodes depends on *i*
- You can access array items directly with equal access time
- Insertions and removals with link-based implementation
  - Do not require shifting data
  - Do require a traversal

End

Chapter 9