List Implementations

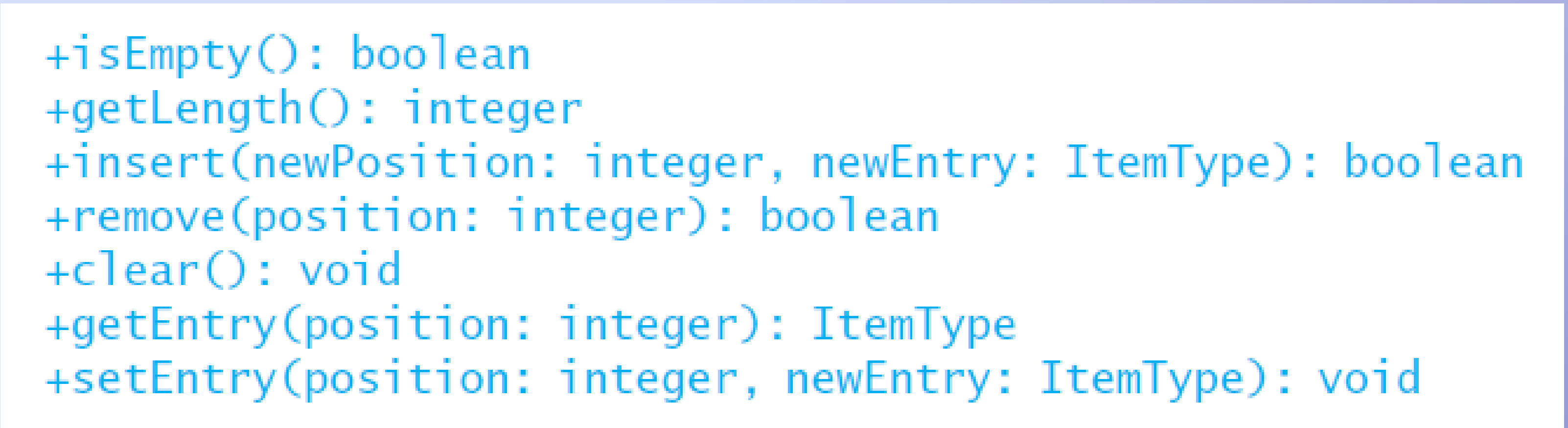
Chapter 9

Contents

- An Array-Based Implementation of the ADT List
- A Link-Based Implementation of the ADT List
- Comparing Implementations

Array-Based Implementation of ADT List

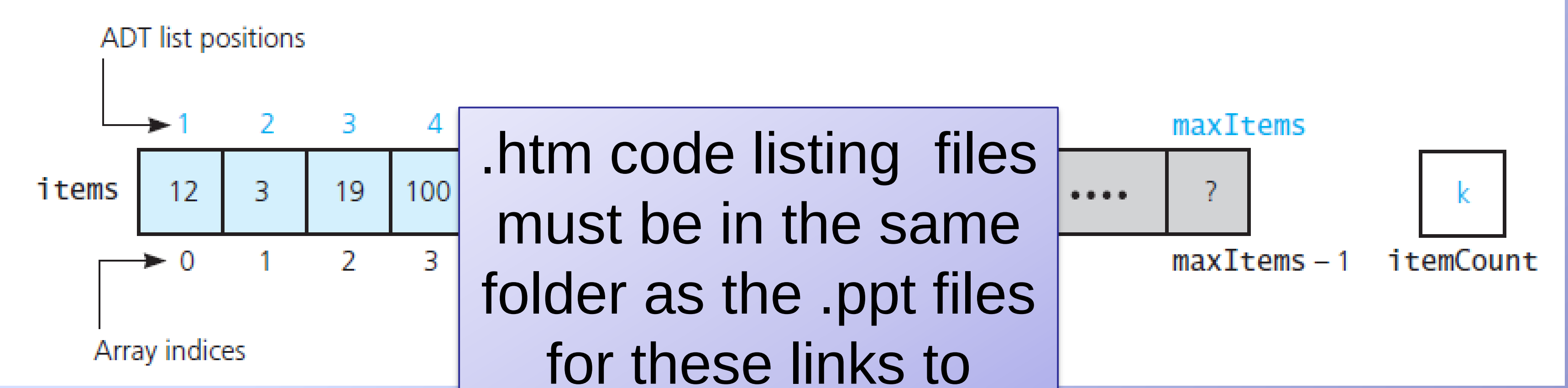
- Recall list operations in UML form



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

The Header File

- View header file for the class **`ArrayList`**, Listing 9-1

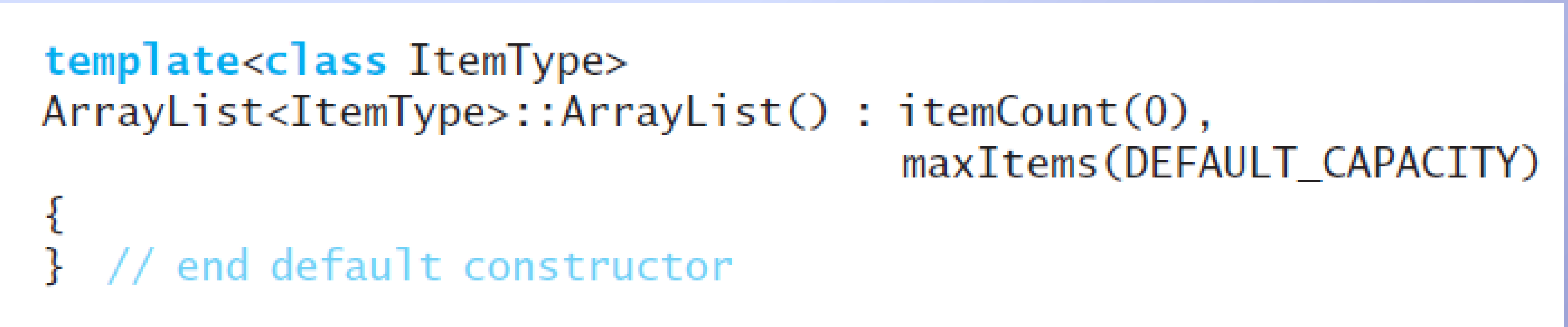


.htm code listing files must be in the same folder as the .ppt files for these links to work

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

The Implementation File

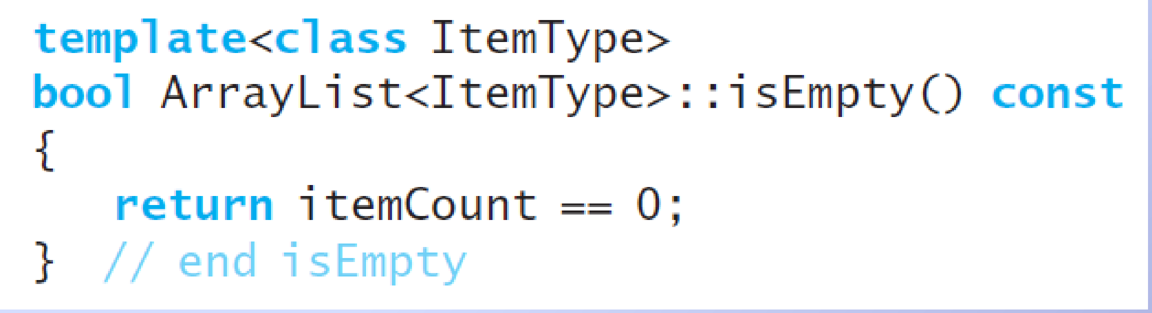
- Constructor



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

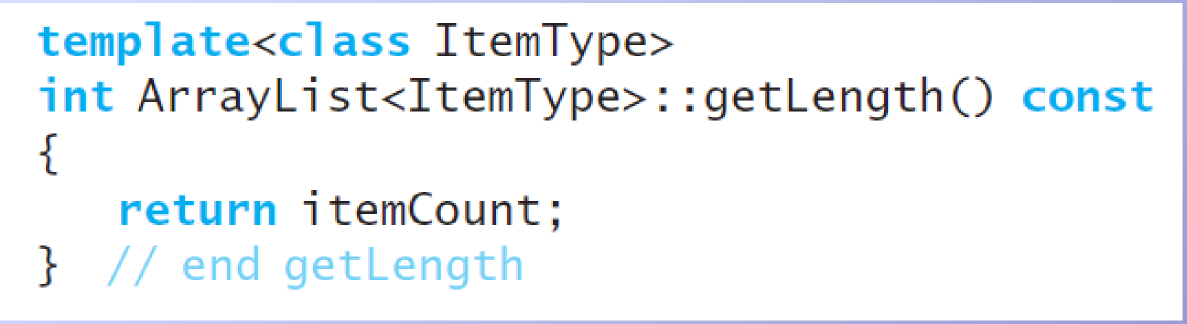
The Implementation File

- **`isEmpty`** tests whether **`itemCount`** is zero



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

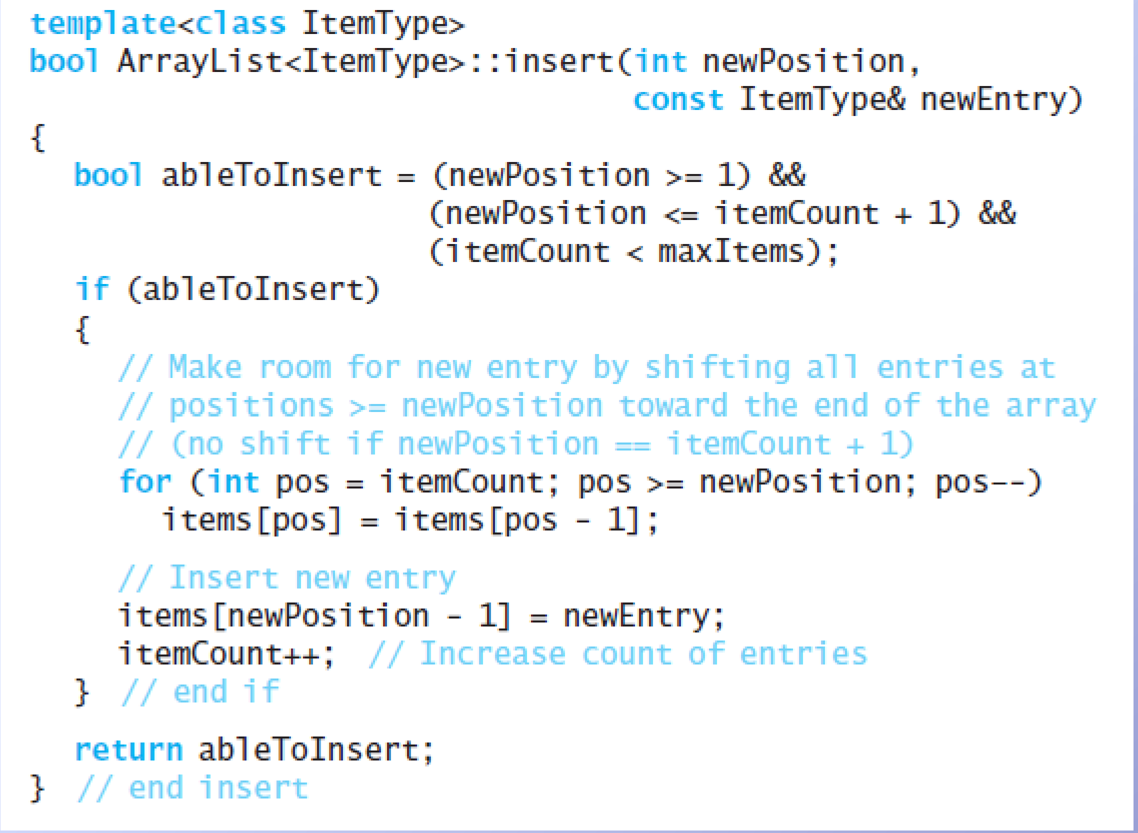
- **`getLength`** simply returns the value of **`itemCount`** :



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

The Implementation File

- Definition of the method **`insert`**

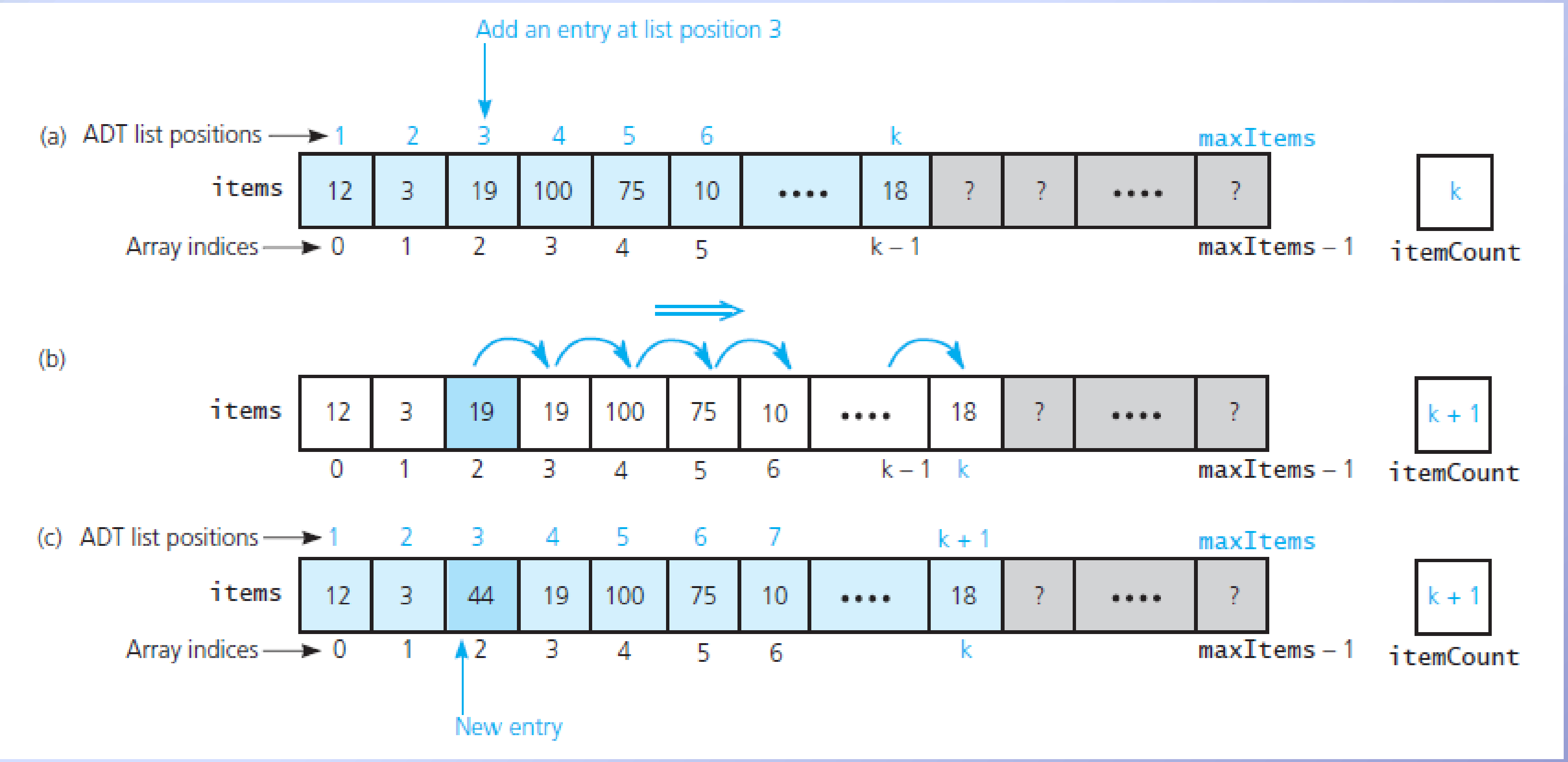


```
template<class ItemType>
bool ArrayList<ItemType>::insert(int newPosition,
                                 const ItemType& newEntry)
{
   bool ableToInsert = (newPosition >= 1) &&
                       (newPosition <= itemCount + 1) &&
                       (itemCount < maxItems);
   if (ableToInsert)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

      // Insert new entry
      items[newPosition - 1] = newEntry;
      itemCount++;  // Increase count of entries
   }  // end if

   return ableToInsert;
}  // end insert
```

The Implementation File



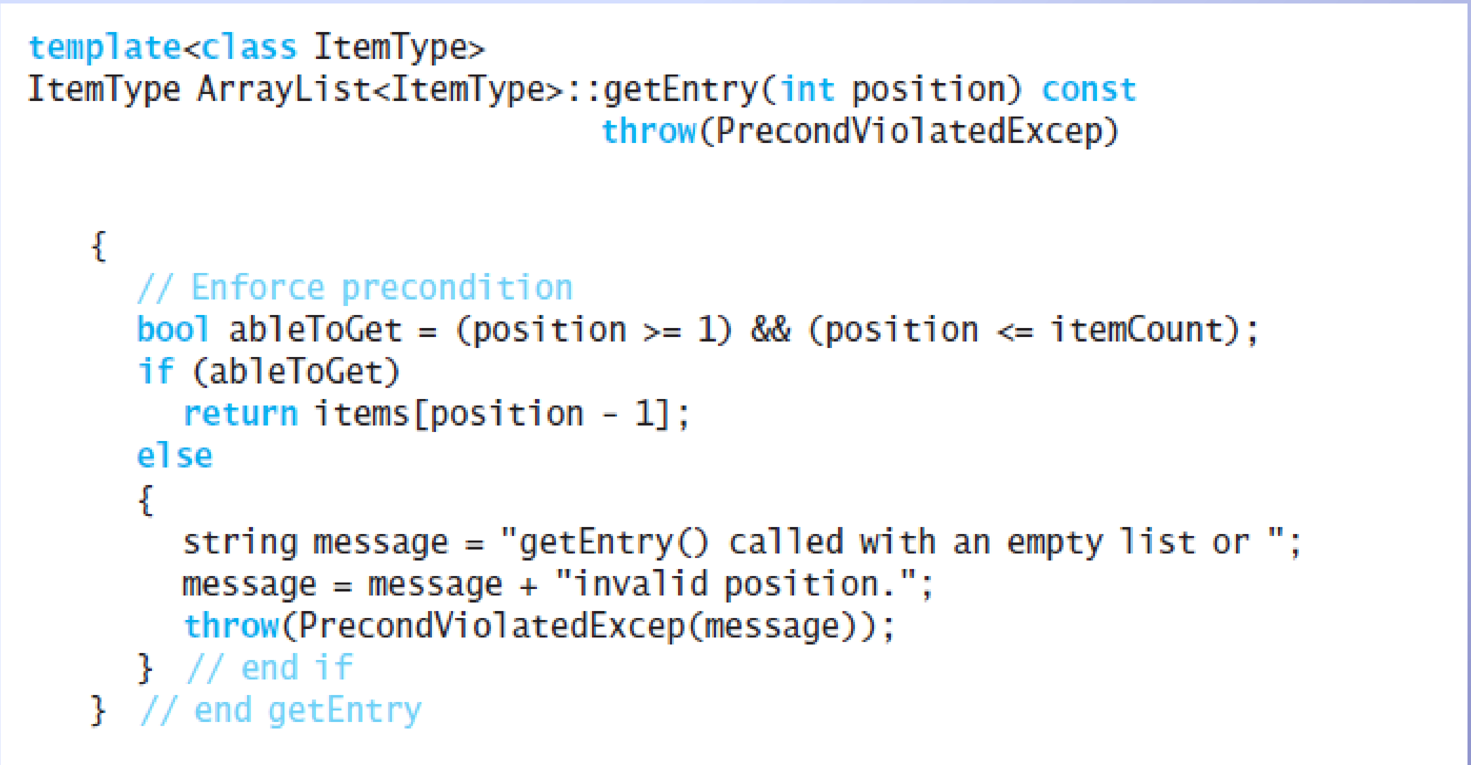
Add an entry at list position 3
(a) ADT list positions 1 2 3 4 5 6 k maxItems
items 12 3 19 100 75 10 •••• 18 ? ? •••• ?
k
Array indices 0 1 2 3 4 5 k – 1 maxItems – 1
itemCount
(b)
items 12 3 19 19 100 75 10 •••• 18 ? •••• ?
k + 1
0 1 2 3 4 5 6 k – 1 k maxItems – 1
itemCount
(c) ADT list positions 1 2 3 4 5 6 7 k + 1 maxItems
items 12 3 44 19 100 75 10 •••• 18 ? •••• ?
k + 1
Array indices 0 1 2 3 4 5 6 k maxItems – 1
itemCount
New entry

FIGURE 9-2 Shifting items for insertion: (a) the list before the insertion; (b) copy items to produce room at position 3; (c) the result

Data Structures and Problem Solving with C++: Walls and Mirrors, Carrano and Henry, © 2013

The Implementation File

- The method **`getEntry`**.

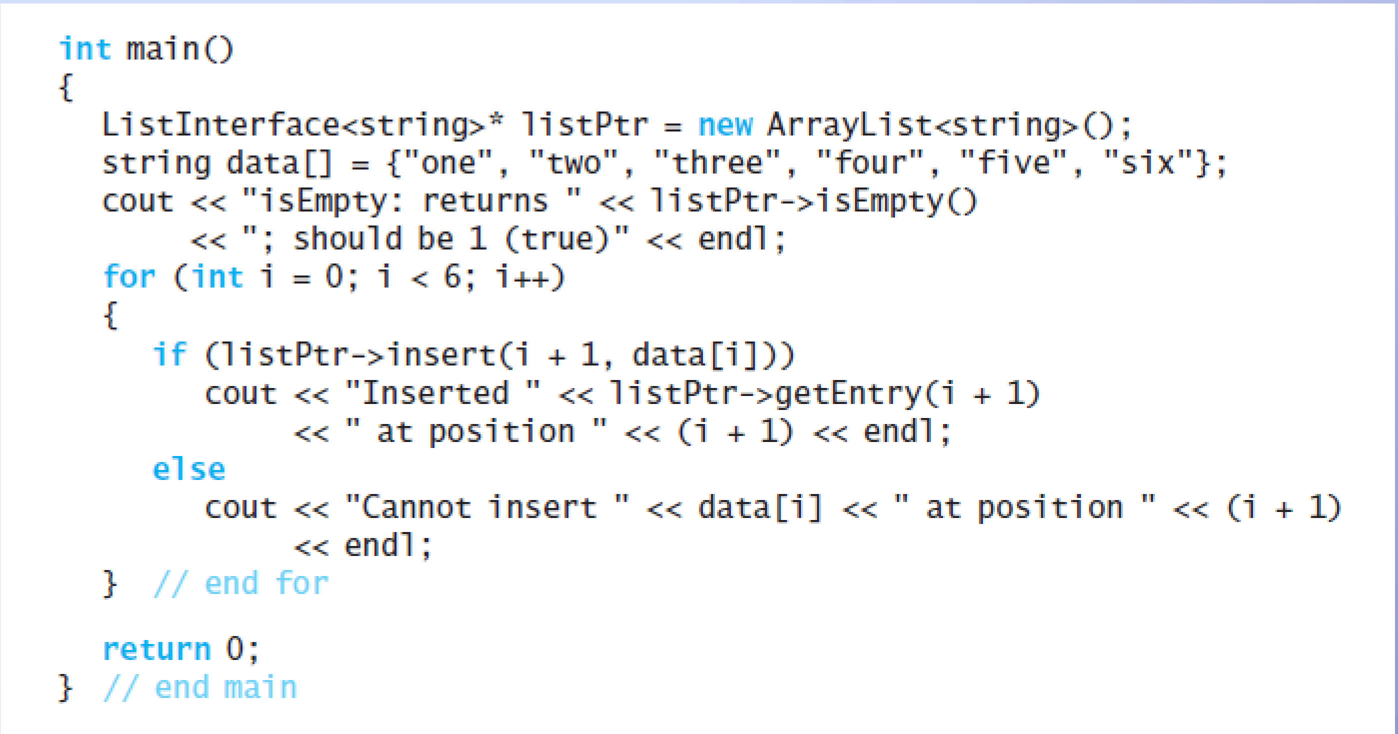


```
template<class ItemType>
ItemType ArrayList<ItemType>::getEntry(int position) const
                                 throw(PrecondViolatedExcep)


   {
      // Enforce precondition
      bool ableToGet = (position >= 1) && (position <= itemCount);
      if (ableToGet)
         return items[position - 1];
      else
      {
         string message = "getEntry() called with an empty list or ";
         message = message + "invalid position.";
         throw(PrecondViolatedExcep(message));
      }  // end if
   }  // end getEntry
```

The Implementation File

- Testing core group of methods

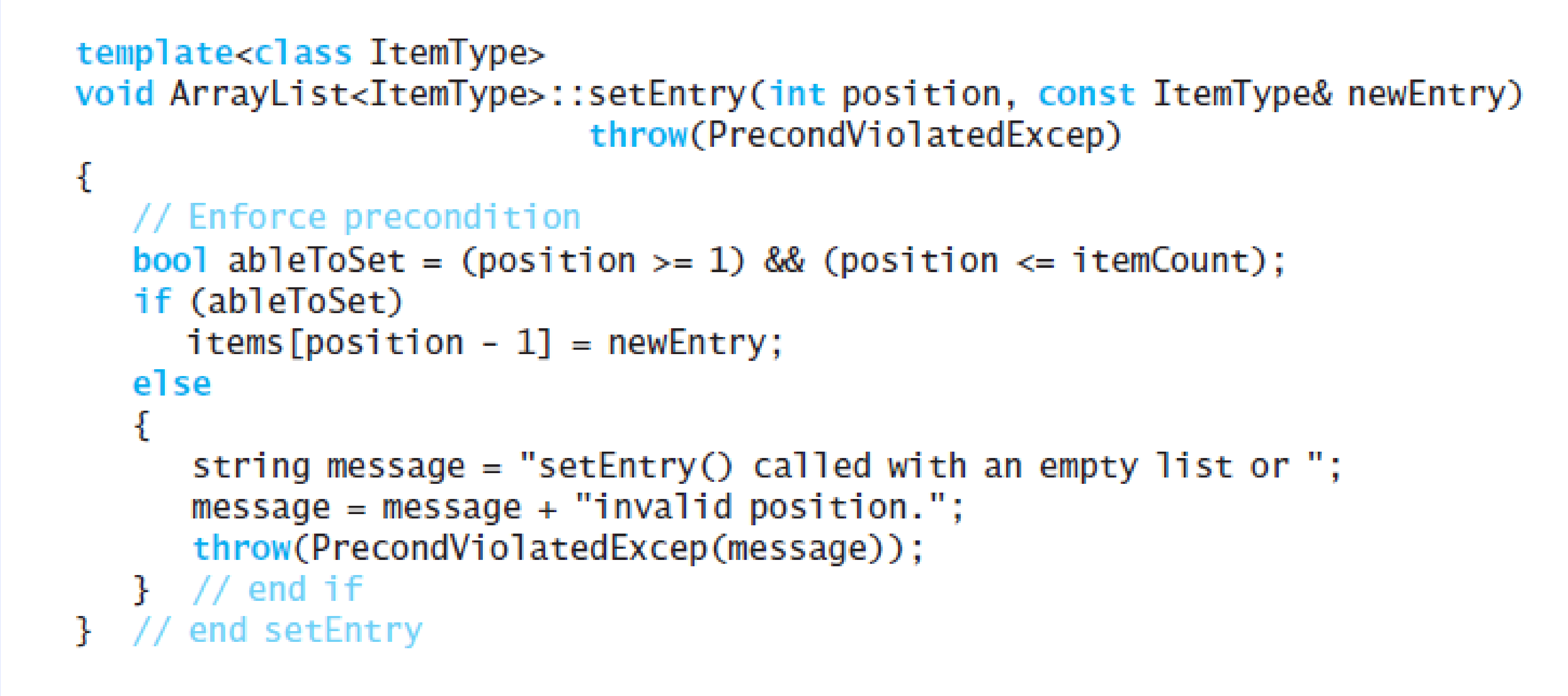


```
int main()
{
   ListInterface<string>* listPtr = new ArrayList<string>();
   string data[] = {"one", "two", "three", "four", "five", "six"};
   cout << "isEmpty: returns " << listPtr->isEmpty()
        << "; should be 1 (true)" << endl;
   for (int i = 0; i < 6; i++)
   {
      if (listPtr->insert(i + 1, data[i]))
         cout << "Inserted " << listPtr->getEntry(i + 1)
              << " at position " << (i + 1) << endl;
      else
         cout << "Cannot insert " << data[i] << " at position " << (i + 1)
              << endl;
   }  // end for

   return 0;
}  // end main
```

The Implementation File

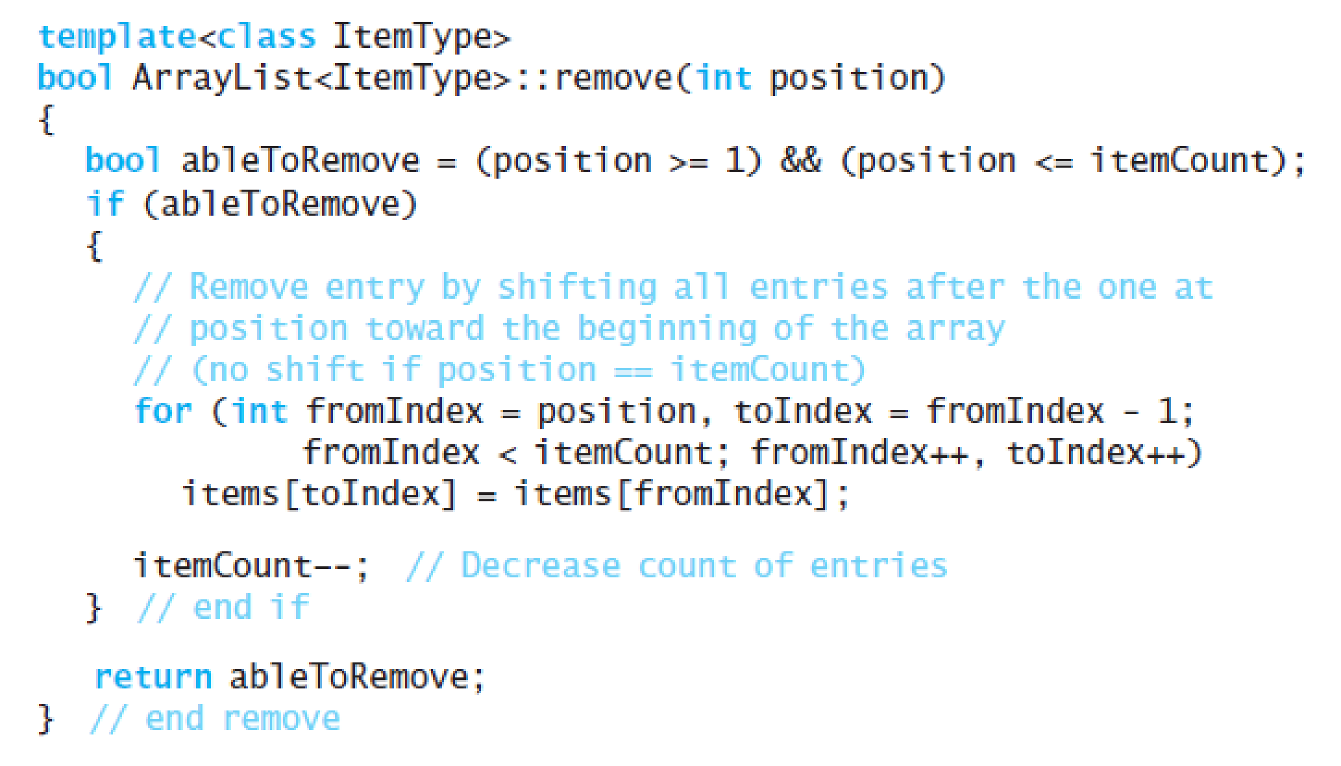
- The method **`setEntry`**



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

The Implementation File

- The definition of **`remove`**

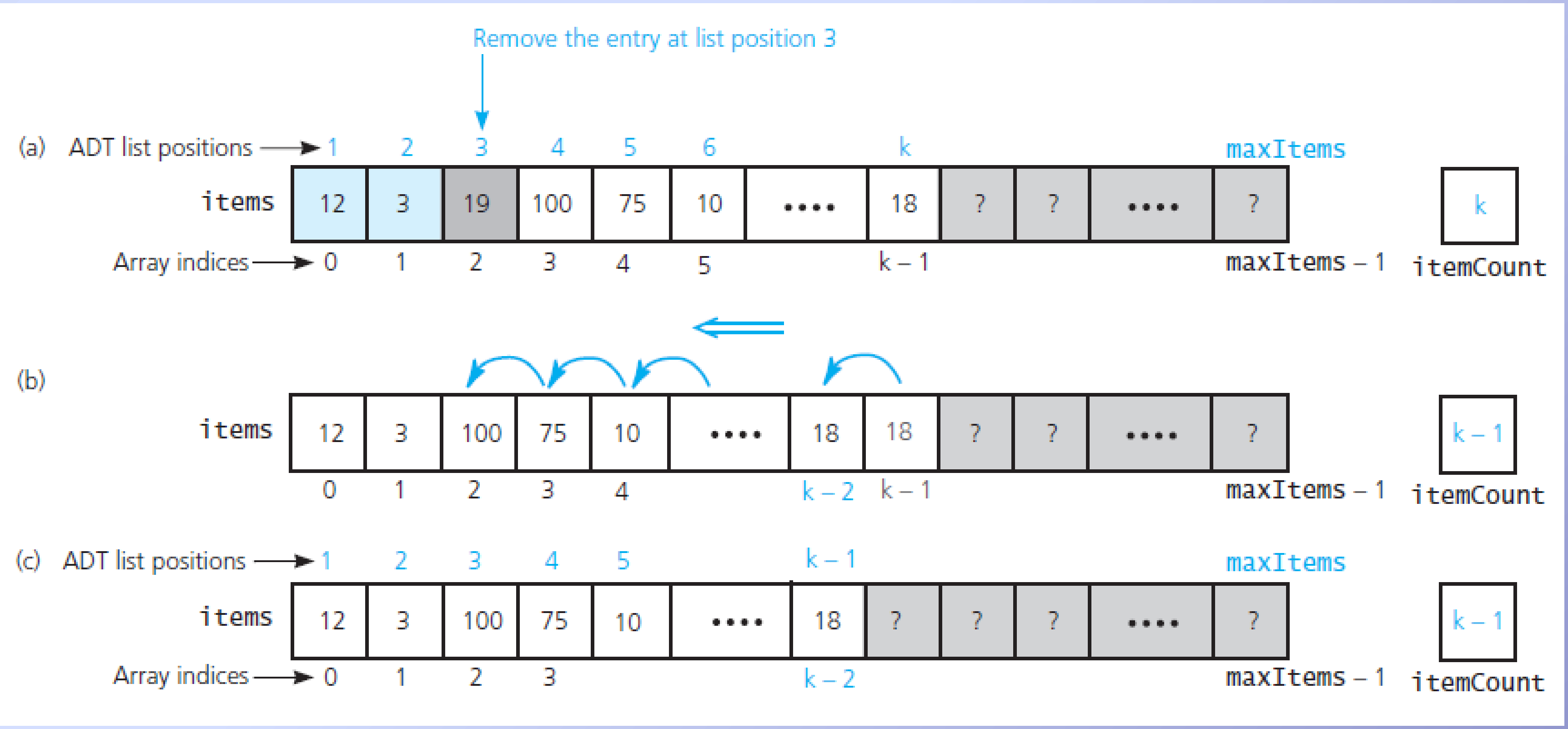


```
template<class ItemType>
bool ArrayList<ItemType>::remove(int position)
{
   bool ableToRemove = (position >= 1) && (position <= itemCount);
   if (ableToRemove)
   {
      // Remove entry by shifting all entries after the one at
      // position toward the beginning of the array
      // (no shift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

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

   return ableToRemove;
}  // end remove
```

The Implementation File



Remove the entry at list position 3

(a) ADT list positions → 1 2 3 4 5 6 k maxItems

items 12 3 19 100 75 10 •••• 18 ? ? •••• ? k

Array indices → 0 1 2 3 4 5 k – 1 maxItems – 1 itemCount

(b)

items 12 3 100 75 10 •••• 18 18 ? ? •••• ? k – 1

0 1 2 3 4 k – 2 k – 1 maxItems – 1 itemCount

(c) ADT list positions → 1 2 3 4 5 k – 1 maxItems

items 12 3 100 75 10 •••• 18 ? ? ? •••• ? k – 1

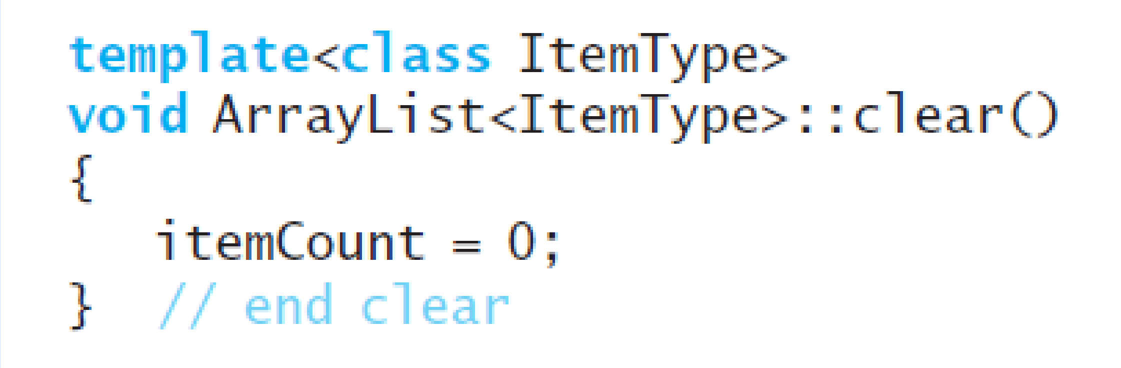
Array indices → 0 1 2 3 k – 2 maxItems – 1 itemCount

FIGURE 9-3 (a) Deletion can cause a gap; (b) shift items to prevent a gap at position 3; (c) the result

Data Structures and Problem Solving with C++: Walls and Mirrors, Carrano and Henry, © 2013

The Implementation File

- The method **`clear`**.



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

A Link-Based Implementation of the ADT List

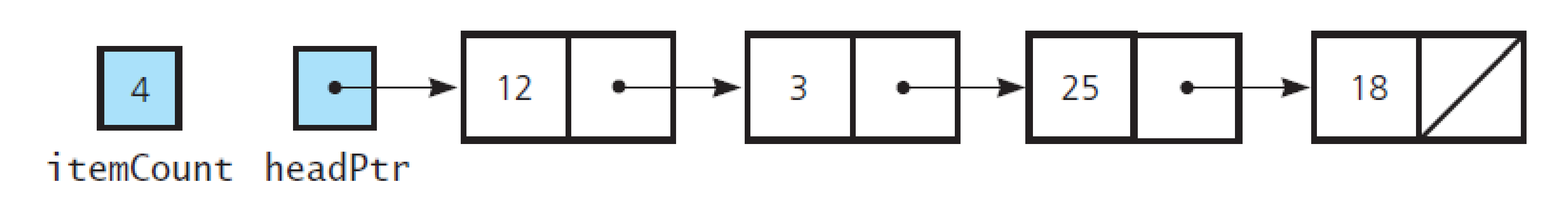


FIGURE 9-4 A link-based implementation of the ADT list

End

Chapter 9