Documentacion y explicacion de los codigos "DLList.cpp" y "DLList.h"

DLList.h

Primero se empieza con el archivo de tipo header (.h) debido a que será utilizada más adelante en el codigo principal.

```
#include <utility>
#include <iostream>
```

Primero se mandan a llamar funciones de utility y iostream.

```
template <typename Object>
class DLList {
private:
    struct Node {
        Object data;
        Node *prev;
        Node *next;

        Node(const Object &d = Object{}, Node *p = nullptr, Node *n = nullptr);
        Node(Object &&d, Node *p = nullptr, Node *n = nullptr);
```

`template <typename Object>` Primero se crea un class template con el nombre "Object".

`class DLList` Despues se manda a crear una clase con el nombre de "DLList".

Despues se le definen nodos privados a esta clase cada uno siendo:

`Object data;` define "data" que es de tipo Object.

`Node *prev;` Se encargara de un puntero al nodo anterior en una lista de tipo "node".

`Node *next;` Se encargara de un puntero al nodo siguiente en una lista de tipo "nodo".

`Node(const Object &d = Object{}, Node *p = nullptr, Node *n = nullptr);` Es un constructor que se encargar de darles datos default con "data" teniendo "Object" como default, "prev" con un "nullptr" y "next" con un "nullptr".

`Node(Object &&d, Node *p = nullptr, Node *n = nullptr);` Es un constructor que se encarga de tomar un rvalue como referencia para el "Object", le da a "data" un valor de "Object" como default, "prev" con un "nullptr" y "next" con un "nullptr".

```
public:
    class const_iterator{
    public:
        const_iterator();
        const Object &operator*() const;
        const_iterator &operator++();
        const_iterator operator++ (int);
        bool operator== (const const_iterator& rhs) const;
        bool operator!= (const const_iterator& rhs) const;

    protected:
        Node* current;
        Object& retrieve() const;
        const_iterator(Node *p);

        friend class DLList<Object>;
```

Aqui se crea una clase publica llamada "const_iterator"

En el apartado publico tenemos:

`const_iterator();` Se encarga como Constructor para la clase del mismo nombre.

`const Object &operator*() const;` Es una operadora de referencia que devuelve una referencia al "Object" a la posicion del iterador actual.

`const_iterator &operator++();` Es un operador que sirve como prefijo de aumento para mover al iterador al siguiente elemento de la lista (este es llamado para antes del incremento).

`const_iterator operator++ (int);` Es un operador que sirve como prefijo de aumento para mover al iterador al siguiente elemento de la lista (este es llamado para despues del incremento).

`bool operator== (const const_iterator& rhs) const;` Se encarga de comparar a dos iteradores para checar si son iguales.

`bool operator!= (const const_iterator& rhs) const;` Se encarga de comparar a dos iteradores para checar si son diferentes.

Ahora para el apartado "protejido" (este hace que los miembros son accesibles por las clases pero no para el resto) tenemos.

`Node* current;` Es un puntero al nodo actual de la iteracion.

`Object& retrieve() const;` Se encarga de recuperar al Object a la posicion actual del iterador.

`const_iterator(Node *p);` Es un constructor que toma un puntero a un nodo como parametro, debido a que esta en protejido se evita la creacion arbitraria de const_iterator de objetos fuera de la clase.

`friend class DLList<Object>;` Esto indica que la clase "DLList" tenga acceso a los miembros protegidos de esta clase, esto para que esta clase pueda crear y manipular objetos del const_iterator.

```
class iterator : public const_iterator {
public:
iterator();
Object& operator*();
const Object& operator*() const;
iterator & operator++ ();
iterator &operator--();
iterator operator++ (int);
iterator operator-- (int);
iterator operator+ (int steps) const;

    protected:
        iterator(Node *p);

        friend class DLList<Object>;
```

`class iterator : public const_iterator` Este se dedica en definir una derivada de las clase del "iterator" que herera los miembros publicos del const_iterator.

`iterator();` Es el constructor en default del "iterator".

`Object& operator*();` Es un operador que regresa como referencia al "Object" de la posicion actual del iterador.

`const Object& operator*() const;` Una version "const" del operador anterior.

`iterator & operator++ ();` Es un prefijo de aumento para mover al iterador al siguiente elemento de la lista. (Este es llamado antes del incremento)

`iterator &operator--();` Es un prefijo de disminucion para mover al iterador al elemento anterior de la lista. (Este es llamado antes de la disminucion)

`iterator operator++ (int);` Es un prefijo de aumento para mover al iterador al siguiente elemento de la lista. (Este es llamado despues del incremento)

`iterator operator-- (int);` Es un prefijo de disminucion para mover al iterador al elemento anterior de la lista. (Este es llamado despues de la disminucion).

`iterator operator+ (int steps) const;` Se encarga de crear un nuevo iterador llamado "steps" y lo posiciona adelante del iterador actual.

Ahora pasando a los miembros protegidos.

`iterator(Node *p);` Es un constructor que toma un puntero al node como parametro, debido a que esta protegido se evita la creacion arbitraria de un objeto del iterador fuera de la clase.

`friend class DLList<Object>;` Esto indica que la clase "DLList" tenga acceso a los miembros protegidos de esta clase, esto para que esta clase pueda crear y manipular objetos del iterador.

```
public:
    DLList();
    DLList(std::initializer_list<Object> initList);
    ~DLList();
    DLList(const DLList &rhs);
    DLList &operator=(const DLList &rhs);
    DLList(DLList &&rhs);
    DLList &operator=(DLList &&rhs);

    iterator begin();
    const_iterator begin() const;
    iterator end();
    const_iterator end() const;

    int size() const;
    bool empty() const;

    void clear();

    Object &front();
    const Object &front() const;
    Object &back();
    const Object &back() const;

    void push_front(const Object &x);
    void push_front(Object &&x);
    void push_back(const Object &x);
    void push_back(Object &&x);

    void pop_front();
    void pop_back();

    iterator insert(iterator itr, const Object &x);
    iterator insert(iterator itr, Object &&x);

    void insert(int pos, const Object &x);
    void insert(int pos, Object &x);

    iterator erase(iterator itr);
    void erase(int pos);
    iterator erase(iterator from, iterator to);

    void print() const;
```

`DLList();` Es un constructor que crea una lista doblemente enlazada.

`DLList(std::initializer_list<Object> initList);` Un constructor que toma una lista inicializadora para iniciar una lista doblemente enlazada con los elementos de la lista.

`~DLList();` Se encarga de destruir y limpiar elementos de la lista doblemente enlazada.

`DLList(const DLList &rhs);` Copia el constructor para crear una copia de una lista doblemente enlazada.

`DLList &operator=(const DLList &rhs);` Se encarga de copiar la asignacion de un operador para asignarle los elementos de una lista doblemente enlazada a otra.

`DLList(DLList &&rhs);` Se encarga de mover un constructor para crear una nueva lista doblemente enlazada usando datos de otras listas.

`DLList &operator=(DLList &&rhs);` Se encarga de mover la asignacion de un operador para mover elementos de una lista doblemente enlazada a otra robando datos externos.

`iterator begin();` Se encarga de regresar un iterador apuntando al primer elemento de la lista.

`const_iterator begin() const;` Regresa un const_iterator apuntando al primer elemento de la lista (Se usa para objetos de tipo const).

`iterator end();` Se encarga de regresar un iterador apuntando en frente del ultimo elemento de la lista.

`const_iterator end() const;` Regresa un const_iterator apuntando en frente del ultimo elemento de la lista (Se usa para objetos de tipo const).

`int size() const;` Se encarga de imprimir el numero de elementos en una lista.

`bool empty() const;` Se encarga de mostrar true o false en caso de que la lista este vacia o no.

`void clear();` Elimina todos los elementos de la lista.

`Object &front();` Se encarga de imprimir como referencia al primer elemento de la lista.

`const Object &front() const;` Se encarga de imprimir como referencia al primer elemento de la lista como constante.

`Object &back();` Se encarga de imprimir una referencia del ultimo miembro de una lista.

`const Object &back() const;` Se encarga de imprimir una referencia del ultimo miembro de una lista como constante.

`void push_front(const Object &x);` Se encarga de agregar un elemento en frente de la lista.

`void push_front(Object &&x);` Se encarga de agregar un elemento en frente de la lista usando semanticas de movimiento.

`void push_back(const Object &x);` Se encarga de agregar un elemento al final de la lista.

`void push_back(Object &&x);` Se encarga de agregar un elemento al final de la lista usando semanticas de movimiento.

`void pop_front();` Se encarga de eliminar el primer elemento de la lista.

`void pop_back();` Se encarga de eliminar el último elemento de la lista.

`iterator insert(iterator itr, const Object &x);` Se encargará de insertar un elemento antes del elemento apuntado por el iterador.

`iterator insert(iterator itr, Object &&x);` Se encargará de insertar un elemento antes del elemento apuntado por el iterador usando semanticas de movimiento.

`void insert(int pos, const Object &x);` Se encargara de insertar un elemento en una posicion especifica de la lista.

`void insert(int pos, Object &x);` Se encargara de insertar un elemento en una posicion especifica de la lista usando semanticas de movimiento.

`iterator erase(iterator itr);` Se encarga de eliminar un elemento apuntado por el iterador y regresa un iterador al siguiente elemento.

`void erase(int pos);` Se encarga de eliminar un elemento a una posicion especificada de la lista.

`iterator erase(iterator from, iterator to);` Se encarga de eliminar elementos dentro de un rango para luego regresar un iterador al elemento que procede del ultimo elemento eliminado.

`void print() const;` Se encarga de imprimir los elementos de una lista.

Y ya por ultimo los elementos privados de la clase que son:

```
private:
int theSize;
Node *head;
Node *tail;

    void init();
    iterator get_iterator(int pos);
```

primero se declaran un int llamado theSize y dos nodos con punteros llamados "head" y "tail".

`void init();` Se encarga de enicializar una lista doblemente enlazada para preparar el estado inicial de la lista.

`iterator get_iterator(int pos);` Se encarga de regresar un iterador a la posicion del elemento especificada de la lista. Esto para que sirva como apoyo a otras funciones que requieran iterar sobre esta lista.

Ya por último se agrega el "#include "DLList.cpp" para asegurarse que el header file solo sean incluidos una sola vez al transladarse a otro archivo. esto requiere del ifndef que se encuentra al inicio de este archivo.

DLList.cpp

Ahora pasamos al archivo .cpp el cual primero se encarga de incluir el header dentro del archivo con "#include "DLList.h" para poder usar todas las funciones dentro de este. luego tenemos lo siguiente:

```
template<typename Object>
DLList<Object>::Node::Node(const Object &d, Node *p, Node *n)
        : data{d}, prev{p}, next{n} {}
```

Para empezar todos los codigos que lleven `"template<typename Object>"` es porque son parte del class template DLList para evitar tenes que escribir el mismo significado una y otra vez.

`DLList<Object>::Node::Node(const Object &d, Node *p, Node *n)` Especifica al constructor por el struct del nodo. el parametro de "const Object &d" es el dato almacenado en el nodo y el Node *p es el puntero del nodo anterior mientras que el Node *n es el puntero del siguiente nodo.

`data{d}, prev{p}, next{n} {}` Se encarga de inicializar los miembros "data", "prev" y "next" del struct "node" con los valores "d","p","n" respectivamente antes de entrar al cuerpo del constructor.

```
template<typename Object>
DLList<Object>::Node::Node(Object &&d, Node *p, Node *n)
        : data{std::move(d)}, prev{p}, next{n} {}
```

Este codigo hace lo mismo que el anterior con la diferencia de que en:

`data{std::move(d)}, prev{p}, next{n} {}` El "std::move(d)" es usado para mover "d" al nodo "data" convirtiendolo en un rvalue.

```
template<typename Object>
DLList<Object>::const_iterator::const_iterator() : current{nullptr} {}
```

`DLList<Object>::const_iterator::const_iterator()` Se encarga de especificar el constructor default para la clase "const_iterator".

`current{nullptr} {}` Inicia al miembro "current" de la clase "const_iterator" con "nullptr", esto causa que el iterador apunte a nada, lo que significa que no será válido hasta que se le asigne un valor valido.

```
template<typename Object>
const Object &DLList<Object>::const_iterator::operator*() const {
return retrieve();
```

`const Object &DLList<Object>::const_iterator::operator*() const` Se encarga de especificar la funcion "operator*()" dentro de la clase "const_iterator".

`return retrieve();` Se encarga de llamar la funcion "retrieve()" para tener al objeto referenciado por el iterador y regresarlo, debido a que "retrieve()" es una funcion privada de la clase "const_iterator" que es usado para acceder al dato del nodo que el iterador apunta.

```
template<typename Object>
typename DLList<Object>::const_iterator &DLList<Object>::const_iterator::operator++() {
    current = current->next;
    return *this;
```

`typename DLList<Object>::const_iterator &DLList<Object>::const_iterator::operator++()` Debido a que tiene el prefijo "typename" eso significa que es un tipo dependiente y con el "&" despues de la funcion indica que este regresa una referencia al iterador.

`current = current->next;` Se encarga de avanzar el iterador al siguiente nodo de la lista.

`return *this;` Esto se encarga de regresar como referencia al mismo iterador despues de ser incrementado, esto permite anclar las operaciones de incremento.

```
template<typename Object>
typename DLList<Object>::const_iterator DLList<Object>::const_iterator::operator++(int) {
const_iterator old = *this;
++(*this);
return old;
```

`typename DLList<Object>::const_iterator DLList<Object>::const_iterator::operator++(int)` Esto se encargara del incremento despues de la funcion del operador que toma al "int" como parametro y regresa un nuevo objeto del iterador.

`const_iterator old = *this;` Se encarga de crear una copia del iterador actual y los almacena en la variable "old".

`++(*this);` Esto se encarga de incrementa el iterador actual al siguiente nodo de la lista usando el operador de pre-incremento.

`return old;` Se encarga de regresar la copia del iterador que se formó antes del incremento.

```
template<typename Object>
bool DLList<Object>::const_iterator::operator==(const const_iterator &rhs) const {
    return current == rhs.current;
```

`bool DLList<Object>::const_iterator::operator==(const const_iterator &rhs) const` Es un booleano que se encarga de definir si la clase "const_iterator" es igual a otra, en este caso toma a "const_iterator" como

referencia a "rhs", en caso de que si regresa un booleano indicando si es verdad o no.

`return current == rhs.current;` Se encarga de comparar los punteros de "current" de los dos iteradores, en caso de que ambos apunten al mismo nodo de la lista seran considerado iguales con "true" si no se envia "false".

```
template<typename Object>
bool DLList<Object>::const_iterator::operator!=(const const_iterator &rhs) const {
    return !(*this == rhs);
```

Este codigo se encarga de lo opuesto al anterior en donde ahora checa si son diferentes, en caso de que si se envia un "true" y en caso de ser iguales envia un "false".

```
template<typename Object>
Object &DLList<Object>::const_iterator::retrieve() const {
    return current->data;
```

`Object &DLList<Object>::const_iterator::retrieve() const` Esto se encarga de definir el metodo "retrieve" para la clase "const_iterator" dentro del DLList, regresa una referencia al "Object" y lo marca como "const" para que indique que no debe modificar al iterador o a la lista.

`return current->data;` Se encarga de regresar una referencia al miembro "data" del nodo apuntado por el iterador "current", esto permite acceder y modificar los datos en el nodo usando al iterador sin modificar el propio iterador.

```
template<typename Object>
DLList<Object>::const_iterator::const_iterator(Node *p) : current{p} {}

template<typename Object>
DLList<Object>::iterator::iterator() {}

template<typename Object>
Object &DLList<Object>::iterator::operator*() {
    return const_iterator::retrieve();
```

`DLList<Object>::const_iterator::const_iterator(Node *p) : current{p} {}` Es un constructor para la clase "const_iterator" que toma un puntero al nodo como un parametro e inicializa el miembro "current" con este puntero.

`DLList<Object>::iterator::iterator() {}` Esto se encarga de darle datos default a la clase del iterador aun si no tiene un inicializador.

`Object &DLList<Object>::iterator::operator*() { return const_iterator::retrieve(); }` Este overload del Operator es para la clase del iterador y regresa una referencia al objeto apuntado por el operador. Llama al metodo "retrieve" de la clase const_iterator para obtener sus datos.

```
template<typename Object>
const Object &DLList<Object>::iterator::operator*() const {
return const_iterator::operator*();
```

`const Object &DLList<Object>::iterator::operator*() const` Se encarga de definir el overload del "Operator" para la clase del iterador, regresa una constante referenciando a un "Object" y lo marca como "const" para indicar que no modifique al iterador de la lista.

`return const_iterator::operator*();` Se encarga de llamar al "Operator" de la clase "const_iterator" para obtener sus datos.

```
template<typename Object>
typename DLList<Object>::iterator &DLList<Object>::iterator::operator++() {
    this->current = this->current->next;
    return *this;
```

`typename DLList<Object>::iterator &DLList<Object>::iterator::operator++()` Se encarga de definir al operador de pre-incremento para la clase del iterador, regresa un iterador y avanza al iterador al siguiente nodo en la lista.

`this->current = this->current->next;` Se encarga de avancan el iterador al siguiente nodo en la lista actualizando el puntero current para que apunte al siguiente nodo.

`return *this;` Se encarga de regresar una referencia al iterador actualizado permitiendo que las operaciones de incremento puedan ser encadenadas entre ellas.

```
template<typename Object>
typename DLList<Object>::iterator &DLList<Object>::iterator::operator--() {
    this->current = this->current->prev;
    return *this;
```

Este codigo se encarga de hacer lo opuesto al codigo anterior, en vez de incrementar disminuye el iterador al nodo anterior.

```
template<typename Object>
typename DLList<Object>::iterator DLList<Object>::iterator::operator++(int) {
iterator old = *this;
++(*this);
return old;
```

`typename DLList<Object>::iterator DLList<Object>::iterator::operator++(int)` Se encarga de definir el post-incremento para la clase del "Iterador", toma el parametro del "int" y regresa un nuevo objecto para el iterador (siendo una copia antes del incremento).

`iterator old = *this;` Se encarga de crear una copia del iterador actual y lo almacena en la variable "old".

`++(*this);` Se encarga de incrementar el iterador actual usando el operador de pre-incremento al siguiente nodo de la lista.

`return old;` Se encrga de regresar la copia del iterador antes del incremento.

```
template<typename Object>
typename DLList<Object>::iterator DLList<Object>::iterator::operator--(int) {
    iterator old = *this;
    --(*this);
    return old;
```

Este codigo se encarga de hacer lo opuesto al anterior, en ves de incrementar disminuye el iterador actual al nodo anterior en la lista.

```
template<typename Object>
typename DLList<Object>::iterator DLList<Object>::iterator::operator+(int steps) const {
    iterator new_itr = *this;
    for(int i = 0; i < steps; ++i) {
        ++new_itr;
    }
    return new_itr;
```

`typename DLList<Object>::iterator DLList<Object>::iterator::operator+(int steps) const` Se necarga de definir el operador de adicion para la clase del iterador donde toma un "int" con "steps" como parametro y regresa un nuevo objeto de iterador que es avanzado por los nodos de "step".

`iterator new_itr = *this;` Se encarga de crear una copia del iterador actual almacenandolo en la variable "new_itr"

`for(int i = 0; i < steps; ++i) { ++new_itr; }` Se encarga de avanzar al iterador "new_itr" por nodos de "steps", donde avanza repetidamente usando el operador de pre-incrementos hasta que "i" alcanze a "steps".

`return new_itr;` Se encarga de regresar el valor de new_itr ahora apuntando al nodo de "steps".

```
template<typename Object>
DLList<Object>::iterator::iterator(Node *p) : const_iterator{p} {}

template<typename Object>
DLList<Object>::DLList() : theSize{0}, head{new Node}, tail{new Node} {
head->next = tail;
tail->prev = head;
```

`DLList<Object>::iterator::iterator(Node *p) : const_iterator{p} {}` Se trata de un constructor para la clase del iterador el cual toma un puntero del nodo como parametro e inicializa la basa para el "const_iterator" con el puntero

`DLList<Object>::DLList() : theSize{0}, head{new Node}, tail{new Node} { head->next = tail; tail->prev = head; }` Se encarga de inicializar variables en donde.

`theSize` Se le da un valor de 0.

para `head` y `tail` se les crean un nuevo nodo.

Y crea una estructura de lista en donde `head` apunta a `tail` y `tail` apunta a `head` .

```
template<typename Object>
DLList<Object>::DLList(std::initializer_list<Object> initList) : DLList() {
for(const auto &item : initList) {
push_back(item);
```

`DLList<Object>::DLList(std::initializer_list<Object> initList) : DLList() {` Se encarga de un constructor que inicializa una lista de "Object" como parametro usando DLList() como defualt.

`for(const auto &item : initList) {` Se encargara de loopear sobre cada elemento en la lista "initlist".

`push_back(item);` Durante el loop cada elemento "item" es insertado en una lista doblemente enlazada usando "push_back" el cual agrega un elemento al final de la lista.

```
template<typename Object>
DLList<Object>::~DLList() {
    clear();
    delete head;
    delete tail;
```

`DLList<Object>::~DLList() {` Se encarga de definir al destructor para la clase.

`clear();` Se encarga de eliminar todos los elementos de la lista y limpiar sus datos de la memoria.

`delete head;` Se encarga de eliminar el nodo "head" de la lista.

`delete tail;` Se encarga de eliminar el nodo "tail" de la lista.

```
template<typename Object>
DLList<Object>::DLList(const DLList &rhs) : DLList() {
for(auto &item : rhs) {
push_back(item);
```

`DLList<Object>::DLList(const DLList &rhs) : DLList() {` Se encarga de definir el constructor de copia para la clase "DLList" donde toma como referencia a otro "DLList" como su parametro.

`DLList()` Esta parte del constructor anterior se encarga de llamar al constructor en dafault para inicializar la nueva lista y asi asegurarse que este vacia.

`for(auto &item : rhs) {` Se encargara de loopear sobre cada elemento de la lista "rhs" mientras que "auto &item" declara una referencia a cada elemento de la lista permitiendo que estos puedan ser modificados.

`push_back(item);` Durante el loop cada "item" de la lista "rhs" es insertada en una nueva lista usando el "push_back" el cual se encarga de copiar los elementos de "rhs" a la nueva lista.

```
template<typename Object>
DLList<Object> &DLList<Object>::operator=(const DLList &rhs) {
    DLList copy = rhs;
    std::swap(*this, copy);
    return *this;
```

`DLList<Object> &DLList<Object>::operator=(const DLList &rhs) {` Se encarga de definir el constructor de copia para la clase "DLList" donde toma como referencia a otro "DLList" como su parametro.

`DLList copy = rhs;` Se encarga de crear una copia del "rhs" usando el constructor anterios, la cosa aqui es que esta copia contiene nuevos nodos con copias de los datos de rhs.

`std::swap(*this, copy);` Se encarga de intercambiar los contenidos en "*this" y "copy" lo cual significa que reemplaza la lista actual con la copia.

`return *this;` Se encarga de regresar a la lista actual.

```
template<typename Object>
DLList<Object>::DLList(DLList &&rhs) : theSize{rhs.theSize}, head{rhs.head}, tail{rhs.tail} {
    rhs.theSize = 0;
    rhs.head = nullptr;
    rhs.tail = nullptr;
```

`DLList<Object>::DLList(DLList &&rhs) : theSize{rhs.theSize}, head{rhs.head}, tail{rhs.tail} {` Se encarga de definir el constructor de movimiento para la clase DLList, el cual toma como referencia el rvalue a otro "DLList" como parametro.

`theSize{rhs.theSize}, head{rhs.head}, tail{rhs.tail}` Se encarga de inicializar "theSize", "head" y "tail" con los valores de rhs.

`rhs.theSize = 0; rhs.head = nullptr; rhs.tail = nullptr;` Se encarga de resetear los valores a los miembros anteriores para evitar que la lista rhs elimine sus nodos cuando este sea destruida debido a que pertenecen a una nueva lista.

```
template<typename Object>
DLList<Object> &DLList<Object>::operator=(DLList &&rhs) {
std::swap(theSize, rhs.theSize);
std::swap(head, rhs.head);
std::swap(tail, rhs.tail);
return *this;
```

`DLList<Object> &DLList<Object>::operator=(DLList &&rhs) {` Se encarga de definir el operador de movimiento que toma como referencia el rvalue de otro DLList como su parametro y regresa una referencia de la lista actual.

`std::swap(theSize, rhs.theSize); std::swap(head, rhs.head); std::swap(tail, rhs.tail);` Cada uno se encarga de cambiar sus valores de la lista actual con los valores de la lista rhs lo que causa que tambien cambien de propietario pasando de "rhs" a "*this".

`return *this;` Se encarga de regresar una referencia de la lista actual que es "*this".

```
template<typename Object>
typename DLList<Object>::iterator DLList<Object>::begin() {
    return {head->next};
```

`typename DLList<Object>::iterator DLList<Object>::begin() {` Primero se define el metodo "begin" para la clase, el cual regresa un iterador que apunta al primer miembro de la lista.

`return {head->next};` Se encarga de crear un nuevo objecto de iterador inicializado con un puntero al nodo head de la lista, lo que ocaciona al nodo a apuntar al primer miembro de la lista.

```
template<typename Object>
typename DLList<Object>::const_iterator DLList<Object>::begin() const {
    return {head->next};
```

Hace lo mismo que el anterior la unica diferencia es que utiliza "const" para indicar que la lista no se modifica.

```
template<typename Object>
typename DLList<Object>::iterator DLList<Object>::end() {
    return {tail};
```

`typename DLList<Object>::iterator DLList<Object>::end() {` Se encarga de definir el metodo "end" de la clase, el cual regresa un iterador apuntando a mas alla de el ultimo elemento de la lista.

`return {tail};` Se encarga de crear un nuevo objeto iterador con un puntero al nodo "tail"de la lista, asegurandose que sobre pase al ultimo elemento de la lista.

```
template<typename Object>
typename DLList<Object>::const_iterator DLList<Object>::end() const {
return {tail};
```

Hace lo mismo que el anterior la unica diferencia es que utiliza "const" para indicar que la lista no se modifica.

```
template<typename Object>
int DLList<Object>::size() const {
```

```
    return theSize;
```

`int DLList<Object>::size() const {:` Se encarga de definir "size" para la clase y es marcado como const para indicar que no modifica la lista.

`return theSize;` Se encarga de regresar el valor del miembro privado "theSize" el cual representa el numero de elementos en la lista.

```
template<typename Object>
bool DLList<Object>::empty() const {
return size() == 0;
```

`bool DLList<Object>::empty() const {:` Es un booleano que se encarga de definir el metodo "empty" de la clase y es marcado como const para indicar que no modifica la lista.

`return size() == 0;` Se encarga de usar el metodo de size para comparar el numero de elementos en una lista con el valor de cero, en caso de ser iguales envia un true, si no envia un false.

```
template<typename Object>
void DLList<Object>::clear() {
    while(!empty()) {
        pop_front();
```

`void DLList<Object>::clear() {` Se encarga de definir el metdoo de "clear" a la clase con un Object como parametro.

`while(!empty()) {` Se encarga de hacer un loop a travez de la lista y continua siempre y cuando esta no este vacia, se utiliza el "empty" para checar si esta vacia o no.

`pop_front();` Durante el loop la funcion pop_front es usada para eliminar el primer elemnto de la lista y asi se repite hasta que la lista este vacia.

```
template<typename Object>
Object &DLList<Object>::front() {
    return *begin();
```

`Object &DLList<Object>::front() {` Se encarga de definir el metdoo de "front" a la clase con un Object como parametro.

`return *begin();` Se encarga de referenciar al primer elemento de la lista diferenciandolo del iterador regresado por el metodo "begin"

```
template<typename Object>
const Object &DLList<Object>::front() const {
```

```
return *begin();
```

Este codigo hace lo mismo que el anterior, la unica diferencia es que utiliza const para dar a entender que la lista no se modifica.

```
template<typename Object>
Object &DLList<Object>::back() {
    return *(--end());
```

`Object &DLList<Object>::back() {` Se encarga de definir el metodo "back" de la clase con un parametro de Object.

`return *(--end());` Se encarga de disminuir el iterador por el metodo "end" para asi tener al iterador apuntando al último elemento de la lista para luego regresar una referencia del último elemento.

```
template<typename Object>
const Object &DLList<Object>::back() const {
    return *(--end());
```

Este codigo hace lo mismo que el anterior, la unica diferencia es que utiliza const para dar a entender que la lista no se modifica.

```
template<typename Object>
void DLList<Object>::push_front(const Object &x) {
    insert(begin(), x);
```

`void DLList<Object>::push_front(const Object &x) {` Se encarga de definir el metodo "push_front" de la clase, usado con un parametro de "Object" y toma como referencia un const de un "Object" con "&x" como su parametro.

`insert(begin(), x);` Se encarga de llamar al metodo "insert" pasandolo como un iterador apuntando al inicio de la lista y al valor de "x" a insertar, util para insertar nuevos elementos al inicio de la lista.

```
template<typename Object>
void DLList<Object>::push_front(Object &&x) {
    insert(begin(), std::move(x));
```

Si bien no identicos esta funcion tiene similitudes con el anterior las diferencias siendo.

`push_front(Object &&x)` Se utiliza con una referencia a un rvalue como su parametro lo que permite al metodo mover los valores a la lista.

`std::move(x)` Esto sirve para darle un rvalue a "x" permitiendo que se pueda mover a la lista.

```
template<typename Object>
void DLList<Object>::push_back(const Object &x) {
    insert(end(), x);

template<typename Object>
void DLList<Object>::push_back(Object &&x) {
    insert(end(), std::move(x));
```

Este codigo es lo inverso de los dos ultimos codigos en donde, va e inserta nuevos elementos a la lista empezando por el ultimo miembro de esta.

De igual forma la segunda parte es para facilitar el movimiento del valor "x" a la lista.

```
template<typename Object>
void DLList<Object>::pop_front() {
erase(begin());
```

`void DLList<Object>::pop_front() {` Se encarga de definir el metodo "pop_front" de la clase, usado con un parametro de "Object"

`erase(begin());` Se encarga de eliminar el primer elemento de la lista gracias al iterador que apunta hacia esta.

```
template<typename Object>
void DLList<Object>::pop_back() {
    erase(--end());
```

`void DLList<Object>::pop_back() {` Se encarga de definir el metodo "pop_front" de la clase, usado con un parametro de "Object"

`erase(--end());` Se encarga de disminuir el iterador regresado por "end" el cual apunta al ultimo miembro de la lista. para luego eliminarlo de esta.

```
template<typename Object>
typename DLList<Object>::iterator DLList<Object>::insert(iterator itr, const Object &x) {
    Node *p = itr.current;
    theSize++;
    return {p->prev = p->prev->next = new Node{x, p->prev, p}};
```

`typename DLList<Object>::iterator DLList<Object>::insert(iterator itr, const Object &x) {` Se encarga de definir el metodo de "insert" con el parametro "Object", tomando el iterador "itr" y una referencia de tipo const con "Object" como su parametro.

`Node *p = itr.current;` Se encarga de tener un puntero "p" en el nodo actualmente apuntado por el iterador.

`theSize++;` Se encarga de incrementar el tamaño de la lista tomando en cuanta nuevos elementos que sea agregados.

`return {p->prev = p->prev->next = new Node{x, p->prev, p}};` Se encarga de crear un nuevo nodo con el valor de "x" y se inserta a la lista antes del nodo especificado por itr, actualizando de paso los punteros "next" y "prev" que rodean el nuevo nodo a la lista de forma apropiada para despues regresar el iterador apuntando al elemento recien insertado.

```
template<typename Object>
typename DLList<Object>::iterator DLList<Object>::insert(iterator itr, Object &&x) {
    Node *p = itr.current;
    theSize++;
    return {p->prev = p->prev->next = new Node{std::move(x), p->prev, p}};
```

Este codgio escencialmente es lo mismo que el anterior pero con algunas diferencias.

En `insert(iterator itr, Object &&x)` ahora se usa una referencia de tipo rvalue en vez de const.

Y en `new Node{std::move(x), p->prev, p}` se crea el nodo con valor de "x" usando std::move para que el valor se mueva de forma eficiente en la lista y lo inserta antes del nodo apuntado por itr.

```
template<typename Object>
void DLList<Object>::insert(int pos, const Object &x) {
    insert(get_iterator(pos), x);
```

`void DLList<Object>::insert(int pos, const Object &x) {` Se encarga de definir el metodo de "insert" con el parametro "Object", tomando el iteger "pos" y una referencia de tipo const con "Object" como su parametro.

`insert(get_iterator(pos), x);` Se encarga de llamar el metodo privado "get_iterator" para obtener el iterador apuntando a la posicion "pos" en la lista para luego llamar al metodo "insert" para insertar el elemento "x" a la posicion.

```
template<typename Object>
void DLList<Object>::insert(int pos, Object &&x) {
    insert(get_iterator(pos), std::move(x));
```

Es casi identico al anterior pero con algunas modificaciones por ejemplo:

En `insert(int pos, Object &&x)` el "Object" es referenciado como un rvalue y no un const.

Mientras que en `insert(get_iterator(pos), std::move(x));` se crea el nodo con valor de "x" usando std::move para que el valor se mueva de forma eficiente en caso de que sea un rvalue.

```
template<typename Object>
typename DLList<Object>::iterator DLList<Object>::erase(iterator itr) {
    Node *p = itr.current;
    iterator retVal(p->next);
    p->prev->next = p->next;
    p->next->prev = p->prev;
    delete p;
    theSize--;
    return retVal;
```

`typename DLList<Object>::iterator DLList<Object>::erase(iterator itr) {` Se encarga de definir el metodo de "erase" con el parametro "Object", tomando un iterador "itr" como su parametro, despues regresa un iterador apuntando al elemento que sigue despues del que fue eliminado.

`Node *p = itr.current;` Se encarga de conseguir el puntero "p" al nodo que esta siendo apuntado por el iterador itr.

`iterator retVal(p->next);` Se encarga de crear el iterador "retVal" apuntando al elemento siguiente del eliminado.

`p->prev->next = p->next; and p->next->prev = p->prev;` Se encarga de actualizar los punteros "next" y "prev" de los nodos alrededor del nodo que fue eliminado.

`delete p;` Se encarga de eliminar el nodo que fue eliminado de la memoria.

`theSize--;` Se encarga de disminuir el tamaño de la lista para reflejar el elemento eliminado.

`return retVal;` Se encarga de regresar el iterador "retVal" el cual apunta al elemento que fue eliminado.

```
template<typename Object>
void DLList<Object>::erase(int pos) {
    erase(get_iterator(pos));
```

`void DLList<Object>::erase(int pos) {` Se encarga de definir el metodo de "erase" con el parametro del template "Object" y tomando el iteger "pos" como parametro.

`erase(get_iterator(pos));` Se encarga de llamar al metodo privado "get_iterator" para tener el iterador apuntando a la posicion de "pos" en la lista, para luego llamar al metodo "erase" para elimianr al elemento de esa posicion.

```
template<typename Object>
typename DLList<Object>::iterator DLList<Object>::erase(iterator from, iterator to) {
    for(iterator itr = from; itr != to;) {
        itr = erase(itr);
    }
    return to;
```

`typename DLList<Object>::iterator DLList<Object>::erase(iterator from, iterator to) {` Se encarga de definir el metodo de "erase" con el parametro del template "Object" y tomando a dos iteradores "from" y "to" para representar el rango de elementos eliminados, para luego regresar el iterador apuntando al siguiente elemento despues del eliminado.

`for(iterator itr = from; itr != to;) {` Se encarga de inicializar el iterador "itr" al "from" y luego iteraria sobre el rango (from, to), el loop continua siempre y cuando "itr" no sea igual a "to".

`itr = erase(itr);` Durante el loop cualquier elemento apuntado por "itr" pasara por el metodo de "erase" con el iterador apuntando al elemento siguiente del eliminado el cual se le asignara "itr".

`return to;` Se encarga de regresar el iterador "to" el cual apunta al elemento siguiente del elemento dentro del rango (from, to).

```
template<typename Object>
void DLList<Object>::print() const {
    for(const auto &item : *this) {
        std::cout << item << " ";
    }
    std::cout << std::endl;
```

`void DLList<Object>::print() const {` Se encarga de definir el metodo de "print" con el parametro del template "Object" y es marcado como const para indicar que no modifica la lista.

`for(const auto &item : *this) {` Se encarga de empezar el loop que itera sobre cada elemento de la lista baado en rango la clave "auto" es usado automaticamente para deducir el tipo de cada elemento y "const auto &item" crea una referencia para cada elemento.

`std::cout << item << " ";` Dentro del loop cada elemento de tipo "item" es imprimido.

`std::cout << std::endl;` Despues del loop se imprimime los resultados.

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

`void DLList<Object>::init() {` Se encarga de definir el metodo de "init" con el parametro del template "Object"

De ahi se le da un valor a cada de los siguientes miembros siendo.

`theSize` se le da un valor de cero.

`head` crea un nuevo nodo y funciona como un valor defualt.

`tail` crea un nuevo nodo y funciona como un valor defualt.

`head->next = tail;` pone el puntero "next" del nodo de "head" para que apunte al nodo "tail".

`tail->prev = head;` pone el puntero "prev" del nodo de "tail" para que apunte al nodo "head".

```
template<typename Object>
typename DLList<Object>::iterator DLList<Object>::get_iterator(int pos) {
    iterator itr = begin();
    for(int i = 0; i < pos; ++i) {
        ++itr;
    }
    return itr;
```

`typename DLList<Object>::iterator DLList<Object>::get_iterator(int pos) {` Se encarga de definir el metodo de "init" con el parametro del template "Object", tomando el iteger "pos" como parametro y regresa un iterador apuntando al elemento a la posicion "pos".

`iterator itr = begin();` Se encarga de inicializar el iterador "itr" al inicio de la lista usando el metodo "begin".

`for(int i = 0; i < pos; ++i) {` Se encarga de iniciar el loop que itera tiempos "pos", avanzando el iterador "itr" al siguiente elemento de la lista cada vez.

`++itr;` Durante el loop el iterador "itr" es incrementado para que apunte al siguiente elemento de la lista.

`return itr;` Despues del loop el metodo regresa el iterador "itr" el cual ahora apunta al elemento en la posicion "pos" de la lista.

Y ya con esto se termina de documentar ambos codigos en un solo archivo.