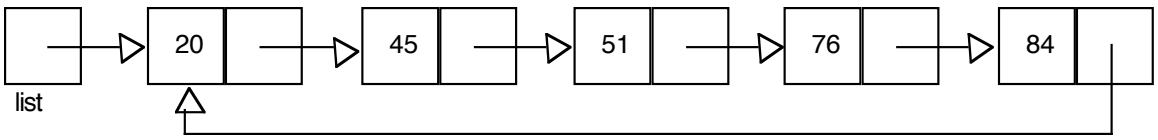
Chapter 4-3, Lecture notes

Variations of the Linked Lists

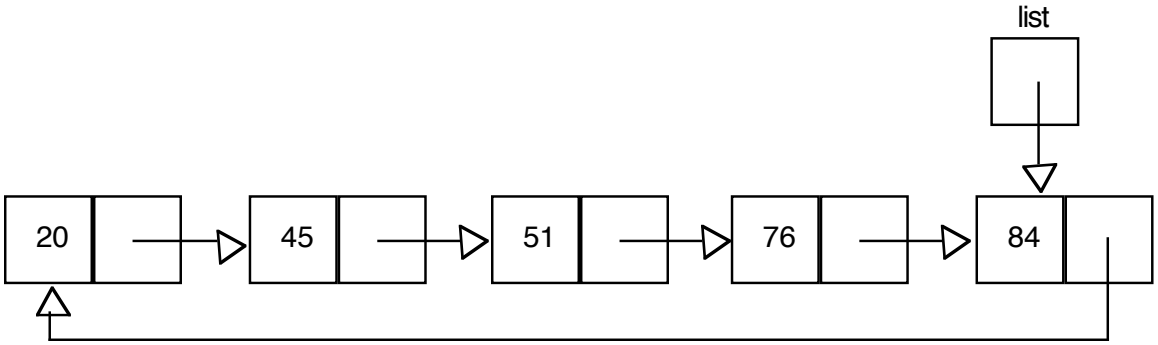
1. Thought the concept of linked lists is difficult, it is important.
   1.1 Many mechanisms that create a linked list can be used in other data structures such as binary trees (cover later on).
   1.2 There are also variations on linked lists.

   1.3 Consider a server (i.e. a computer that will give services to client computers – share resources) that has a list of clients or users that wait for services.

      1.3.1 Can put the users in a linked list to keep order.
      1.3.2 But it is inconvenient when you hit the end of the list and there is no other node to point to.
      1.3.3 You want to go to the beginning of the list.
      1.3.4 You can access head pointer again or – you can have a "circular linked list" where the last node points back to the first node.
      1.3.5 Like so (see figure 4-25, page 216):



      1.3.6 The other linked lists of the previous section were a "linear linked list".
      1.3.7 With a circular linked list, every node had a successor so you can start at any node and traverse the entire list.
         1.3.7.1 Instead of using "head" as the pointer to the linked list, you can use "list".
         1.3.7.2 The way to tell if you have traversed the entire list is to check the current pointer to the list pointer to see if you have wrapped around to the beginning.
         1.3.7.3 You can also have list point to the last node like so (see figure 4-26, page 217):



1.3.7.4 The first node would actually be list->next and not list.

1.3.7.5 One advantage of the above is that you can easily add to the end of the list.

1.3.8 Let's look at the display routine that would display the values of the above list (note: the "display" function would display the data in item in the proper format):

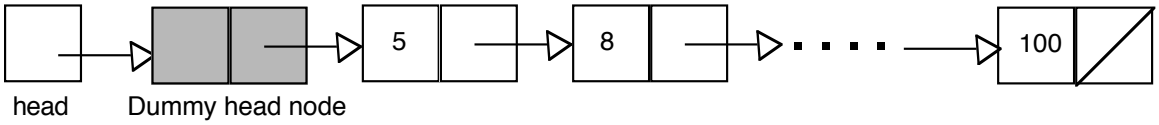
```
// display the data in a circular linked list; list points to its last node
if (list != NULL)
{  // list is not empty
   Node *first = list->next; // point to first node

   Node *cur = first;          // start at the first node
   // Loop invariant: cur points to the next node to display
   do
   {  display(cur->item);     // write data portion
      cur = cur->next;         // point to the next node
    } while (cur != first);    // list traversed?
} // end if
```

1.4 Another variation of the linked list is to overcome the problem with the special case of checking for the beginning of the linked list for the insertion and deletion algorithms.

1.4.1 One way to avoid that special case is to have a "dummy head node" – one node that is ALWAYS there but not used!

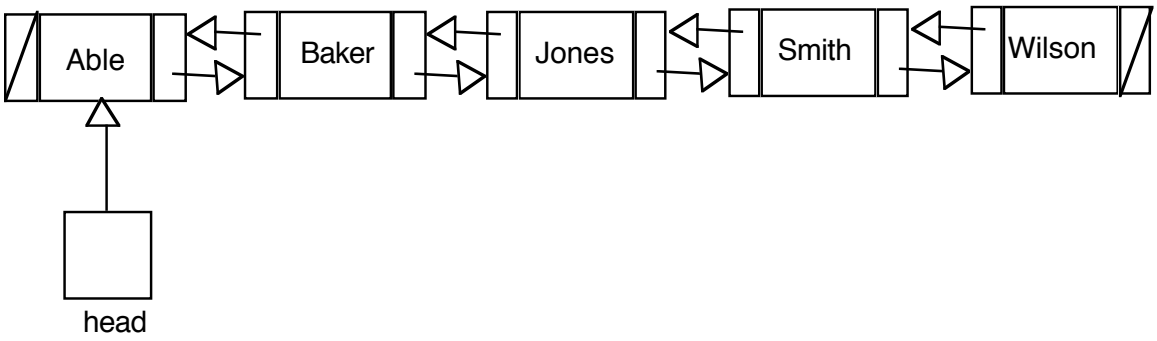
1.4.2 Like so (see figure 4-27, page 218):



1.4.3 The code for inserting and deleting is simpler since you do not have to worry about the special case (a small price for wasting one node). The dummy head node is especially useful for doubly linked lists.

2 One problem with linked lists is if you want to "backup" to the previous node, you need to have the "prev" pointer to do so.

2.1 However, with "doubly linked list", you would have not only a "next" pointer but also a "precede" pointer to point to the previous node. Like so (see figure 4-28):



2.2 So you can do: prev = cur->precede;

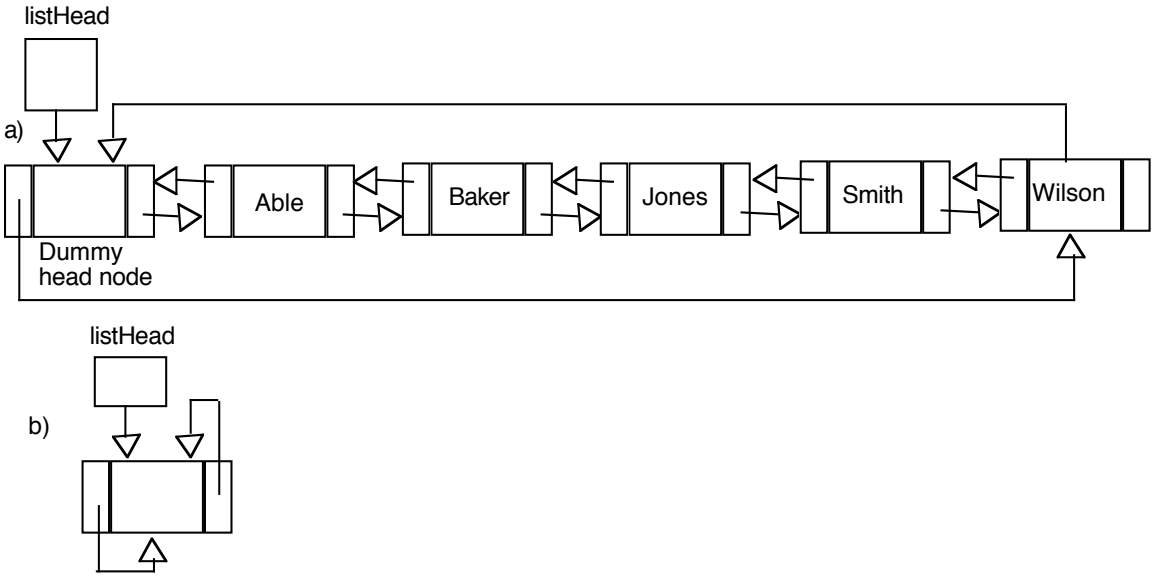
2.2.1 And thus go backwards if you desire.

2.2.2 The downside is that inserting and deleting in a doubly linked list is a bit more involved.

2.2.3 That is one reason that a dummy head node is more attractive for a doubly linked list.

2.3 Let's extend it a bit further and have a "circular doubly linked list".

2.3.1 The following shows it with a part a for a circular doubly linked list and part b for an empty one (see figure 4-29, page 219):

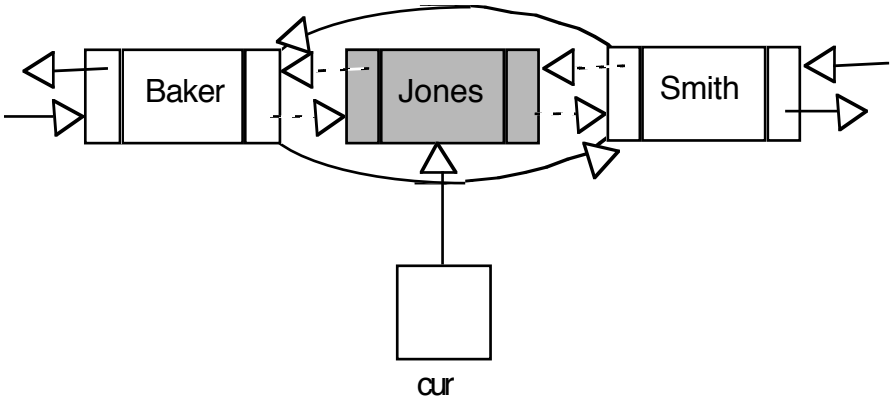


2.3.2 To delete a node N you must:

- Change the "next" pointer of the node that precedes N so that it points to the node that follows N

- Change the precede pointer of the node that follows N so that it points to the node that precedes N

2.3.3 The following shows what has to be changed to delete a node in a circular doubly linked list (see figure 4-30, page 220):



2.3.3.1 The C++ code to accomplish the above is:

```
// delete the node to which cur points
(cur->precede)->next = cur->next;
(cur->next)->precede = cur->precede;
```

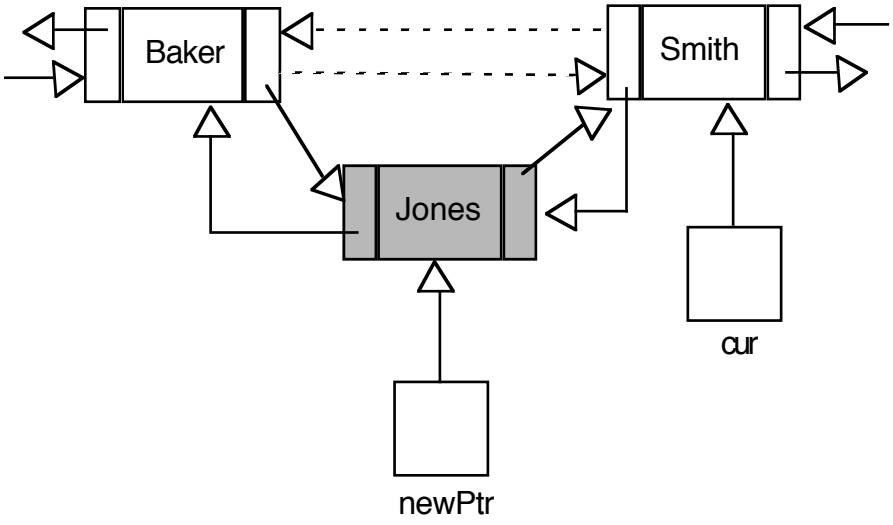
2.3.3.2 To insert in the circular doubly linked list, you must first find the point to insert the node. The following pseudocode does that:

```
// find the insertion point
cur = listHead->next // point to the first node, if any
while (cur ≠ listHead and newName > cur->item)
        cur = cur->next
```

2.3.4 Note that if you want to insert the new node either at the end of the list or into an empty list, the loop will set cur to point to the dummy head node. To insert you must:

a. Set next pointer in the new node to point to the node that is to follow
b. Set precede pointer in new node to point to node that is to precede
c. Set the precede node in the node that is to follow the new node so that is points to the new node
d. Set the next pointer in the node that is to precede the new node so that it points to the new node

2.3.5 The following illustrates what happens (see figure 4-31, page 221):



2.4 The following C++ code does these four steps:

```
// insert the new node pointed to by newPtr before the node pointed to by cur
newPtr->next = cur;
newPtr->precede = cur->precede;
cur->precede = newPtr;
newPtr->precede->next = newPtr;
```

2.5 Look over the above statements to see that they indeed work!

Chapter 4-4, Application: Maintaining an Inventory

1 Let's try an application.

1.1 Imagine you work at a local DVD store.

1.1.1 The store manager wants you to write an interactive program to maintain the store's inventory of DVDs that are for sale.

1.1.2 The following is needed for each title:

- Have value: number of DVDs currently in stock
- Want value: number of DVDs that should be in stock (if less than have value, order more DVDs)
- Wait list: list of names of people waiting for the title if it's sold out

1.2 The program has to be able to write out the inventory to disk and to read it back in. The program input would be:

- A file that contains a previously saved inventory
- A file that contains information on an incoming shipment of DVDs
- Single-letter commands – with arguments if necessary – that inquire about or modify the inventory and that is interactive

1.3 The program output would be:

- A file that contains the updated inventory
- Output as specified by the individual commands

1.4 The program commands would be:

H	(help)	Provide a summary of available commands
I <title>	(inquire)	Display inventory information for a specified DVD
L	(list)	List the entire inventory
A <title>	(add)	Add a new title to the inventory. Prompt for initial want value
M <title>	(modify)	Modify the want value for a specified title
D	(delivery)	Take delivery of a shipment of DVDs, assuming that the clerk has entered the shipment information (titles and counts) into a file. Read the file, reserve DVDs for the people on the wait list, and update the have values in the inventory accordingly. The program must add an item to the inventory if a delivered title is not present in the current inventory
O	(order)	Write a purchase order for additional DVDs based on a comparison of the have values with the want values in the inventory
R	(return)	Write a return order based on a comparison of the have and want values in the inventory and decrease the have values accordingly (make the return). Reduce the have value to the want value
S <title>	(sell)	Decrease the count for the specified title by 1. If sold out, put a name on the wait list for the title
Q	(quit)	Save the inventory and wait lists in a file and terminate execution

2 Unfortunately, the solution to the above programming problem is not clear-cut – you must make choices as you are creating the program.

2.1 The problem is data management and needs certain program commands.

2.2 You might need the following commands:

- List the inventory in alphabetic order by title (L command)
- Find the inventory item associated with a title (I, M, D, O, and S commands)
- Replace the inventory item associated with a title (M, D, R, and S commands)
- Insert new inventory items (A and D commands)

2.3 You might have a wait list for each title. You must be able to:

- Add new people to the end of the wait list when they want a DVD that is sold out (S command)
- Delete people from the beginning of the wait list when new DVDs are delivered (D command)
- Display the names on a wait list for a particular title (I and L commands)
- Save the current inventory and associated wait lists when program execution terminates (Q command)
- Restore the current inventory and associated wait lists when program execution begins again

2.4 You can think of the operations of the ADT inventory.

2.4.1 The next step would be to specify each of the operations fully.

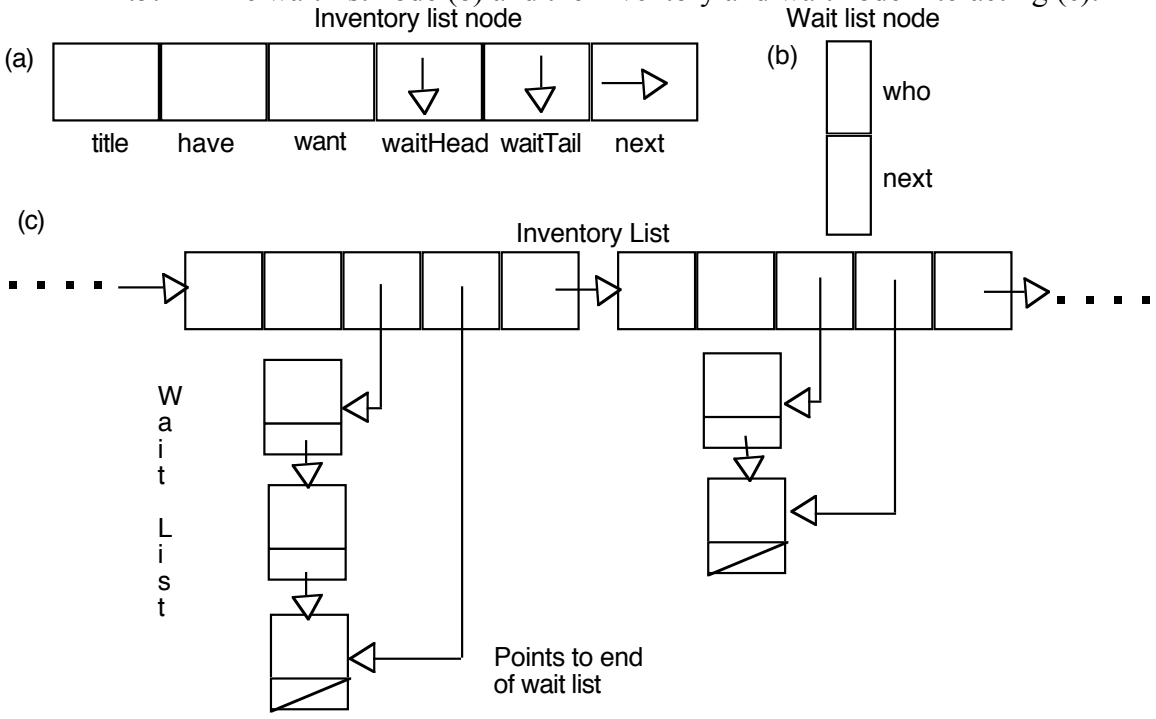
2.4.2 You can use a linked list implementation that each entry could have another linked list that a node points to.

2.4.3 You can have as many DVD entries as desired and as many on the wait list.

2.5 The following shows a possible implementation (figure 4-32 page 224).

2.5.1 The "Inventory list node" in (a) would have the title, have, and the want with a link to the waitHead and the waitTail.

2.5.2 The wait list node (b) and the inventory and wait node interacting (c).



2.6 So the above nodes will do:

- The inventory is a linked list of data items, sorted by the title that each item represents
- Each inventory items contains a title, a have value, a want value, a pointer to the beginning of a linked list of people's names (wait list), and a pointer to the last name in the wait list

2.6.1 The C++ statements are:

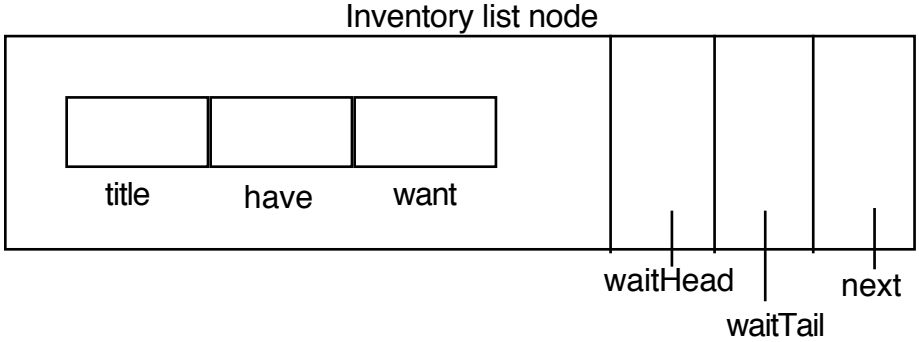
```
// node for wait list of people waiting for certain titles
struct WaitNode
{       string who;
        WaitNode *next;
}; // end WaitNode

// node for inventory list of stock items
struct StockNode
{       string   title;
        int      have, want;
        WaitNode  *waitHead, *waitTail;
        StockNode *next;
}; // end StockNode
```

2.6.2 However, the above can have problems in writing the values to a file. But you could do the following:

- Redefine the inventory structure (figure 4-33 page 226) so that the title, the have value, and the want value are in a new structure that you can write to a file
- Use a second file for the names in the wait lists
- To restore the wait lists, you must know their lengths (i.e. when putting all the wait lists into one file, you need to know when one starts and the other ends)

1. The revised nodes look like:



2.6.3 And the revised class is:

```
// stock list
class StockList
{
public:
        …
private:
        // code for wait list of people waiting for certain titles
        struct WaitNode
        {       string   who;
                WaitNode *next;
        }; // end WaitNode

        // Stock item
        class StockItem
        {
        public:
                …
        private:
                string   title;
                int      have, want;
        }; // end StockItem

        // node for inventory list of stock items
        struct StockNode
        {       StockItem       item;
                WaitNode        *waitHead, *waitTail;
                StockNode       *next;
        }; // end StockNode

        // pointer to first stock node on the lsit
        StockNode *head;
}; // end StockList
```

3 Note that we doe not have a remove operation.
  3.1 If you set the have and want values to zero and have an empty wait list, when you terminate the program and write the values to the file, you can omit the nodes with no demand.
  3.2 When the program runs the next day, it will have removed the items automatically.

Chapter 4-5, The C++ Standard Template Library

1 It is common for a modern programming language, like C++, to provide classes that implement the more used ADTs.

1.1 Many of these classes are defined in the "Standard Template Library", or "STL".
1.2 The STL has a number of template classes that can be applied to nearly any type of data.

2 Many of the ADTs that are presented in this text have a corresponding ADT in the STL. For example, a "list" class is defined in the STL that is like the list class in the book.
2.1 So why if the ADT is in the STL, do we develop ADTs in class?
2.2 Reasons are:
- Developing simple ADTs provides a foundation for learning to develop other ADTs
- You may wind up working with a language that does not have any predefined ADTs. You need to have the ability to develop your own
- If the ADTs defined by the language are inadequate, you need to develop your own or improve the existing ones.

2.3 The STL gives the ADTs through three basic items: containers, algorithms, and iterators.
2.3.1 Containers hold other objects like a list.
2.3.2 Algorithms, like sorting algorithms, act on containers.
2.3.3 Iterators cycle through the contents of container.
2.4 Let's try STL containers and iterators.

2.4.1 Containers rely on the C++ "class template" construct.
2.4.1.1 Class templates allow you to develop a class and defer certain data-type information until you use the class.
2.4.1.2 We had used the "typedef" statement to define the actual ListItemType.
2.4.1.3 But with templates, this data type is left as a data-type parameter in the definition of the class.
2.4.1.4 The class definition would start with "template <typename T>" where the data-type parameter "T" would be the data type that the client would specify. For example:

```
template <typename T> class MyClass
{
public:
        MyClass();
        MyClass(T initialDate);

        void setData(T newData);
        T getData();
private:
        T theData;
}; // end MyClass
```

2.4.2 The client part would invoke it like so:

```
int main()
{
        MyClass<int> a;
        MyClass<double> b(5.4);

        a.setData(5);
        cout << b.getData() << endl;
```

2.4.3 Note how the declarations of a and b specify the data type.

2.4.4 STL containers actually use two data-type parameters.

2.4.4.1 The first is usually the data type for the items in the container.
2.4.4.2 The second is an "allocator" which manages the memory allocation for a container (use a class "allocator").

2.4.5 Iterators are a generalization of pointers.
2.4.5.1 They give the ability to cycle through the items in a container – like we used a point to traverse a linked list.
2.4.5.2 If you have an iterator called curr, you can access the item in the container that curr references by using the notation *curr.

2.4.6 STL classifies iterators into five categories – the categories determine the operations available with the iterator.
2.4.6.1 The STL container classes in this text support the "bidirectional iterator".
2.4.6.2 You can move to the next item or the previous one. The operation of:

```
++curr;
```

2.4.6.3 Would use the "++" increment operator to move the iterator to the next item in the container. To move the iterator back use the "—curr" or the decrement operator.

2.4.7 The container classes usually provide at least two methods that are useful with iterators.
2.4.7.1 The first method is:

```
iterator begin();
```

2.4.7.2 to put the iterator at the first item in the container.
2.4.7.3 The second method is:

```
iterator end();
```

2.4.7.4 to return a value that says whether you have reached the end of the container.
2.4.7.5 Note that this value is not NULL though it is used like the value NULL.
2.4.7.6 The following example uses the iterator:

```
list<int> myList;
list<int>::iterator curr;

// right now, the list is empty;
// start at the iterator at the beginning of myList
curr = myList.begin();
// test for empty list
if (curr == myList.end())
        cout << "The list is empty" << endl;

// insert five items into the list myList
for (int j = 0; j < 5; j++)
        // places item j at the front of the list
        curr = myList.insert(curr, j);

// now output each item in the list starting with the first item in the list; keep moving
// the iterator to the next item in the list until the end of the list is reached
for (curr = myList.begin(); curr != myList.end(); curr++)
        cout << *curr << " ";
cout << endl;
```

2.5 The rest of the section gives the example of the STL of the class "list".