```cpp
#pragma once

#include "DoublyLinkedList.h"
#include "DoublyLinkedListIterator.h"

template<typename T>
class List
{
private:
    using Node = typename DoublyLinkedList<T>::Node;

    Node fHead; // first element
    Node fTail; // last element
    size_t fSize; // number of elements

public:

    using Iterator = DoublyLinkedListIterator<T>;

    List() noexcept
    {}// default constructor (2)

    // copy semantics
    List( const List& aOther )
    {
        *this = aOther;
    }// copy constructor (10)
    List& operator=( const List& aOther )
    {
        // Clear the existing list
        fHead = nullptr;
        fTail = nullptr;
        fSize = 0;

        // Iterate over the elements of the source list and push
them into the destination list
        for (const auto& element : aOther)
        {
            push_back(element);
        }

        return *this;
    }// copy assignment (14)

    // move semantics
    List( List&& aOther ) noexcept
    {
        *this = aOther;
    }// move constructor (4)
    List& operator=( List&& aOther ) noexcept
    {
        fHead = nullptr;
```

```
        fTail = nullptr;
        fSize = 0;

        for (const auto& element : aOther)
        {
            push_back(element);
            aOther.remove(element);
        }

        return *this;
    }// move assignment (8)
    //void swap( List& aOther ) noexcept; // swap elements (9)

    // basic operations
    size_t size() const noexcept
    {
        return fSize;
    }// list size (2)

    template<typename U>
    void push_front( U&& aData )
    {
        Node newNode =
DoublyLinkedList<T>::makeNode(std::forward<U>(aData));
        if (fSize == 0)
        {
            fHead = newNode;
            fTail = newNode;
        }
        else
        {
            //not sure if i need this, but this is in the case
that the chain is empty
            newNode->fNext = fHead;
            fHead->fPrevious = newNode;
            fHead = newNode;
        }
        fSize++;

    }// add element at front (24)

    template<typename U>
    void push_back( U&& aData )
    {
        Node newNode =
DoublyLinkedList<T>::makeNode(std::forward<U>(aData));

        if (fSize == 0)
        {
            fHead = newNode;
            fTail = newNode;
        }
```

```
        else
        {
            newNode->fPrevious = fTail;
            fTail->fNext = newNode;
            fTail = newNode;
        }

        fSize++;
    }// add element at back (24)

    void remove( const T& aElement ) noexcept
    {
        Node lMatch = fHead;
        while (lMatch != nullptr)
        {
            if (lMatch->fData == aElement)
            {
                Node nextNode = lMatch->fNext;
                Node previousNode = lMatch->fPrevious.lock();

                if (lMatch == fHead)
                {
                    fHead = nextNode;
                }
                if (lMatch == fTail)
                {
                    fTail = previousNode;
                }

                lMatch->isolate();

                fSize--;
                return;
            }
            else
            {
                lMatch = lMatch->fNext;
            }
        }
    }// remove element (36)

    const T& operator[]( size_t aIndex ) const
    {
        Node lnode = fHead;
        if (aIndex == 0)
        {
            return fHead->fData;
        }
        for (int i = 0; i < aIndex; i++)
        {
            lnode = lnode->fNext;
        }
```

```
        return lnode->fData;
    }
    // list indexer (14)

    // iterator interface
    Iterator begin() const noexcept
    {
        return Iterator(fHead, fTail).begin();
    }
    Iterator end() const noexcept
    {
        return Iterator(fHead, fTail).end();
    }
    Iterator rbegin() const noexcept
    {
        return Iterator(fHead, fTail).rbegin();
    }
    Iterator rend() const noexcept
    {
        return Iterator(fHead, fTail).rend();
    }

};
```