Example of Linked List as Template

Posted by Alejandro G. Carlstein Ramos Mejia on October 15, 2010 October 15, 2010 About Programming / C++

Example of Linked List as Template.

This code was compiled and tested on a Linux machine

NOTIFICATION: These examples are provided for educational purposes. Using this code is under your own responsibility and risk. The code is given 'as is'. I do not take responsibilities of how they are used.

List.cc:

```
#ifndef LIST_CC
#define LIST_CC

#include <iostream>

using namespace std;

template <typename T>
class Node {

 public:

  // Create a linked list node by setting its data to the input parameter
  // and the next node in the linked list to NULL (0)
  Node(const T& data_input);

  //The default constructor doesn't initialize the data members
  Node();

  //The default destructor doesn't initialize the data members
  ~Node();

  // Contains the data of the linked list node
  T data;

  // Contains a pointer to the next item in the linked list
  Node<T> *next;

  // Contains a pointer to the previous item in the linked list
  Node<T> *previous;

};

template <typename T>
class List {
```

```
 private:

  // Contains a pointer to the first node in the
  // linked list (the head of the list)
  Node<T> *head;

  // Contains a pointer to the last node in the
  // linked list (the head of the list)
  Node<T> *tail;

  int num_elements;

 public:

  // Create a Linked list by initializing it's head to NULL (0)
  List(void);

  List(const List<T>& list);

  // Delete all the elements in the linked list when the list is deleted
  ~List(void);

  bool isempty() const;

  // Takes an item as a parameter, appends that item to
  // the front of the list, and returns the list
  List<T> *addtofront(const T& data_input);

  // Takes an item as a parameter, appends that item to
  // the back of the list, and returns the list
  List<T> *addtoback(const T& data_input);

  // Returns a pointer to a copy of the first item in the list,
  // leaving it in the list
  Node<T> *getfirst() const;

  int size() const;

  // Takes no parameters, and returns a list containing
  // all but the first element of the list
  List<T> *getrest() const;

  // Print all the elements in the linked list
  void show();

};

// ------------------------------------
// ------------Node class--------------
// ------------------------------------

//
// Create a linked list node by setting its data to the input parameter
```

```
// and the next node in the linked list to NULL (0)
template <typename T>
Node<T>::Node(const T& data_input) {

   data = data_input;

    next = NULL;

    previous = NULL;

}

// Default constructor leaves data items unspecified
template <typename T>
Node<T>::Node() {

    next = NULL;

    previous = NULL;
}

// Default constructor leaves data items unspecified
template <typename T>
Node<T>::~Node() {
}

// --------------------------------------
// ------------List class---------------
// --------------------------------------

//
// Default constructor creates an empty list
//
template <typename T>
List<T>::List(void) {

    head = NULL;

 tail = NULL;

    num_elements = 0;

}

template <typename T>
List<T>::List(const List<T>& list){

    if ( list.head != NULL ){

  Node<T> *current_node = list.head;

  while (current_node != NULL){
```

```
   addtoback(current_node->data);

   current_node = current_node->next;

   num_elements++;
  }
 }
}

//
// Delete all the elements in the linked list when the list is deleted
//
template <typename T>
List<T>::~List(void) {

   Node<T> *next_node;

    Node<T> *current_node = head;

    while (current_node != NULL) {

        next_node = current_node->next;

        delete(current_node);

        current_node = next_node;

    }

}

template <typename T>
bool List<T>::isempty() const{
 return((num_elements < 1));
}

template <typename T>
int List<T>::size() const{
 return num_elements;
}

// Takes an item as a parameter, appends that item to
// the front of the list, and returns the list
template <typename T>
List<T> *List<T>::addtofront(const T& data_input){

    Node<T> *new_node;

    new_node = new Node<T>;

    new_node->data = data_input;

 // If list_head == 0 then assign new_node as the list_head
```

```
 // else point new_node->next to list_head. After make that
 // new node the list_head
    if (head == NULL) {

        head = new_node;

        tail = new_node;

    } else {

        new_node->next = head;

        head = new_node;

    }

    num_elements++;

 return (this);
}

// Takes an item as a parameter, appends that item to
// the back of the list, and returns the list
template <typename T>
List<T> *List<T>::addtoback(const T& data_input){

    Node<T> *new_node;

    new_node = new Node<T>;

    new_node->data = data_input;

    if (tail == NULL) {

        tail = new_node;

     if (head == NULL) {

         head = new_node;

     } else {

         new_node->next = head;

         head = new_node;

     }

    } else {

  tail->next = new_node;

  new_node->previous = tail;
```

```
  tail = new_node;

    }

    num_elements++;

    //delete(new_node);

 return (this);
}

// Returns a pointer to a copy of the first item in the list,
// leaving it in the list
template <typename T>
Node<T> *List<T>::getfirst() const{
 return (new Node<T>(head->data));
}

// Takes no parameters, and returns a list containing
// all but the first element of the list
template <typename T>
List<T> *List<T>::getrest() const{

 List<T> *rlist = new List<T>();

 Node<T> *current_node;

 if (head->next != NULL){

  current_node = head->next;

  while (current_node != NULL){

   rlist->addtoback(current_node->data);

   current_node = current_node->next;
  }
 }

 return (rlist);
}

//
// Print all the elements in the linked list
//
template <typename T>
void List<T>::show() {

    cout << 'List of ' << num_elements << ' elements: ' ;

    Node<T> *current_node = head;
```

```
    while (current_node != NULL) {

        cout << current_node->data << ' ';

        current_node = current_node->next;

    }

    cout << endl;
}

#endif
```

MainDriver.cpp

```
#include <iostream>
#include <string>
#include 'List.cc'

using namespace std;

template <typename T>
List<T> *reverse(List<T>& list);

int main (int argc, char *argv[]) {

 List<string> my_list;

 List<string> *my_list_2;

 cout << endl << 'BUILD ORIGINAL---------------------------' << endl;

 my_list.addtoback('one(1)');
 my_list.addtofront('nine(9)');
 my_list.addtoback('eight(8)');

 my_list.show();

 cout << endl << 'REVERSE ORIGINAL---------------------------' << endl;

 my_list_2 = reverse<string>(my_list);

 cout << endl << 'New: ';

 my_list_2->show();

 delete my_list_2;

 return 0;
}

// Write a templated reverse() function
// (not a member function of your class, but instead
```

```
// a standalone templated function)
// that operates on a doubly linked list object
// (that you implemented for Part 1).
// Your reverse function should take as input a
// doubly linked list object,
// and return a copy of the list,
// but in reverse order.

// Your function must be recursive,
// and must use the four functions above.

// A reversed list can be built by taking an element off one side,
// reversing the list without that element in it,
// and then putting that element back into the list,
// on the other side.... use that reasoning to design your recursive
// reverse function.

template <typename T>
List<T> *reverse(List<T>& list){

 if (list.size() < 1){

  List<T> *rlist = new List<T>;

  return (rlist);

 }else{

  Node<T> *node;

  List<T> *rlist = new List<T>;

  node = list.getfirst();

  rlist = list.getrest();

  rlist = reverse<T>(*rlist);

  rlist->addtoback(node->data);

  delete (node);

  return (rlist);
 }

}
```

If you encounter any problems or errors, please let me know by providing an example of the code, input, output, and an explanation. Thanks.