```
#include <iostream>
#include "List.h"

/
************************************************************************
*************************
Oscar Maldonado
005487270
Spring 2019
CSE330
LAB05
************************************************************************
*************************/

template <typename Object>
void List<Object>::List::init()//init()
{
theSize = 0;
head = new Node;
tail = new Node;
head->next = tail;
tail->prev = head;
}

template <typename Object>
struct List<Object>::List::Node//Node struct
{
Object data;
Node* next;
Node* prev;

Node(Object & d,Node* n = nullptr, Node* p = nullptr) : data{d},
next{n}, prev{p} {}
Node(Object && d,Node* n = nullptr, Node* p = nullptr) :
data{std::move(d)}, next{std::move(n)}, prev{std::move(p)}{}
};

template <typename Object>
class List<Object>::List::const_iterator //iterator class
{
Node* current;
friend class List<Object>;
Object& retrieve() const
{
return current->data;
}
const_iterator(Node* p) : current{p} {}

public:
```

```
const_iterator() : current{nullptr}{}

Object& operator*() const {return retrieve();

const_iterator& operator++()
{
current = current->next;
return *this;
}
const_iterator operator++(int)
{
const_iterator copy = this;
*(++this);
return copy;
}

const_iterator& operator--()
{
current = current->prev;
return *this;
}
const_iterator operator--(int)
{
const_iterator copy = *this;
--(*this);
return copy;
}

bool operator==(const const_iterator & rhs) const{ return current ==
rhs.current; }
bool operator!=(const const_iterator & rhs) const { return current !=
rhs.current; }
                                         head->next = tail;
tail->prev = head;
}

template <typename Object>
struct List<Object>::List::Node//Node struct
{
Object data;
Node* next;
Node* prev;

Node(Object & d,Node* n = nullptr, Node* p = nullptr) : data{d},
next{n}, prev{p} {}
Node(Object && d,Node* n = nullptr, Node* p = nullptr) :
data{std::move(d)}, next{std::move(n)}, prev{std::move(p)}{}
};

template <typename Object>
```

```
class List<Object>::List::const_iterator //iterator class
{
Node* current;
friend class List<Object>;
Object& retrieve() const
{
return current->data;
}
const_iterator(Node* p) : current{p} {}

public:

const_iterator() : current{nullptr}{}

Object& operator*() const {return retrieve();

const_iterator& operator++()
{
current = current->next;
return *this;
}
const_iterator operator++(int)
{
const_iterator copy = this;
*(++this);
return copy;
}

const_iterator& operator--()
{
current = current->prev;
return *this;
}
const_iterator operator--(int)
{
const_iterator copy = *this;
--(*this);
return copy;
}

bool operator==(const const_iterator & rhs) const{ return current ==
rhs.current; }
bool operator!=(const const_iterator & rhs) const { return current !=
rhs.current; }

};

template <typename Object>
class List<Object>::interator : public List<Object>::const_iterator//
iterator class
```

```
{
iterator(){}
const Object& operator*()
{
this->current = this->current->next;
return *this;
}

iterator operator++(int)
{
iterator copy = *this;
++(*this);
return copy;
}

iterator& operator--()
{
this->current = this->current->prev;
return *this;
}

iterator operator--(int)
{
iterator copy = *this;
--(*this);
return copy;
}
private:
iterator(Node* obj) : List<Object>::const_iterator{obj}{}
friend class List<Object>;
};
template <typename Object>
List<Object>::List(const List & rhs) : theSize{rhs.theSize},
head{rhs.head} , tail{rhs.tail}{} // copy constructor
template <typename Object>
List<Object>::List(List&& rhs) : theSize{std::move(rhs.theSize)},
head{std::move(rhs.head)}, tail{std::move(rhs.tail)} {} //move
constructor
template <typename Object>
List<Object>::~List()
{
clear();
head = nullptr;
tail = nullptr;
}
template <typename Object>
List<Object>::List& List<Object>::operator=(const List& obj)
{
List copy = obj;
std::swap(*this,copy);
```

```
return *this;
}
template <typename Object>
List<Object>::List& List<Object>::operator=(List&& obj)
{
std::swap(theSize,obj.theSize);
std::swap(head,obj.head);
std::swap(tail,obj.tail);
return *this;
}
template <typename Object>
typename List<Object>::iterator List<Object>::insert(iterator
itr,const Object& x);//insert
{
Node* p = itr.current;
theSize++;
return {p->prev = p->prev->next = new Node(x,p->prev,p)};
}
template <typename Object>
typename List<Object>::iterator List<Object>::insert(iterator itr,
Object&& x)//move insert
{
node* p =itr.current;
theSize++;
return {p->prev = p->prev->next = new Node(x,std::move(p->prev),p)};
}
template <typename Object>
typename List<Object>::iterator List<Object>::erase(iterator itr)//
erase
{
Node* p = itr.current;
iterator retVal{p->next};
p->prev->next = p->next;
delete p;
theSize--;

return retVal;
}
template <typename Object>
typename List<Object>::iterator List<Object>::erase(iterator from,
iterator to)//erase range
{
for(iterator itr = from; itr!= to;)
        itr = erase(itr);
return to;
}
template <typename Object>
void List<Object>::clear()
{
while(!empty())
```

```
        pop_front();
}
```