//Barthelemy Peter

// Class implementation file: linklist.cpp

// Implementation of class defined in linked\_list.h

// This class can be used to create and maintain an unsorted

// linked list of data as defined in element.h. The list node

// form is defined in list\_node.h (implemented in list\_node.cpp).

#include <assert.h> // used to test list consistancy

#include "linked\_list.h" // this class header file

#include "list\_node.h"

// Class constructor - sets the object that the client is declaring

// to be the empty list

list::list()

{

// list is empty

// to be safe initialize current

// & previous

head = NULL;

previous = NULL;

current= NULL;

}

// Copy constructor - copies lst to the new object that the client

// is declaring

/\* list::list(list& lst)

{ // start of copy constructor

// Create a temporary pointer to follow the parameter lst

list\_node \*temp = new list\_node;

temp = lst.head;

head = temp;

previous = current;

current = head;

if (temp != NULL) // there is a list to copy

{

// set receiver's current to point to new node

current->put\_next(temp);

// set new node's data to lst's data

temp->put\_item();

// initialize new node's next field

temp->get\_next()=NULL;

// receiver's head pointer must point to new node

head->put\_next(temp);

// move temp to next node in lst

temp= temp->get\_next();

// as long as there are more nodes in lst

while (temp != NULL)

{

// make current node's next field point to a new node

current->put\_next(new list\_node);

// make current point to the new node

current= current->get\_next();

// set current node's data

current->put\_item(lst);

// set current node's next field

current->get\_next()=NULL;

// advance temp to next node

temp= temp->get\_next();

} // end while

} // end if

else // there is no list to copy, so head is NULL:

head = NULL;

}\*/ // end of copy constructor

// Destructor - removes all the receiver's list\_node's from memory

// freeing memory space for other uses

list::~list()

{

// make all pointers point to the first node

head = NULL;

previous = head;

current = head;

// as long as there are more nodes to delete

while (current != NULL)

{

// move current to point to next node

previous = current;

current= current->get\_next();

// remove the previous node from memory

delete previous;

// advance previous to next node

previous = current->get\_next();

}

// set all pointers to NULL - list is empty

} // end of destructor

// Add a new item to the end of the list

void list::append(const element &item)

{

list\_node \*newNode = new list\_node(item);

if (empty()) // then make a new node and point head at it

{

head = newNode;

current = head;

}

else // must find end of list and addd new item

{

// start at beginning of list

// not at end of list keep moving

while (current->get\_next() != NULL)

current = current -> get\_next();

// now current points at last node in list

// make next field point to a new node

current->put\_next(newNode);

// make current point to new node

// previous->put\_next(current);

current = newNode;

// set current node to new value

// current->put\_item(item);

// mark new node as last node in list

newNode->put\_next(NULL);

}

}

// recursive insert - recalls itself passing a pointer to the next node (list)

// this is private to the class

void insrt(list\_node \*lst, const element &item)

{

if (lst != NULL) // then not at the end of the list - terminal condition

if (item > lst->get\_item()) // then the new item goes farther down the l$

{

insrt(lst->get\_next(), item); // recursive call

// on return put the new item in the list

if ((lst->get\_next() == NULL) ||

(item < lst->get\_next()->get\_item())) // then new item goes here

{ // create a new node and hook it up

list\_node \*current = new list\_node;

current->put\_item(item);

current->put\_next(lst->get\_next());

lst->put\_next(current);

}

// otherwise item has been inserted previously so just return again

}

}

// public insert function

void list::insert(const element &item)

{

list\_node \* newNode = new list\_node(item);

if (empty()) // then new item starts the list

{

head = newNode;

current = head;

previous = nullptr;

}

else // it may be a new first item in the list

if (head->get\_item() >= item) // then new item is the new first

{

newNode->put\_next(head);

head = newNode;

// temp->put\_next(head);

}

else // the item goes somewhere further down the list.

previous->put\_next(current);

current->put\_item(item);

// Implement utility functions

// Test for empty list

bool list::empty()

{

if(head == NULL)

return true;

else

return false;

// if head is NULL, list is empty

}

// Test for end of list

bool list::at\_end()

{

// if list is empty, must be at end OR

// if current point to last node then at end of list

if(empty() || current->get\_next()== NULL)

return true;

else if (current->get\_next() == NULL)

return true;

else

return false;

//return ((head == NULL) || ((current != NULL) &&

//(current->get\_next() == NULL)));

}

void list::move\_to\_next()

{

// check current so that illegal reference is not made

if (current != NULL)

{

current = current ->get\_next();

// advance cuurent

}

}

void list::move\_to\_start()

{

current = head;

}

element list::get\_current()

{

assert(current != NULL);

return current->get\_item();

}

// End of file linked\_list.cpp

list\_node::list\_node()

{

data ="";

next = NULL;

}

list\_node::list\_node(const element &item)

{

data = item;

next = NULL;

}

void list\_node::put\_item(const element &item)

{

data = item;

}

void list\_node::put\_next(list\_node \*ptr)

{

next = ptr;

}

element list\_node::get\_item()

{

return data;

}

list\_node \*list\_node::get\_next()

{

return next;

}