Lists

1. Lists
   - efficient updating a sequence
   - linking nodes

2. List Operations
   - constructors
   - appending and traversing
   - chasing pointers

3. the STL list
   - constructor and iterator
   - find, erase, and insert
Lists

1. Lists
   - efficient updating a sequence
   - linking nodes

2. List Operations
   - constructors
   - appending and traversing
   - chasing pointers

3. the STL list
   - constructor and iterator
   - find, erase, and insert
efficient updates with lists


STL vector is an encapsulation of an array,
+ random access: $O(1)$ to retrieve element.
  − Insert or erase element of vector is $O(n)$ because we shift on average half of the number $n$ of elements in the vector.
+ We can insert at the current position an element in $O(1)$ via a linked list.
  − Retrieving element from list is $O(n)$.
The linked list data structure we consider is a self referential (or recursive) data structure.
Lists

1. Lists
   - efficient updating a sequence
   - linking nodes

2. List Operations
   - constructors
   - appending and traversing
   - chasing pointers

3. the STL list
   - constructor and iterator
   - find, erase, and insert
linking nodes

A linked list to store 3, 5, 4 looks like:

![Linked List Diagram]

The type `data` is generic, defined in

```cpp
template <typename data>
class List
```

The `next` is a pointer to `Node`. 
defining a struct

A recursive data structure:

```cpp
struct Node {
    T data; // T is template parameter
    Node *next; // pointer to next node

    Node(const T& item, Node* ptr = NULL) :
        data(item), next(ptr) {}
};
```

The struct is in the private part of a class List.

A list stores two pointers to a node:
to the first and the last element in a list.
namespace mcs360_list
{
    template <typename T>
    class List
    {
        private:

            struct Node { }; // omitted
            Node *first;
            Node *last;

        public:
            // operations
            // operations
    };
}
#include "mcs360_list_tc"
#endif
Most compilers (including \texttt{g++}) do not allow the definition of member functions of templated classes in a \texttt{.cpp} file.

Work around:

1. Definitions of member operations of the class with header file \texttt{mcs360_list.h} are placed in the file \texttt{mcs360_list.tc}.

2. We put \texttt{#include "mcs360_list.tc" before #endif} in the file \texttt{mcs360_list.h}.

Compromise: although member functions are not compiled separately, as developers we can have several files with different implementations using the same interface.
Lists

1. Lists
   - efficient updating a sequence
   - linking nodes

2. List Operations
   - constructors
   - appending and traversing
   - chasing pointers

3. the STL list
   - constructor and iterator
   - find, erase, and insert
constructors in mcs360_list.tc

namespace mcs360_list
{
    template <typename T>
    List<T>::List()
    {
        first = NULL;
        last = NULL;
    }

    template <typename T>
    List<T>::List(T item)
    {
        first = new Node(item);
        last = first;
    }
}
Lists

1. Lists
   - efficient updating a sequence
   - linking nodes

2. List Operations
   - constructors
   - appending and traversing
   - chasing pointers

3. the STL list
   - constructor and iterator
   - find, erase, and insert
appending to a list

Three cases to consider:

1. the list is empty
   test: `if(first == NULL)`

2. the list has one element
   test: `else if(first == last)`

3. the list has two or more elements
   test: `else case`
appending an item

template <typename T>
void List<T>::append(T item)
{
  if(first == NULL)
  {
    first = new Node(item);
    last = first;
  }
  else if(first == last)
  {
    first->next = new Node(item);
    last = first->next;
  } else
  {
    last->next = new Node(item);
    last = last->next;
  }
}
traversing a list

template <typename T>
void List<T>::write()
{
    Node *ptr = first;
    while(ptr != NULL)
    {
        std::cout << "->" << ptr->data;
        ptr = ptr->next;
    }
}
searching a list

The function `bool member(T item);` returns true if the `item` belongs to the list, returns false otherwise.

```cpp
template <typename T>
bool List<T>::member(T item)
{
    Node *ptr = first;
    while(ptr != NULL)
    {
        if(ptr->data == item)
            return true;
        ptr = ptr->next;
    }
    return false;
}
```
using lists

#include "mcs360_list.h"
#include <iostream>
using namespace std;
using namespace mcs360_list;

int main()
{
    List<int> L(3);

    L.append(5);
    L.write();
    cout << endl;

    return 0;
}
Lists

1. Lists
   - efficient updating a sequence
   - linking nodes

2. List Operations
   - constructors
   - appending and traversing
   - chasing pointers

3. the STL list
   - constructor and iterator
   - find, erase, and insert
To delete the data element 5:

previous->next = current->next;
delete current;
erasing an item

void erase(T item) removes an item

Cases to consider:

1. the list may be empty,
2. the list may have only one element,
3. we delete the last element.
template <typename T>
void List<T>::erase(T item)
{
    if(first != NULL)
    {  
        Node *current = first;

        if(current->data == item)
            first = first->next;
        else
        {
            Node *previous = first;
            current = first->next;
        }
    }
}
erase continued

```c
if (current != NULL)
{
    while ((current->data != item)
           && (current->next != NULL))
    {
        previous = current;
        current = current->next;
    }
    if (current->data == item)
    {
        previous->next = current->next;
        delete current;
    }
    if (previous->next == NULL)
    {
        last = previous;
    }
}
```
Lists

1. Lists
   - efficient updating a sequence
   - linking nodes

2. List Operations
   - constructors
   - appending and traversing
   - chasing pointers

3. the STL list
   - constructor and iterator
   - find, erase, and insert
the STL list

```cpp
#include <string>
#include <list>

int main()
{
    list<string> L;

    L.push_back("hello");
    L.push_back("how");
    L.push_back("are");
    L.push_back("you");

    return 0;
}
```
length and iterator

Number of elements in a list $L$:

```cpp
text n = L.size();
```

To write a list:

```cpp
void write ( list<string> L )
{
    for(list<string>::const_iterator i = L.begin();
        i != L.end(); i++)
    {
        cout << " " << *i;
        cout << endl;
    }
}
```

No `at` and no `[ ]` because $O(n)$. 

1 Lists
   - efficient updating a sequence
   - linking nodes

2 List Operations
   - constructors
   - appending and traversing
   - chasing pointers

3 the STL list
   - constructor and iterator
   - find, erase, and insert
finding elements

cout << "give a word : ";
string word; cin >> word;

list<string>::iterator i;

i = find(L.begin(), L.end(), word);

if(i != L.end())
    cout << "found \"" << *i << "\"" << endl;
else
    cout << "did not find \"" << word << "\"" << endl;
erase and insert

```cpp
list<string>::iterator j;
j = find(L.begin(), L.end(), "how");
L.erase(j);
cout << "after erasing \"how\" :";
write(L);

j = find(L.begin(), L.end(), "are");
L.insert(j, "who");
cout << "after inserting \"who\" :";
write(L);
```
Covered single linked lists of Chapter 4.

Exercises:

1. Add to `mcs360_list::List` a method `insert` to insert an item before the first element of a list.
2. Write a method `size()` for an object of `mcs360_list::List` that returns the length.
3. Give code for `insert(T item1, T item2)` on a `mcs360_list::List` to insert `item1` before `item2`. If `item2` does not occur in the list, then `insert` appends `item1` to the list.
4. Define a function to reverse the order of the elements of a `mcs360_list::List`. 