**C++ Part II (INFO1-CE9265) Spring 2015 – Homework 7**

Clement Chan

**Linked List Questions**

Linked\_List.cpp

#include <iostream>

template<typename T>

class SList;

//Defining the Iterator type

template <class NodeType>

class Iterator {

private:

friend class SList <typename NodeType :: value\_type>; //Creating a friend class with Single Linked List

NodeType \*Nodeptr;

Iterator(NodeType \*the\_Nodeptr) : Nodeptr(the\_Nodeptr){};

public:

void operator++(){Nodeptr = Nodeptr -> next;}; //Implement ++ to check on the next value

void operator++(int){Nodeptr = Nodeptr -> next;};

bool operator!=(Iterator<NodeType> value){return !(Nodeptr == value.Nodeptr);};

bool operator==(Iterator<NodeType> value){return (Nodeptr == value.Nodeptr);};

typename NodeType::value\_type operator\*(){return Nodeptr -> data;};

Iterator<NodeType> operator+(int size)

{

Iterator<NodeType> iter = \*this;

for (int i = 0; i < size; i++)

{

if(iter.Nodeptr)

++iter;

else

break;

}

return iter;

}

};

template <typename T>

class Node

{

private:

T data;

Node<T> \*next;

friend class SList<T>; //Create friend class with Single Linked List

friend class Iterator<Node<T> >; // Create friend class with Iterator

Node() : next(0) {};

Node(T the\_data) : data(the\_data), next(0) {};

Node(T the\_data, Node<T> \*the\_next) : data(the\_data), next(the\_next){};

Node(Node<T> \*the\_next) : next(the\_next) {};

public:

typedef T value\_type;

};

template <typename T>

class SList

{

private:

Node<T> \*head; //Define the type node as 'head', which is empty

public:

typedef Iterator<Node<T> > iterator; //Define the iterator class that is subnested into Linked List

typedef T valuetype;

//Default Constructor

SList() : head(0){std::cout<<"Initiating the Constructor of Linked List ...

"<<std::endl;}

//Destructor, Delete the nodes created one by one

~SList()

{ std::cout<<"Initiating Destructor of Linked List ... " << std::endl;

if(head)

{

Node<T> \*iteration = head;

while(iteration != 0)

{

Node<T> \*temp = iteration; // Define temp equal to iteration

iteration = iteration -> next;

delete temp;

}

}

}

iterator begin() {return iterator(head);} //Begin returns the node at the front, utilizing iterator parameterized constructor

iterator end() {return iterator(0);}

//Insert data at the beginning

void insert(T data)

{

if(head){

Node<T> \*temp = new Node<T>(data);

temp -> next = head;

head = temp;}

else{

head = new Node<T>(data);

}

}

bool erase(iterator &the\_Node)

{

//First, Check the case where the\_Node is the same as head

if(the\_Node.Nodeptr == head)

{

head = head -> next;

delete the\_Node.Nodeptr;

return true;

}

else //If not, we will iterate forward to find the node that is the same as head

{

for(Node<T> \*iteration = head; iteration-> next; iteration = iteration -> next){

if(iteration -> next == the\_Node.Nodeptr){

iteration -> next = the\_Node.Nodeptr -> next;

delete the\_Node.Nodeptr;

return true;

}

}

}

return false;

}

};

int main()

{

SList<int> List\_int;

List\_int.insert(3);

List\_int.insert(8);

List\_int.insert(10);

List\_int.insert(20);

List\_int.insert(100);

//Print out the list

std::cout << " " << std::endl;

std::cout << "The List is : " << std::endl;

for(SList<int>::iterator iteration = List\_int.begin(); iteration != List\_int.end();

iteration++)

{

std::cout << (\*iteration) << std::endl;

}

//Delete the third node

SList<int>::iterator temp = List\_int.begin() + 2;

List\_int.erase(temp);

std::cout << " " << std::endl;

std::cout << "The List after removal of data is : " << std::endl;

for(SList<int>::iterator iteration = List\_int.begin(); iteration != List\_int.end();

iteration++)

{

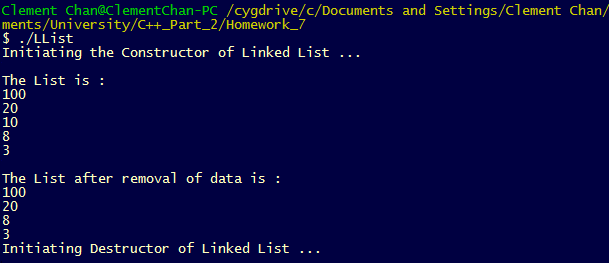
std::cout << (\*iteration) << std::endl;

}

return 0;

}

**Output**

****