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

SLList.h

Empezamos por SLList.h debido a que la funcion .cpp manda a llamar al .h en varias ocaciones por lo que es mejor explicar .h primero.

```
#ifndef SLLLIST_H
#define SLLLIST_H

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

Primero tenemos en el header dos definiciones del mismo archivo "SLLLIST_H".

ifndef se encarga de checar si el macro "SLLLIST_H" esta definido o no, en el caso de que no este definido todo el codigo que este entre "ifndef" y "endif" sera incluido.

define se encarga de definir SLLLIST_h lo que causa que cada vez que el header sea incluido en algun codigo no escriba todo el contenido entre "ifndef" y "endif".

ya de ahi include solo se encarga de incluir funciones de iostream y utility.

```
template<typename Object>
class SLList {
public:
    struct Node {
        Object data;
        Node *next;

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

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

Aqui se esta creando un template con el placeholder de "Object" que sirve como parametro para no tener que escribir el mismo codigo en diferentes tipos de datos.

Despues se esta creando una clase con sus atributos en publico lo que permite que cualquiera pueda verlos y editarlos (Tanto fuera como adentro de este archivo).

Luego se crea un struct que sirve para definir un tipo de "node" en el cual se encuentra: `Object data:` se encarga de almacenar datos de tipo "Object".

`Node *next:` es un puntero que apunta al siguiente nodo de la lista.

`Node(const Object &d = Object{}, Node *n = nullptr):` es un constructor que se encarga de darles valores predeterminados a "data" y "next" con data siendo "Object{}" y next con "nullptr",

`Node(Object &&d, Node *n = nullptr):` otro constructor con la misma funcion, pero en este caso inicializa los nodos con rvalue para "data" y a "next" con el mismo "nullptr".

```
public:
    class iterator {
    public:
        iterator();

        Object &operator*();

        iterator &operator++();

        const iterator operator++(int);

        bool operator==(const iterator &rhs) const;

        bool operator!=(const iterator &rhs) const;

    private:
        Node *current;

        iterator(Node *p);

        friend class SLList<Object>;
    };
```

Otra funcion publica, esta vez con una clase llamada "iterator".

Empezando por las variables publicas tenemos:

`iterator():` el constructor por default de la clase.

`Object &operator*():` se encarga de sobrecargar el operador '*' para eliminar la referencia al iterador y devolver una referencia al Object en el nodo actual.

`iterator &operator++():` se encarga de sobrecargar el prefijo '++' del operador para mover el iterador al siguiente nodo de la lista y regresar la referencia del iterador actualizado.

`const iterator operator++(int):` se encarga de sobrecargar el postfijo '++' del operador para mover al iterador al siguiente nodo en la lista pero regresando una copia del iterador antes del cambio.

`bool operator==(const iterator &rhs) const:` un booleano que se encarga de comparar dos iteradores e identificar si son iguales (en este caso identificar si el operator es igual al ietrator).

`bool operator!=(const iterator &rhs) const:` un booleano que se encarga de comparar dos iteradores e identificar si son diferentes (en este caso identificar si el operator es differente al ietrator).

Ahora con las funciones privadas que en este caso son datos ocultos y que no cualquiera puede modificar.

`Node *current:` esta variable pose un puntero al nodo mas reciente referenciado por el iterator.

`iterator(Node *p):` es un constructor que se encarga de inicializar el iterador con un puntero a un nodo.

`friend class SLList<Object>:` se encarga de declarar esta class template como amigo del iterador actual permitiendole a esta clase a acceder a las funciones privadas del iterador.

```
  public:
      SLList();

      SLList(std::initializer_list <Object> init_list);

      ~SLList();

      iterator begin();

      iterator end();

      int size() const;

      bool empty() const;

      void clear();

      Object &front();

      void push_front(const Object &x);

      void push_front(Object &&x);

      void pop_front();

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

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

      iterator erase(iterator itr);

      void print();
```

Estas son funciones publicas de la funcion en general, en orden son:

`SLList():` Manda a llamar a la clase.

`SLList(std::initializer_list <Object> init_list):` Un constructor que inicializa una lista con los elementos dados.

`~SLList():` Manda a destruit a la clase.

`"iterator begin()" e "iterator end()":` ambos regresan un iterador, begin apunta al primero mientras que end apunta al ultimo.

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

`bool empty() const:` Un booleano que regresa un "true" si la lista esta vacia y un "false" de lo contrario.

`void clear():` Se encarga de eliminar todos los elementos de la lista.

`Object &front():` Se encarga de regresar el primer elemento de la lista en Object.

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

`void push_front(Object &&x):` Se encarga de mover un elemento al inicio de la lista.

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

`iterator insert(iterator itr, const Object &x):` Se encarga de insertar un elemento antes del elemento indicado por el itirador "itr".

`iterator insert(iterator itr, Object &&x):` Se encarga de insertar un elemento antes del elemento indicado por el itirador "itr" usando semanticas de movimiento.

`iterator erase(iterator itr):` Se encarga de eliminar un elemento de la lista indicado por el iterador.

`void print():` imprime los elementos de la lista en la terminal. Y ya por ultimo en este archivo tenemos.

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

      void init();
  };

  #include "SLList.cpp"

  #endif
```

En las funciones publicas tenemos.

`Node *head:` Es un puntero al primer nodo de la lista.

`Node *tail:` Es un puntero al ultimo nodo de la lista.

`int theSize:` Es un integral que sigue el numero de elementos de la lista.

`void init():` Se encarga de inicializar la lista.

Ya por ultimo "#include "SLList.cpp" se encarga de separar la implementacion de un class template de su declaracion y "endif" es 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.

SLList.cpp

Ahora pasamos al archivo principal que usara el header que acabamos de explicar.

Para empezar tenemor "#include "SLList.h"" el cual sirve para mandar a llamar al header. ahora si a por el codigo!!

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

Primero tenemos

`template (typename Object):` Aqui se manda a llamar al class template dentro del header.

`SLList<Object>::Node::Node(const Object &d, Node *n):` Dentro de esta funcion se encuentran dos parametros const Object &d: indica que el "Object" se almacenada en un node.

`Node *n:` un puntero al siguiente nodo de la lista.

`: data{d}, next{n} {}:` se encarga de inicializar "data" y "next" con data teniendo un valor de "d" y next un valor de "n".

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

Es casi identico al anterior con las diferencias siendo!

`Object &&d:` un rvalue referenciado a "Object" que se almacenada en un node.

`data{std::move(d)}:` se encarga de inicializar "data" moviendo "d" a otro lado de "data".

```
template<typename Object>
SLList<Object>::iterator::iterator() : current{nullptr} {}
```

Se manda a llamar al class template.

`SLList<Object>::iterator::iterator() : current{nullptr} {}:` Esta es el constructor de la definicion del iterador, este es iniciado con un valor de "nullptr" en el puntero de "current".

```
template<typename Object>
Object &SLList<Object>::iterator::operator*() {
    if(current == nullptr)
        throw std::logic_error("Trying to dereference a null pointer.");
    return current->data;
```

`Object &SLList<Object>::iterator::operator*():` Se encarga de definir la funcion del operator*() a las clase del iterator y regresa una referencia al Object almacenado en el nodo mas reciente.

`if(current == nullptr) throw std::logic_error("Trying to dereference a null pointer."):` Despues se usa un "if" en donde si current es igual a nullptr se imprime una advertencia de error con.

`return current->data:` Siendo una linea que regresa una referencia al miembro de "data" del nodo.

```
template<typename Object>
typename SLList<Object>::iterator &SLList<Object>::iterator::operator++() {
    if(current)
        current = current->next;
    else
        throw std::logic_error("Trying to increment past the end.");
    return *this;
```

Funciona casi identico al anterior con la diferencia de que:

`typename SLList<Object>::iterator &SLList<Object>::iterator::operator++():` Define el prefijo de incremento "operator++" para el iterador, al final regresa una referencia el iterador actualizado despues del incremento.

`if(current) current = current->next else throw std::logic_error("Trying to increment past the end."):` Se encarga de checar si current NO es nullptr, en caso de que no sean el puntero se movera al siguiente nodo en la lista, en caso de que si sea mandara el mismo error de antes.

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

El Typename realiza lo mismo que la funcion anterior con la diferencia de que: `:operator++(int):` Toma el parametro del integer para diferenciarlo del prefijo del incremento de "operator+"

`iterator old = *this:` Este se encarga de crear una copia del iterator actual nombrandolo "old" en el proceso.

`++(*this);` Se encarga de incrementar el iterator actual usando el prefijo de "operator++"

`return old;` Regresa una copia de "old" el valor del iterator antes del incremento.

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

`"bool SLList<Object>::iterator::operator==(const iterator &rhs) const"` Esta operacion se encarga de definir.

`"operator=="` para el iterator, toma como referencia la constante de otro itirator en este caso (rhs) y regresa un booleano indicando si son iguales o no.

`return current == rhs.current;` checa si el current es identico al current del rhs y en el caso de que si envia un true y de lo contario un false.

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

`bool SLList<Object>::iterator::operator!=(const iterator &rhs) const` La unica diferencia con el anterior es que este busca la diferencia en operator en vez de igualdad.

`return !(*this == rhs);` Lo que hace es checar si son identicos o no, con la diferencia de que si son iguales regresan un "false" en vez de "true" y viceversa.

```
template<typename Object>
SLList<Object>::iterator::iterator(Node *p) : current{p} {}

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

`SLList<Object>::iterator::iterator(Node *p) : current{p} {}` Es un constructor que agarra un puntero al node como un parametro e inicia el miembro "current" con ese puntero.

`SLList<Object>::SLList() : head(new Node()), tail(new Node()), theSize(0)` Es el constructor default de la class template, manda a llamar el head y tail al nodo y le asigna un valor de cero a "TheSize".

`head->next = tail;` Se encarga de establecer el puntero de head al nodo de tail.

```
template<typename Object>
SLList<Object>::SLList(std::initializer_list <Object> init_list) {
head = new Node();
tail = new Node();
```

```
head->next = tail;
theSize = 0;
for(const auto& x : init_list) {
push_front(x);
}
```

`SLList<Object>::SLList(std::initializer_list<Object> init_list)` Un constructor que se encarga de tomar el initializer_list de Object y lo pone como parametro.

`head = new Node():` inicializa el puntero de head como un nodo.

`tail = new Node():` inicializa el puntero de tail como un nodo.

`head->next = tail:` Se encarga de establecer el puntero de head al nodo de tail.

`theSize = 0` Le da un valor de cero al miembro de TheSize.

`for(const auto& x : init_list) { push_front(x); }` Esto se encarga de repetir cada elemento del init_list para luego insertarlas al inicio de una lista, debido a que se usa el metodo de push_front hace que todos los elementos se ordenen basados en la lista original.

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

`SLList<Object>::~SLList()` Se encarga de eliminar y limpiar la memoria de la info de la lista.

`clear(), delete head, delete tail:` Estas tres se encargan de eliminar los datos en general, de la head y de la tail.

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

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

template<typename Object>
int SLList<Object>::size() const { return theSize; }

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

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

template<typename Object>
Object &SLList<Object>::front() {
    if(empty())
```

```
        throw std::logic_error("List is empty.");
    return *begin();
```

`typename SLList<Object>::iterator SLList<Object>::begin() { return {head->next}; }` Se encarga de regresar un iterador que apunta al final de la lista (En este caso es "head").

`typename SLList<Object>::iterator SLList<Object>::end() { return {tail}; }` Se encarga de regresar un iterador que apunta al final de la lista (En este caso es "tail").

`int SLList<Object>::size() const { return theSize; }` Se encarga de regresar el numero de elementos en una lista.

`bool SLList<Object>::empty() const { return size() == 0; }` Se encarga de checar si la lista esta vacia, envia un true si no envia un false.

`void SLList<Object>::clear() { while (!empty()) pop_front(); }` Se encarga de eliminar todos los elementos de una lista usando la funcion pop_front() repetidamente hasta que la lista este vacia.

`Object &SLList<Object>::front() { ... }` Se encarga de regresar el primer elemento de la lista, en caso de que la lista este vacia enviara un error a la consola.

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

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

template<typename Object>
void SLList<Object>::pop_front() {
if(empty())
throw std::logic_error("List is empty.");
erase(begin());
```

`void SLList<Object>::push_front(const Object &x) { insert(begin(), x); }` Se encarga de insertar un nuevo elemento al inicio de la lista por el metodo de insert, con el iterador apuntando al inicio de la lista y los elementos a insertar.

`void SLList<Object>::push_front(Object &&x) { insert(begin(), std::move(x)); }` Es igual a la funcion anterior con la diferencia de que el nuevo elemento es de tipo rvalue y por moviendo el elemento atravez de la lista.

`void SLList<Object>::pop_front() { ... }` Primero se encarga de eliminar el primer elemento de la lista , luego checa si la lista esta vacia, en caso de que si envia un error a la consola y por ultimo se utiliza el metodo de "erase" con un iterador que apunta al inicio de la lista para eliminar un elemento.

```
template<typename Object>
typename SLList<Object>::iterator SLList<Object>::insert(iterator itr, const Object &x) {
```

```
    Node *p = itr.current;
    Node *newNode = new Node{x, p->next};
    p->next = newNode;
    theSize++;
    return iterator(newNode);
```

`typename SLList<Object>::iterator SLList<Object>::insert(iterator itr, const Object &x)` Esto sirve como definicion del metodo del "insert" para la class template, luego toma el iterador "itr" y referencia constantemente "Object" y "x" como parametros.

`Node *p = itr.current;` Se encarga de obtener el puntero "p" al nodo utilizando el itirador "itr".

`Node *newNode = new Node{x, p->next};` Se encarga de crear un nuevo nodo usando "x" como dato y con el puntero "next" obteniendo el nodo con "p".

`p->next = newNode;` Se encarga de actualizar el puntero "next" con el puntero "p" para que el resultado sobreescriba el "newnode".

`theSize++;` Se encarga de incrementar el tamaño de la lista de "TheSize".

`return iterator(newNode);` Se encarga de regresar el iterador que muestra al mas reciente "newNode".

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

En si la funcion es identica a la anterior con unos cambios, por ejemplo en:

`typename SLList<Object>::iterator SLList<Object>::insert(iterator itr, Object &&x)` Esto sirve como definicion del metodo del "insert" para la class template, luego toma el iterador "itr" y un rvalue que va como "Object" y "x" como parametros y los regresa como iteradores.

y

`Node *newNode = new Node{std::move(x), p->next};` Se encarga de crear un nuevo nodo llamado "newNode" con pointer y un dato "x" usando un std::move(x).

```
template<typename Object>
typename SLList<Object>::iterator SLList<Object>::erase(iterator itr) {
if (itr == end())
throw std::logic_error("Cannot erase at end iterator");
Node *p = head;
while (p->next != itr.current) p = p->next;
Node *toDelete = itr.current;
```

```
p->next = itr.current->next;
delete toDelete;
theSize--;
return iterator(p->next);
```

`typename SLList<Object>::iterator SLList<Object>::erase(iterator itr)` Se encarga de definir el metodo "erase" para el class template, donde toma al "itr" como parametro y lo regresa como iterador.

`if (itr == end()) throw std::logic_error("Cannot erase at end iterator");` Se encarga de checar si el iterador "itr" apunta al final de la lista, en caso de que si termina enviando un error a la consola.

`Node *p = head; while (p->next != itr.current) p = p->next;` Es un loop que se encarga de encontrar el nodo "p" que anteceda al nodo que apunta el iterador.

`Node *toDelete = itr.current;` Esta parte se encarga de asignar un puntero al nodo "current" que apunta el itr".

`p->next = itr.current->next;` Se encarga de actualizar el puntero next, del nodo p para remover el nodo apuntando por el itr, sobreescribiendolo.

`delete toDelete;` Se encarga de eliminar el nodo que el itr esta referenciando, desasignadolo de la memoria.

`theSize--` Se encarga de decrementar el tamaño de la lista que se encuentra en "TheSize".

`return iterator(p->next);` Se encarga de regresar la funcion que un iterador este apuntando al nodo despues del que fuera eliminado.

```
template<typename Object>
void SLList<Object>::print() {
iterator itr = begin();
while (itr != end()) {
std::cout << *itr << " ";
++itr;
}
std::cout << std::endl;
```

`void SLList<Object>::print()` Se encarga de definir "print" para el class template.

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

`while (itr != end()) { ... }` Se encarga de loopear el iterador para que pase por cada elemento de la lista usando itr hasta alcanzar el final de esta.

`std::cout << *itr << " ";` Cuando se encuentre en el loop esto se encargara de imprimir los valores de cada elemento que apunte el iterador itr.

`++itr;` Se encarga de incrementar el valor del itr para que apunte al siguiente elemento de la lista.

`std::cout << std::endl;` Despues del loop se imprimiran un nuevo character para que se muevan a la siguiente linea de depuracion.

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

`void SLList<Object>::init()` Se encarga de definir "init" para el class template.

`theSize = 0;` Se encarga de establecer el numero de miembros en la lista a cero.

`head->next = tail;` Se encarga de reemplazar la funcion head del "next" por tail.