This chapter presents a sample implementation of a linked list, encapsulated in a C++ class.

The primary goals of this implementation are:
- to provide a proper separation of functionality.
- to design the list to serve as a container; i.e., the list should be able to store data elements of any type.

First, a `LinkNode` class is used to encapsulate the low-level pointer operations.

Second, a `LinkList` class is used to encapsulate a list of `LinkNode` objects.

Third, an `Item` class is used to encapsulate the data and separate it from the pointers that define the list structure.

The basic view is that each list node provides a data "socket" that is capable of accepting any type of data element:

Warning: the `LinkList` class given in this chapter is intended for instructional purposes. The given implementation contains a number of known flaws, and perhaps some unknown flaws as well. Caveat emptor.
LinkNode class is used to encapsulate pointer operations:

```cpp
class LinkNode {
private:
  Item Data; // data "capsule"
  LinkNode* Next; // pointer to next node
public:
  LinkNode();
  LinkNode(const Item& newData);
  void setData(const Item& newData);
  void setNext(LinkNode* const newNext);
  Item getData() const;
  LinkNode* getNext() const;
};
```

Node Class Constructors

LinkNode constructor implementations:

```cpp
LinkNode::LinkNode() { //explicit initialization of
  Data = Item();
  Next = NULL;
}
```

We are assuming that Item is a class. (Do you see where?)
Node Class Mutators

**LinkNode mutator implementations:**

```cpp
// Sets new value for Data element of object.
// Parameters:
//  newData  Data element to be stored in node
// Pre:      none
// Post:     Data field of object has been modified to hold newData
//
// void LinkNode::setData(const Item& newData) {
//  Data = newData;
//}
```

```cpp
// Sets new value for Next pointer of object.
// Parameters:
//  newNext  new value for pointer field
// Pre:      none
// Post:     Next field of object has been modified to hold newNext
//
// void LinkNode::setNext(LinkNode* const newNext) {
//  Next = newNext;
//}
```

Why is the parameter to setNext not passed as:
```cpp
const LinkNode* const newNext
```

Node Class Reporters

**LinkNode reporter implementations:**

```cpp
// Returns value of Data element of object.
// Parameters: none
// Pre:   object has been initialized
// Post:  Data field of object has been returned
// Item LinkNode::getData() const {
//  return Data;
//}
```

```cpp
// Returns value of Next pointer of object.
// Parameters: none
// Pre:   object has been initialized
// Post:  Next field of object has been returned
// LinkNode* LinkNode::getNext() const {
//  return Next;
//}
```

Uses const to guarantee no modification occurs.
LinkList class is used to encapsulate all high-level list operations:

```cpp
// LinkList.h
// The LinkList class provides an implementation for a
// singly-linked list consisting of ListNode objects.
// User must provide a declaration and implementation
// of a class named Item with a default constructor and
// an overloaded == operator in order for the given
// implementation of LinkNode to be valid.

#ifndef LINKLIST_H
#define LINKLIST_H
#include <cassert>
#include "LinkNode.h" // for node declaration
// #include "Item.h" // must be included by user
class LinkList {
private:
  ListNode* Head; // points to head node in list
  ListNode* Tail; // points to tail node in list
  ListNode* Curr; // points to "current" node
public:
  LinkList(); //constructor
  ~LinkList(); //destructor
  bool isEmpty() const;
  bool inList() const;
  bool PrefixNode(const Item& newData);
  bool Insert(const Item& newData);
  bool Advance();
  void gotoHead();
  void gotoTail();
  bool DeleteCurrentNode();
  bool DeleteValue(const Item& Target);
  Item getCurrentData() const;
  void setCurrentData(const Item& newData);
  // missing: copy constructor, assignment overload FNs
};
#endif
```

One line Fns could be "inline" for efficiency.

One line Fns could be "inline" for efficiency.

See Copying Objects notes for missing functions.
7. LL Class

LinkList Destructor

Code

```cpp
 OnTriggerEnter(// Default destructor for LinkList objects.// Parameters: none
// Pre: LinkList object has been constructed
// Post: LinkList object has been destructed; // all dynamically-allocated nodes // have been deallocated.

void LinkList::~LinkList() {
    LinkNode* toKill = Head;
    while (toKill != NULL) {
        Head = Head->getNext();
        delete toKill;
        toKill = Head;
    }
    Head = Tail = Curr = NULL;
}
```

Compiler generates calls to the destructor automatically whenever a LinkList object goes out of scope (i.e., its lifetime ends: at the end of the function/block in which the objects are defined, when a dynamically allocated object is destroyed with delete(), when an object containing a member object is destroyed).

A class destructor's name is always the tilde followed by the name of the class. It has no parameters or return type and cannot be overloaded.

LinkList needs a destructor in order to properly return the dynamically-allocated nodes to the system heap.

LinkList Reporters

Code

```cpp
bool LinkList::isEmpty() const {
    return (Head == NULL);
}
```

```cpp
bool LinkList::inList() const {
    return (Curr != NULL);
}
```

LinkList uses a pointer (Curr) to keep a sense of the current position in the list as operations are performed. This isn’t absolutely necessary (especially if the list is to be kept sorted in some order), but it is useful for general lists.
7. LL Class

**LinkList PrefixMutator**

---

**Code: inserting at the head of the list**

```cpp
// Inserts a new LinkNode at the front of the list.
//
// Parameters:
//   newData  Data element to be inserted
//
// Post:   LinkNode containing newData has been constructed and inserted at the
//           beginning of the list, if possible.
//
// Returns: true if operation succeeds
//          false otherwise

bool LinkList::PrefixNode(const Item& newData) {
    LinkNode* newNode = new(nothrow) LinkNode(newData);
    if (newNode == NULL) return false;
    if ( isEmpty() ) {
        newNode->setNext(NULL);
        Head = Tail = Curr = newNode;
        return true;
    }
    newNode->setNext(Head);
    Head = newNode;
    return true;
}
```

Uses LinkNode member functions to modify node pointers — this gives a separation between the “high” level list functions the user sees and the “massaging” of the pointers.

---

**LinkList Insert Mutator**

---

**Code: inserting after the current position**

```cpp
// Inserts a new LinkNode immediately after the current position in the list.
//
// Parameters:
//   newData  Data element to be inserted
//
// Post:   LinkNode containing newData has been constructed and inserted after
//           the current position, if possible.
//
// Returns: true if operation succeeds
//          false otherwise

bool LinkList::Insert(const Item& newData) {
    if (Curr == NULL) return false;
    LinkNode* newNode = new(nothrow) LinkNode(newData);
    if (newNode == NULL) return false;
    if ( isEmpty() )
        return false;
    newNode->setNext(Curr->getNext());
    Curr->setNext(newNode);
    if (Curr == Tail)
        Tail = newNode;
    return true;
}
```

Uses LinkNode member functions to modify node pointers — this gives a separation between the “high” level list functions the user sees and the “massaging” of the pointers.

---

Uses LinkNode member functions to modify node pointers — this gives a separation between the “high” level list functions the user sees and the “massaging” of the pointers.

---

**Note test for valid current position.**

---

**Why should this case never occur?**

---

**Pointer dereference. This is a very good place to blow up at runtime if you don’t verify newNode is not NULL prior to this statement. Is this statement necessary? (Included as a precaution?)**
LinkList Position Mutators

Code: changing the current position

```cpp
void LinkList::gotoHead() {
    Curr = Head;
}

void LinkList::gotoTail() {
    Curr = Tail;
}

bool LinkList::Advance() {
    if (Curr != NULL) {
        Curr = Curr->getNext();
        return true;
    } else {
        return false;
    }
}
```

Note test for valid current position.

LinkList Delete Curr Mutator

Code: deleting the current node

```cpp
bool LinkList::DeleteCurrentNode() {
    LinkNode* delThis;
    if (Curr == NULL) return false;
    if (Curr == Head) { // delete Head node
        delThis = Curr;
        Head = Head->getNext();
        Curr = Head;
        if (Tail == delThis) Tail = Curr;
        delThis->setNext(NULL);
        delete delThis;
        return true;
    } else { // locate Curr's previous node
        LinkNode* prevNode = Head;
        while (prevNode != NULL && prevNode->getNext() != Curr) {
            prevNode = prevNode->getNext();
        }
        if (prevNode == NULL) return false;
        // previous found bypass and delete Curr
        delThis = Curr;
        prevNode->setNext(Curr->getNext());
        Curr->setNext(prevNode->getNext());
        if (Tail == delThis) Tail = prevNode;
        delete delThis;
        return true;
    }
}
```

Test for valid current position.
Handle deletion of head node.
Find previous node.
If not found, error.
Handle deletion of node in middle or at tail of list.
### LinkList Delete Value Mutator

**Code: deleting a list value**

```cpp
bool LinkList::DeleteValue(const Item& Target) {
    LinkNode* myCurr = Head;
    LinkNode* myTrailer = NULL;
    while ( (myCurr != NULL) &&
            !(myCurr->getData() == Target) ) {
        myTrailer = myCurr; myCurr = myCurr->getNext();
    }
    if (myCurr == NULL) return false;
    if (myTrailer == NULL)
        Head = Head->getNext();
    else
        myTrailer->setNext(myCurr->getNext());
    if (Curr == myCurr) Curr = myTrailer;
    if (Tail == myCurr) Tail = myTrailer;
    myCurr->setNext(NULL);
    delete myCurr;
    return true;
}
```

- Look for matching node.
- If not found, error.
- Handle deletion of node in middle or at tail of list.

### LinkList Set Curr Mutators

**Code: changing the Data in the current node**

```cpp
void LinkList::setCurrentData(const Item& newData) {
    assert (Curr != NULL);
    Curr->setData(newData);
}
```

This implementation places a burden on the user of the class. If the current position is undefined (e.g., if the list is empty), then the call to `assert()` will cause the program to terminate rather gracelessly. A better design would alert the user/client:

```cpp
bool LinkList::setCurrentData(const Item& newData) {
    if (!Curr) return false;
    Curr->setData(newData);
    return true;
}
```

If no current position, die. 😞
LinkList Reporter

Code: returning the Data in the current node

```cpp
Item LinkList::getCurrentData() const {
  assert (Curr != NULL);
  return (Curr->getData());
}
```

This possible premature termination due to an undefined current position could be eliminated by having the function return a pointer to a copy of the data element, or by having the function use a reference parameter to communicate a copy of the data value to the caller, and also return true/false to indicate success.

Better design: maintain an internal error state in the class. (E.g., similar to the stream status in `<iostream>`).

Note: a pointer to an object in a list (i.e. `Item*`) or a reference to an object in a list (i.e. `Item&`) should NOT be returned by a member function. Why?

Data Element Class

The user must typedef `Item` to match the data class that he/she really wishes to use. Recall the Inventory Class:

```cpp
// *********** INVENTORY CLASS DECLARATION ************
class InvItem {
  private:
    string SKU;    //Stock Unit #: KEY FIELD
    string Description; //Item Details
    int Retail;      //Selling Price
    int Cost;        //Store Purchase Price
    int Floor;       //Number of Items on display
    int Warehouse;   //Number of Items in stock
  public:
    InvItem();       //default constructor
    InvItem(const string& iSKU, const string& iDescription,
             int iRetail, int iCost,
             int iFloor, int iWarehouse);
    //Reporter Member Functions
    // . . . Unchanged from previous declaration
    //Mutator Member Functions
    // . . . Unchanged from previous declaration
    //Operator Overloads
    bool operator==(const InvItem& anItem);
}; // class InvItem
```

typedef InvItem Item;  // Required type name equivalency definition
### Data Class Equality Operator

Inventory class equality operator:

```cpp
//-------------- Operator Overload Functions -----------
// Operator == Fn for InvItem Class
// Parameters: an Item to compare
// Pre: members have been initialized
// Post: T/F comparison of SKUs returned
// bool InvItem::operator==(const InvItem& anItem){
    return (SKU == anItem.getSKU());
}
```

This simple operator overload function is required for the correct use of the LinkList class. The `DeleteValue()` function assumes that two Item objects can be compared for equality, (but not inequality).

By only testing the SKU members for equality the code is reflecting a design decision that the SKU numbers of all Inventory items must be unique.

```cpp
bool InvItem::operator==(const InvItem& anItem){
    return (SKU == anItem.SKU);
}
```

### LinkList Search

Sequential search function for LinkList:

```cpp
// Search function for LinkList Item objects.
// Parameters:
//    List     a LinkList object
//    Item     a Item object
// Pre: LinkList object has been constructed
// Equality oper. overloaded for Item
// Post: returns true if anItem is found in List
//        and false otherwise
//bool Search(LinkList& List, const Item& anItem) {
if (List.isEmpty())
    return false;
else {
    List.gotoHead();
    while( (List.inList()) &&
           !((List.getCurrentData() == anItem)) )
        List.Advance();
    return (List.inList());
}
```

Note: this function is “external” to the LinkList class. The inclusion of the function as a LinkList class member function is left as an exercise.

Why is the second condition in the while boolean expression not stated:

```cpp
!(anItem != (List.getCurrentData()))
```

Even more subtle, why can it not also be stated:

```cpp
!(anItem == (List.getCurrentData()))
```
Alternate Implementation

Alternate PrefixNode() Implementation:

```cpp
/////////////////////////////////////////////////
// Inserts a new LinkNode at the front of the// list.// Parameters: new Data element to be inserted// Pre: LinkList object has been constructed// Post: LinkNode containing newData has been mistakenly constructed and inserted at the// beginning of the list, if// possible.// Returns: true if operation succeeds // false otherwise
bool LinkList::PrefixNode(const Item& newData) {
    LinkNode newNode(newData);
    if ( isEmpty() ) {
        newNode.setNext(NULL);
        Head = Tail = Curr = &newNode;
        return true;
    }
    newNode.setNext(Head);
    Head = &newNode;
    return true;
}
```

Is the above implementation superior or inferior to the original implementation of PrefixNode(), see slide 7.11?

Merge Lists (preservation)

/* Given 2 ascending ordered single linked-lists, return a new ordered list which contains all of the elements of both lists, (the original lists must NOT be destroyed by the merging). */

```cpp
LinkList MergeLists(LinkList L1, LinkList L2){
    Item TmpData;
    LinkList merge;
    L1.gotoHead();
    while (L1.inList()) {
        TmpData = L1.getCurrentData();
        insertion(merge, TmpData);
        L1.Advance();
    }
    L2.gotoHead();
    while (L2.inList()) {
        TmpData = L2.getCurrentData();
        insertion(merge, TmpData);
        L2.Advance();
    }
    return merge;
}
```

```cpp
void AddList(LinkList& target, LinkList source)
{
    Item TmpData;
    source.gotoHead();
    while (source.inList()) {
        TmpData = source.getCurrentData();
        insertion(target, TmpData);
        source.Advance();
    }
}
```

AddList(merge, L1);
AddList(merge, L2);
AddList(merge, L1);
AddList(merge, L2);

WARNING: untested code!
7. LL Class

Ordered Insertion

Ordered Insertion is a non-class, non-member function to perform an ordered LinkList insertion.

```cpp
bool insertion(LinkList& list, const Item & newData) {
    list.gotoHead();
    while (list.inList() && list.getCurrentData() < newData) list.Advance();
    if (list.isEmpty()) return(list.PrefixNode(newData));
    else if (!list.inList()) { //newData > tail
        list.gotoTail();   //append to tail
        return(list.Insert(newData));
    } else { //insert before Current list element
        Item tmpData = list.getCurrentData();
        list.setCurrentData(newData);
        return(list.Insert(tmpData));
    }
}
```

Sets current node to contain newData item and inserts old node data item after current node.

WARNING: untested code!

Merge Lists (no preservation) in situ

Merge Lists (no preservation) is a function to merge two ascending ordered single linked-lists. The lists are returned in the first list, which is the merged list. The second list is destroyed during the merging.

```cpp
void Mergelists(LinkList& L2) {
    LinkNode* MergeHead;
    LinkNode* trail1;
    LinkNode* trail2;
    Item i1, i2;
    if (Head == NULL) { //this empty return L2 List
        Head = L2.Head; L2.Head = NULL;
        Curr = L2.Curr; L2.Curr = NULL;
        Tail = L2.Tail; L2.Tail = NULL;
        return;
    }
    if (L2.Head == NULL) //return this List
        return;
    MergeHead = (Head->getData() < L2.Head->getData()) ? Head : L2.Head;
    while (Head != NULL) && (L2.Head != NULL) {
        i1 = Head->getData(); i2 = L2.Head->getData();
        if (i1 == i2) {//equal current merge items
            trail2 = L2.Head->getNext(); //due to initial/current
            L2.Head->setNext(Head);
            MergeHead->setNext(L2.Head);
        } else if (i1 < i2) {//insert before current item
            trail1 = Head->getNext();
            MergeHead->setNext(trail1);
            MergeHead = Head;
            Head = trail2;
        } else if (i1 > i2) {//append to tail
            trail1 = L2.Head->getNext();
            MergeHead->setNext(trail1);
            MergeHead = L2.Head;
            L2.Head = trail2;
        }
    }
}
```

Assumes elements within list are unique.

Problem: if List2 contains multiple items equal to head of list1?

while conditions rely upon Boolean short-circuiting.

WARNING: untested code!
Merge Lists \textit{in situ} (cont)

```c
else
    if ( i1 < i2 ) { //advance this list
        while ( (Head != NULL) && //until end of list
                    (Head->getData() < i2) ) { //or smaller
            trail1 = Head; // item is found
            Head = Head->getNext();
        } //while
        trail1->setNext(L2.Head);
    } //if

    else { //i2 < i1               //advance L2 list
        while ( (L2.Head != NULL) &&//until end list
                    (L2.Head->getData() < i1) ) { //or
            trail2 = L2.Head; //smaller item is
            L2.Head = L2.Head->getNext(); //found
        } //while
        trail2->setNext(Head);
    } //else
} //while

if ( Head == NULL )//L2 is longer list
    Tail = L2.Tail; //update Tail
Head = MergeHead;
```

Duplicated code should be eliminated.

WARNING: untested code!