Linked List Class Template

1. Linked List Class Template

You created a linked list class for manipulating a linked list of integers. What if you want a linked list of Temperatures? Using the previous method, you would have to create another linked list class. A linked list *template* would allow you to create a linked list for a generic data type, and then the user simply specifies the actual data type to store in the data field of each node in the linked list.

The linked list template header file includes both the declaration and the definitions of the member functions. There cannot be a separate .cpp file for defining the member functions.

A template prefix, such as

template<**class** T>

is placed before the class declaration. The user identifier T in the above statement is a data type parameter. In the class definition, you would use T as the type rather than an actual type.

```
/*
 * Linked List Class Template LinkedList.h
 *
 * Copyright 2010-02-01 Enoch Hwang
 */
#ifndef __LINKEDLISTTEMPLATE__
#define __LINKEDLISTTEMPLATE__
#include <iostream>
using namespace std;

////////////////////////////////////////////////////////////////
// class template declaration for a ListNode
template <class T>
struct ListNode {
   T data; // the data field is of type T whatever type T is passed in
   ListNode* next;
};

////////////////////////////////////////////////////////////////
// class template declaration for a LinkedList
template <class T>
class LinkedList {
private:
   ListNode<T>* Head;
   ListNode<T>* Tail;     // for inserting node at the end of the list

public:
   LinkedList();
   void InsertNode(T d);
   void InsertInOrder(T d);
   ListNode<T>* SearchNode(T d);
   ListNode<T>* SearchNode(T d, ListNode<T>*& pptr);
   bool ChangeNode(T oldD, T newD);
```



```
   bool DeleteNode(T d);

   // there are two different methods to overload a friend operator
   // this is method 1 to overload the friend operator <<
   template <class U>
   friend ostream& operator<<(ostream& out, const LinkedList<U>& ll);

   // this is method 2 to overload the friend operator >>
   // for this method the definition must be done inline here in the declaration
   // this function inserts a new node at the end of the list
   friend istream& operator>>(istream& in, LinkedList& ll) {
      ListNode<T>* ptr = new(ListNode<T>);
      in >> ptr->data;
      ptr->next = NULL;
      if (ll.Tail == NULL) {    // empty list
         ll.Head = ptr;
         ll.Tail = ptr;
      } else {  // insert at end of list
         ll.Tail->next = ptr;
         ll.Tail = ptr;
      }
      return in;
   }
};

////////////////////////////////////////////////////////////////////
// class template definition. Must be in this .h file
//
// constructor initialize list to null
template <class T>
LinkedList<T>::LinkedList() {
   Head = NULL;
   Tail = NULL;
}

// insert new node with the given data at the head of the list
template <class T>
void LinkedList<T>::InsertNode(T d) {
   ListNode<T>* ptr = new(ListNode<T>);
   ptr->data = d;
   ptr->next = Head;
   Head = ptr;
   if (Tail == NULL) {    // empty list
      Tail = ptr;
   }
}

// insert new node with the given data in the list that keeps the list sorted
template <class T>
void LinkedList<T>::InsertInOrder(T d) {
   ListNode<T>* ptr = Head;
   ListNode<T>* pptr = NULL; // previous pointer
   // ...code is hidden below
}

// search for node in list
// returns pointer to node if found
// else returns NULL
```



```
template <class T>
ListNode<T>* LinkedList<T>::SearchNode(T d) {
   ListNode<T>* ptr;
   ptr = Head;
   while (ptr != NULL) { // traverse list
      if (ptr->data == d) { // found node
         return ptr; // return pointer to found node
      }
      ptr = ptr->next; // go to next node
   }
   return NULL;
}

// search for node in list
// returns pointer to node if found
// else returns NULL
// also returns the previous pointer in argument list
template <class T>
ListNode<T>* LinkedList<T>::SearchNode(T d, ListNode<T>*& pptr) {
   ListNode<T>* ptr = Head;
   pptr = NULL;
   while (ptr != NULL) { // traverse list
      if (ptr->data == d) { // found node
         return ptr; // return pointer to found node
      }
      pptr = ptr; // update previous pointer
      ptr = ptr->next; // go to next node
   }
   return NULL;
}

template <class T>
bool LinkedList<T>::ChangeNode(T oldD, T newD) {
   ListNode<T>* ptr = SearchNode(oldD); // search for node
   if (ptr) { // found node to change
      ptr->data = newD;
      return true;
   } else {
      return false;
   }
}

// delete node with the given data from the list
template <class T>
bool LinkedList<T>::DeleteNode(T d) {
   ListNode<T>* ptr;
   ListNode<T>* pptr; // previous pointer
   ptr = SearchNode(d, pptr); // search for node
   if (ptr != NULL) { // found node to delete
      if (ptr == Head) { // need to treat the first node differently
         Head = Head->next; // delete the head node
      } else { // delete a non-head node
         pptr->next = ptr->next;
      }
      delete ptr; // release memory
      return true;
   }
   else {
```



```
        return false;
    }
}

 template <class U>
 ostream& operator<<(ostream& out, const LinkedList<U>& ll) {
    ListNode<U>* ptr;
    ptr = ll.Head;
    while (ptr != NULL) {
        out << ptr->data << ", ";
        ptr = ptr->next;
    }
    cout << endl;
    return out;
 }

#endif
```

The main program…

```
/*
 * main.cpp
 *
 * Copyright 2010-02-01 Enoch Hwang
 */
#include <iostream>
#include "LinkedList.h"
#include "Temperature.h"
using namespace std;

int main() {
    // linked list of integers
    LinkedList<int> L1;
    L1.InsertNode(6);
    L1.InsertNode(5);
    L1.InsertNode(4);
    L1.InsertNode(3);
    L1.InsertNode(2);
    cout << "Enter an int? ";
    cin >> L1;
    cout << "Here's the linked list after inserting 6 integers..." << endl;
    cout << L1;
    L1.DeleteNode(4);
    L1.DeleteNode(1);
    cout << "Here's the linked list after deleting nodes 4 and 1..." << endl;
    cout << L1;

    // linked list of characters
    LinkedList<char> L2;
    L2.InsertNode('U');
    L2.InsertNode('S');
    L2.InsertNode('L');
    L2.InsertNode('@');
    L2.InsertNode('S');
    L2.InsertNode('C');
    cout << "Here's the linked list after inserting 6 characters..." << endl;
    cout << L2;
```



```
    L2.DeleteNode('C');
    cout << "Here's the linked list after deleting node C..." << endl;
    L2.ChangeNode('@', 'A');
    cout << "Here's the linked list after changing node @ with A..." << endl;
    cout << L2;

    // linked list of Temperatures
    LinkedList<Temperatures> L3;
    L3.InsertNode(Temperature(78,'F'));
    L3.InsertNode(Temperature(21,'C'));
    cout << "Here's the linked list of Temperatures..." << endl;
    cout << L3;

    return 0;
}
```


Sample output

```
Enter an int? 13
Here's the linked list after inserting 6 integers...
2, 3, 4, 5, 6, 13,
Here's the linked list after deleting nodes 4 and 1...
2, 3, 5, 6, 13,
Here's the linked list after inserting 6 characters...
C, S, @, L, S, U,
Here's the linked list after deleting node C...
Here's the linked list after changing node @ with A...
S, A, L, S, U,
```

2. Exercises (Problems with an asterisk are more difficult)

1. Using the above linked list class template, create a linked list of Temperatures with 5 nodes. Test out all the functions (<<, >>, search, change and delete).

2. Using the above linked list class template, create a linked list of WeatherStations with 5 nodes. Test out all the functions.

3. Add a member function called InsertInOrder(T d) to the above linked list class template that will insert a node with the data d into the linked list that will keep the list sorted.

