

# ESTRUCTURAS DE DATOS LINEALES

# **LISTAS**

Joaquín Fernández-Valdivia

Javier Abad

Dpto. de Ciencias de la Computación e Inteligencia Artificial

Universidad de Granada



# Listas

- Una **lista** es una estructura de datos lineal que contiene una secuencia de elementos, diseñada para realizar inserciones, borrados y accesos en cualquier posición
- La representaremos como  $\langle a_1, a_2, \dots, a_n \rangle$
- Operaciones básicas:
  - ▶ Set: modifica el elemento de una posición
  - ▶ Get: devuelve el elemento de una posición
  - ▶ Erase: elimina el elemento de una posición
  - ▶ Insert: inserta un elemento en una posición
  - ▶ Size: devuelve el número de elementos de la lista
- En una lista con  $n$  elementos consideraremos  $n+1$  posiciones, incluyendo *la siguiente a la última*, que llamaremos **fin de la lista**

# Listas. Primera aproximación

Esquema de la interfaz

```
#ifndef __LISTA_H__  
#define __LISTA_H__
```

```
typedef char Tbase;
```

```
class Lista{  
private:  
    ... //La implementación que se elija  
public:  
    Lista();  
    Lista(const Lista& l);  
    ~Lista();  
    Lista& operator=(const Lista& l);
```

```
    Tbase get(int pos) const;  
    void set(int pos, Tbase e);  
    void insert(int pos, Tbase e);  
    void erase(int pos);  
    int size() const;
```

```
};
```

```
#endif // __LISTA_H__
```

# Listas. Posibles implementaciones

- **Vectores.** A priori sencilla: las posiciones que se pasan a los métodos son enteras y se traducen directamente en índices del vector. Inserciones y borrados ineficientes (orden lineal)
- **Celdas enlazadas.** Parece más eficiente: inserciones y borrados no desplazan elementos. Los métodos set, get, insertar y borrar tienen orden lineal. El problema son las posiciones enteras
- **Conclusión:** la implementación de las posiciones debe variar en función de la implementación de la lista 😞😞

# Listas. Posiciones

- Vamos a crear una abstracción de las posiciones, encapsulando el concepto de posición en una clase.
- Crearemos una clase **Posicion**. Un objeto de la clase representa una posición en la lista.
  - ▶ En el caso del vector, se implementa como un entero
  - ▶ En el caso de las celdas enlazadas, será un puntero
- ▶ Observaciones:
  - ▶ Para una lista de tamaño  $n$ , habrá  $n+1$  posiciones posibles
  - ▶ El movimiento entre posiciones se hace una a una
  - ▶ La comparación entre posiciones se limita a igualdad y desigualdad (no existe el concepto de anterior o posterior)

# Listas. Clases Posicion y Lista

```
#ifndef __LISTA_H__  
#define __LISTA_H__
```

```
typedef char Tbase;
```

```
class Posicion{  
private:  
    ...    //La implementación que se elija
```

```
public:  
    Posicion();  
    Posicion(const Posicion& p);  
    ~Posicion();  
    Posicion& operator=(const Posicion& p);  
    Posicion& operator++();  
    Posicion operator++(int);  
    Posicion& operator--();  
    Posicion operator--(int);  
    bool operator==(const Posicion& p) const;  
    bool operator!=(const Posicion& p) const;  
};
```

Esquema de la interfaz

# Listas. Clases Posicion y Lista

```
class Lista{  
private:  
    ...    //La implementación que se elija
```

Esquema de la interfaz

```
public:  
    Lista();  
    Lista(const Lista& l);  
    ~Lista();  
    Lista& operator=(const Lista& l);
```

```
    Tbase get(const Posicion & p) const;  
    void set(const Posicion & p, const Tbase & e);  
    Posicion insert(const Posicion & p, const Tbase & e);  
    Posicion erase(const Posicion & p);  
    int size() const;  
    Posicion begin() const;  
    Posicion end() const;  
};  
#endif // __LISTA_H__
```

Se modifican

size() no es fundamental

Necesitamos saber dónde  
empieza y acaba la lista

begin() devuelve la posición del primer elemento  
end() devuelve la posición posterior al último elemento (permite añadir al final)  
En una lista vacía, begin() coincide con end()

# Listas

## Uso de una lista

```
#include <iostream>
#include "Lista.hpp"
using namespace std;
int main() {
    char dato;
    Lista l;

    cout << "Escriba una frase" << endl;
    while((dato=cin.get())!='\n')
        l.insert(l.end(), dato);
    cout << "La frase introducida es:" << endl;
    escribir(l);
    cout << "La frase en minúsculas:" << endl;
    escribir_minuscula(l);
    if(localizar(l, ' ')==l.end())
        cout << "La frase no tiene espacios" << endl;
    else{
        cout << "La frase sin espacios:" << endl;
        Lista aux(l);
        borrar_caracter(aux, ' ');
        escribir(aux);
    }
    cout << "La frase al revés: " << endl;
    escribir(al_reves(l));
    cout << (palindromo(l)? "Es " : "No es ") << "un palíndromo" << endl;
    return 0;
}
```



# Listas

## Uso de una lista

```
bool vacia(Lista& l){  
    return(l.begin()==l.end());  
}
```

```
void todo_minuscula(Lista& l){  
    for(Posicion p=l.begin(); p!=l.end(); ++p)  
        l.set(p, tolower(l.get(p)));  
}
```

```
void escribir(const Lista& l){  
    for(Posicion p=l.begin(); p!=l.end(); ++p)  
        cout << l.get(p);  
    cout << endl;  
}
```

```
void escribir_minuscula(const Lista &l){  
    Lista l2(l);  
    todo_minuscula(l2);  
    escribir(l2);  
}
```

# Listas

## Uso de una lista

```
void borrar_caracter(Lista&l, char c){
    Posicion p = l.begin();
    while(p != l.end())
        if(l.get(p) == c)
            p = l.erase(p);
        else
            ++p;
}
```

```
Lista al_reves(const Lista& l){
    Lista aux;
    for(Posicion p=l.begin(); p!=l.end(); ++p)
        aux.insert(aux.begin(), l.get(p));
    return aux;
}
```

```
Posicion localizar(const Lista& l, char c){
    for(Posicion p=l.begin(); p!=l.end(); ++p)
        if(l.get(p)==c)
            return(p);
    return l.end();
}
```

# Listas

## Uso de una lista

```
bool palindromo(const Lista& l){
    bool es_palindromo = true;
    Lista aux(l);
    int n = l.size();
    if(n>=2){
        borrar_caracter(aux, ' ');
        todo_minuscula(aux);

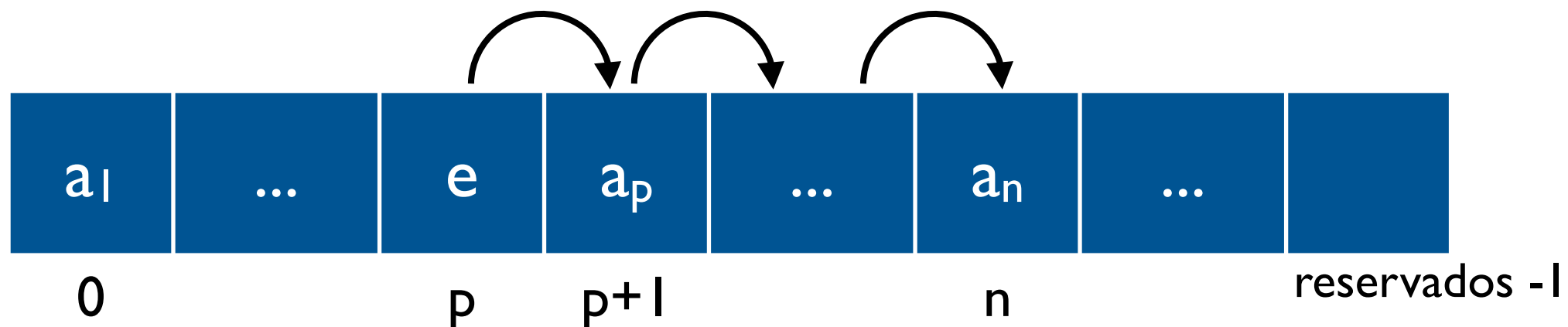
        Posicion p1, p2;
        p1 = aux.begin();
        p2 = aux.end();
        p2--;
        for(int i=0; i<n/2 && es_palindromo; i++){
            if(aux.get(p1) != aux.get(p2))
                es_palindromo = false;
            p1++;
            p2--;
        }
    }
    return es_palindromo;
}
```

# Listas. Implementación con vectores

- Almacenamos la secuencia de valores en un vector. Las posiciones son enteros



- La posición `begin()` corresponde al 0
- La posición `end()` corresponde a  $n$  (después del último)
- Las inserciones suponen desplazar elementos a la derecha y los borrados, a la izquierda



# Lista.h

```
#ifndef __LISTA_H__
#define __LISTA_H__

typedef char Tbase;

class Lista;

class Posicion{
private:
    int i;
public:
    Posicion();
    Posicion(const Posicion& p) = default;
    ~Posicion() = default;
    Posicion& operator=(const Posicion& p) = default;
    Posicion& operator++();
    Posicion operator++(int);
    Posicion& operator--();
    Posicion operator--(int);
    bool operator==(const Posicion& p) const;
    bool operator!=(const Posicion& p) const;
    friend class Lista;
};
```

# Lista.h

```
class Lista{
private:
    Tbase* datos;
    int nelementos;
    int reservados;
public:
    Lista();
    Lista(const Lista& l);
    ~Lista();
    Lista& operator=(const Lista& l);

    Tbase get(const Posicion & p) const;
    void set(const Posicion & p, const Tbase & e);
    Posicion insert(const Posicion & p, const Tbase & e);
    Posicion erase(const Posicion & p);
    int size() const;
    Posicion begin() const;
    Posicion end() const;
private:
    void liberar();
    void redimensionar(int n);
    void copiar(const Lista& l);
    void reservar(int n);
};

#endif // __LISTA_H__
```

# Lista.cpp

```
#include <cassert>
#include "Lista.hpp"
```

```
using namespace std;
```

```
//Clase Posicion
```

```
Posicion::Posicion(){
    i = 0;
}
```

```
//Operador ++ prefijo
```

```
Posicion& Posicion::operator++(){
    i++;
    return *this;
}
```

```
//Operador ++ postfijo
```

```
Posicion Posicion::operator++(int){
    Posicion aux(*this);
    i++;
    return aux;
}
```

```
//Operador -- prefijo
```

```
Posicion& Posicion::operator--(){
    i--;
    return *this;
}
```

```
//Operador -- postfijo
```

```
Posicion Posicion::operator--(int){
    Posicion aux(*this);
    i--;
    return aux;
}
```

```
bool Posicion::operator==(const
Posicion& p) const{
    return i==p.i;
}
```

```
bool Posicion::operator!=(const
Posicion& p) const{
    return i!=p.i;
}
```

# Lista.cpp

```
//Clase Lista
```

```
Lista::Lista(){  
    nelementos = 0;  
    reservar(1);  
}
```

```
Lista::Lista(const Lista& l){  
    reservar(l.nelementos);  
    copiar(l);  
}
```

```
Lista::~~Lista(){  
    liberar();  
}
```

```
Lista& Lista::operator=(const Lista &l){  
    if (this != &l){  
        liberar();  
        reservar(l.nelementos);  
        copiar(l);  
    }  
    return *this;  
}
```



# Lista.cpp

```
void Lista::set(const Posicion & p, const Tbase &e){  
    assert(p.i>=0 && p.i<nelementos);  
    datos[p.i] = e;  
}
```

```
Tbase Lista::get(const Posicion & p) const{  
    assert(p.i>=0 && p.i<nelementos);  
    return datos[p.i];  
}
```

```
Posicion Lista::insert(const Posicion & p, const Tbase & e){  
    if(nelementos == reservados)  
        redimensionar(reservados*2);  
    for(int j=nelementos; j>p.i; j--)  
        datos[j] = datos[j-1];  
    datos[p.i] = e;  
    nelementos++;  
    return p;  
}
```

# Lista.cpp

```
Posicion Lista::erase(const Posicion & p){
    assert(p!=end());
    for(int j=p.i; j<nelementos-1; j++)
        datos[j] = datos[j+1];
    nelementos--;
    if (nelementos<reservados/4)
        redimensionar(reservados/2);
    return p;
}
```

```
int Lista::size() const{
    return nelementos;
}
```

```
Posicion Lista::begin() const{
    Posicion p;
    p.i = 0; //Innecesario
    return p;
}
```

```
Posicion Lista::end() const{
    Posicion p;
    p.i = nelementos;
    return p;
}
```

# Lista.cpp

//Métodos auxiliares privados

```
void Lista::liberar(){
    delete []datos;
    reservados = nelementos = 0;
}

void Lista::reservar(int n){
    assert(n>0);
    reservados = n;
    datos = new Tbase[reservados];
}

void Lista::copiar(const Lista& l){
    nelementos = l.nelementos;
    for(int i=0; i<nelementos; i++)
        datos[i] = l.datos[i];
}

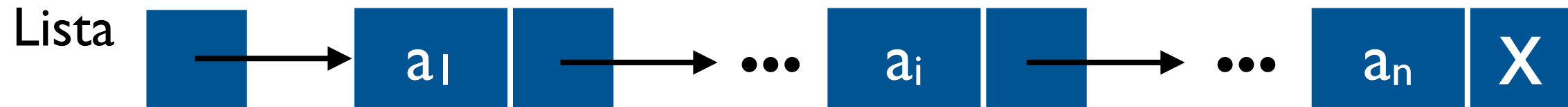
void Lista::redimensionar(int n){
    assert(n>0 && n>=nelementos);
    Tbase* aux = datos;
    reservar(n);
    for(int i=0; i<nelementos; i++)
        datos[i] = aux[i];
    // memcpy(datos, aux, nelementos*sizeof(Tbase));
    delete[] aux;
}
```

Ejercicios propuestos:

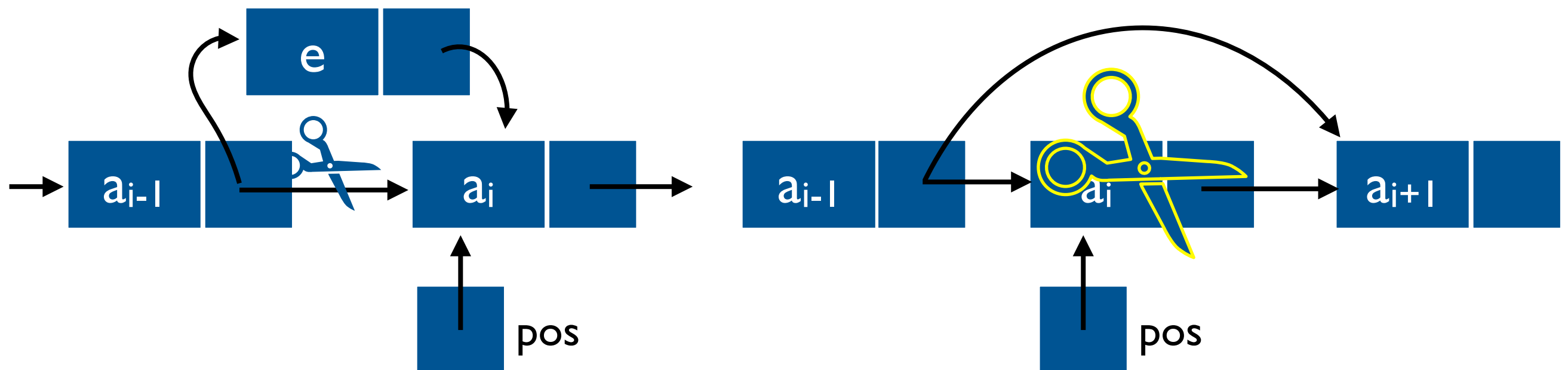
- Desarrollar una clase Lista genérica con templates
- Revisar el programa y las funciones de las transparencias 8-11 para usar listas genéricas

# Listas. Celdas enlazadas

Almacenamos la secuencia de valores en celdas enlazadas

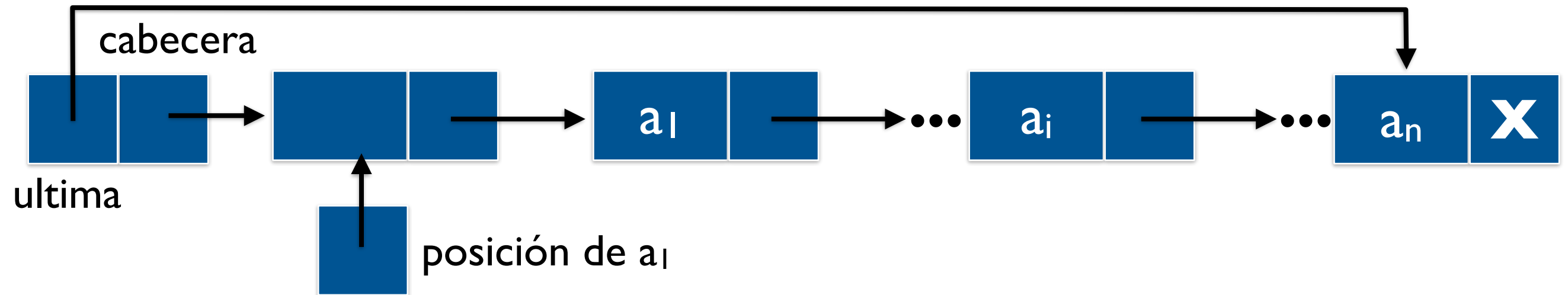


- Una lista es un puntero a la primera celda (si no está vacía)
- Una posición son dos punteros. El segundo (no se muestra) es necesario para algunos operadores
- Inserciones/borrados en la primera posición son casos especiales



# Listas. Celdas enlazadas con cabecera

Almacenamos la secuencia de valores en celdas enlazadas



- Una lista es un puntero a la cabecera (si está vacía, sólo tiene una celda)
- Una posición son dos punteros. El segundo (no se muestra) es necesario para algunos operadores
- La posición de un elemento es un puntero a la celda anterior
- Inserciones/borrados la primera posición **NO** son casos especiales

# Lista.h

```
#ifndef __LISTA_H__
#define __LISTA_H__

typedef char Tbase;

struct CeldaLista{
    Tbase elemento;
    CeldaLista* siguiente;
};

class Lista;

class Posicion{
private:
    CeldaLista* puntero;
    CeldaLista* primera;
public:
    Posicion();
    Posicion(const Posicion& p) = default;
    ~Posicion() = default;
    Posicion& operator=(const Posicion& p) = default;
    Posicion& operator++();
    Posicion operator++(int);
    Posicion& operator--();
    Posicion operator--(int);
    bool operator==(const Posicion& p) const;
    bool operator!=(const Posicion& p) const;
    friend class Lista;
};
```

# Lista.h

```
class Lista{
private:
    CeldaLista* cabecera;
    CeldaLista* ultima;
public:
    Lista();
    Lista(const Lista& l);
    ~Lista();
    Lista& operator=(const Lista& l);

    Tbase get(Posicion p) const;
    void set(Posicion p, Tbase e);
    Posicion insert(Posicion p, Tbase e);
    Posicion erase(Posicion p);
    int size() const;
    Posicion begin() const;
    Posicion end() const;
};

#endif // __LISTA_H__
```

# Lista.cpp

```
#include <cassert>
#include <utility>
#include "Lista.hpp"
using namespace std;
```

```
//Clase Posicion
```

```
Posicion::Posicion(){
    primera = puntero = 0;
}
```

```
//Operador ++ prefijo
```

```
Posicion& Posicion::operator++(){
    puntero = puntero->siguiente;
    return *this;
}
```

```
//Operador ++ postfijo
```

```
Posicion Posicion::operator++(int){
    Posicion p(*this);
    //operator++();
    ++(*this);
    return p;
}
```



# Lista.cpp

```
//Operador -- prefijo
Posicion& Posicion::operator--(){
    assert(puntero!=primera);
    CeldaLista* aux = primera;
    while(aux->siguiente!=puntero){
        aux = aux->siguiente;
    }
    puntero = aux;
    return *this;
}

//Operador -- postfijo
Posicion Posicion::operator--(int){
    Posicion p(*this);
    //operator--();
    --(*this);
    return p;
}

bool Posicion::operator==(const Posicion & p) const{
    return(puntero==p.puntero);
}

bool Posicion::operator!=(const Posicion &p) const{
    return(puntero!=p.puntero);
}
```

# Lista.cpp

```
//Clase Lista
```

```
Lista::Lista(){
    ultima = cabecera = new CeldaLista;
    cabecera->siguiente = 0;
}

Lista::Lista(const Lista& l){
    ultima = cabecera = new CeldaLista;
    CeldaLista* orig = l.cabecera;
    while(orig->siguiente!=0){
        ultima->siguiente = new CeldaLista;
        ultima = ultima->siguiente;
        orig = orig->siguiente;
        ultima->elemento = orig->elemento;
    }
    ultima->siguiente = 0;
}

Lista::~~Lista(){
    CeldaLista* aux;
    while(cabecera!=0){
        aux = cabecera;
        cabecera = cabecera->siguiente;
        delete aux;
    }
}
```

# Lista.cpp

```
Lista& Lista::operator=(const Lista& l){  
    Lista aux(l);  
    swap(cabecera, aux.cabecera);  
    swap(ultima, aux.ultima);  
    return *this;  
}
```

← Función de intercambio de valores  
de la biblioteca estándar de C++  
(incluida en <utility>)

```
void Lista::set(Posicion p, Tbase e){  
    p.puntero->siguiente->elemento = e;  
}
```

```
Tbase Lista::get(Posicion p) const{  
    return p.puntero->siguiente->elemento;  
}
```

```
Posicion Lista::insert(Posicion p, Tbase e){  
    CeldaLista* nueva = new CeldaLista;  
    nueva->elemento = e;  
    nueva->siguiente = p.puntero->siguiente;  
    p.puntero->siguiente = nueva;  
    if(p.puntero == ultima)  
        ultima = nueva;  
    return p;  
}
```

# Lista.cpp

```
Posicion Lista::erase(Posicion p){
    assert(p!=end());
    CeldaLista* aux = p.puntero->siguiente;
    p.puntero->siguiente = aux->siguiente;
    if(aux==ultima)
        ultima = p.puntero;
    delete aux;
    return p;
}
```

```
int Lista::size() const{
    int n=0;
    for(Posicion p=begin(); p!=end(); ++p)
        n++;
    return n;
}
```

```
Posicion Lista::begin()const{
    Posicion p;
    p.puntero = p.primer = cabecera;
    return p;
}
```

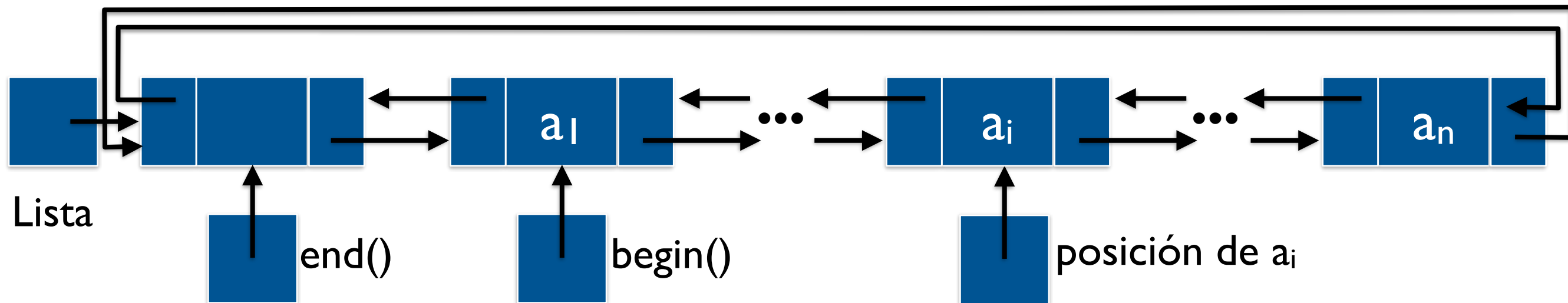
```
Posicion Lista::end() const{
    Posicion p;
    p.puntero = ultima;
    p.primer = cabecera;
    return p;
}
```

Ejercicio propuesto:

- Desarrollar una clase Lista genérica con templates

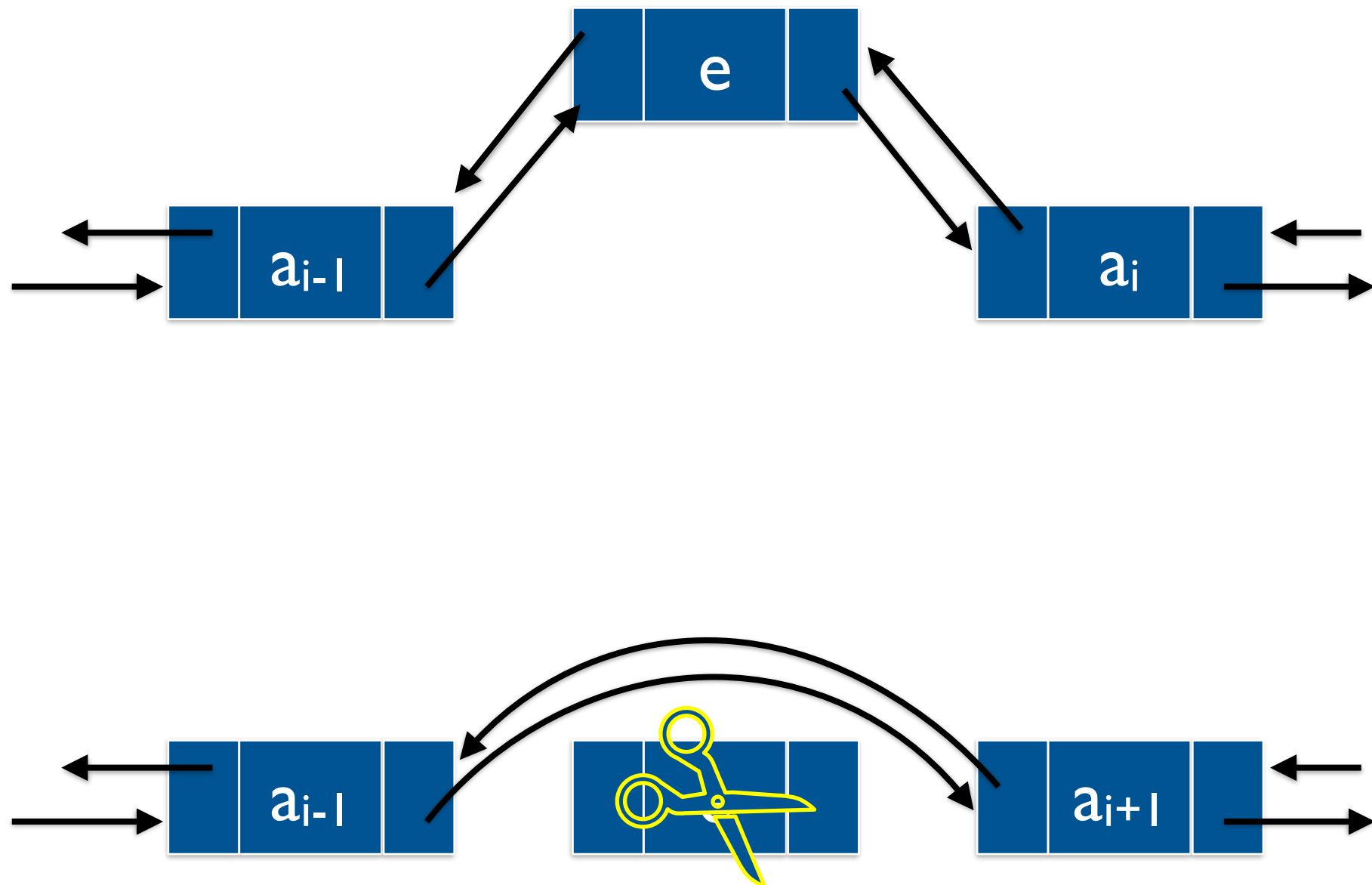
# Listas. Celdas doblemente enlazadas circulares

Almacenamos la secuencia de valores en celdas doblemente enlazadas



- Una lista es un puntero a la cabecera (si está vacía, sólo tiene una celda)
- Una posición es un único puntero a la celda
- Inserciones/borrados son independientes de la posición

# Listas. Celdas doblemente enlazadas circulares



# Lista.h

```
#ifndef __LISTA_H__
#define __LISTA_H__

typedef char Tbase;

struct CeldaLista{
    Tbase elemento;
    CeldaLista* anterior;
    CeldaLista* siguiente;
};

class Lista;

class Posicion{
private:
    CeldaLista* puntero;
public:
    Posicion();
    Posicion(const Posicion& p) = default;
    ~Posicion() = default;
    Posicion& operator=(const Posicion& p) = default;
    Posicion& operator++();
    Posicion operator++(int);
    Posicion& operator--();
    Posicion operator--(int);
    bool operator==(const Posicion& p) const;
    bool operator!=(const Posicion& p) const;
    friend class Lista;
};
```

# Lista.h

```
class Lista{
private:
    CeldaLista* cabecera;
public:
    Lista();
    Lista(const Lista& l);
    ~Lista();
    Lista& operator=(const Lista& l);
    void set(Posicion p, Tbase e);
    Tbase get(Posicion p) const;
    Posicion insert(Posicion p, Tbase e);
    Posicion erase(Posicion p);
    int size() const;
    Posicion begin() const;
    Posicion end() const;
};
#endif //__LISTA_H__
```



# Lista.cpp

```
#include <cassert>
#include "Lista.hpp"
using namespace std;
```

```
//Clase Posicion
```

```
Posicion::Posicion(){
    puntero = 0;
}
```

```
//Operador ++ prefijo
```

```
Posicion& Posicion::operator++(){
    puntero = puntero->siguiente;
    return *this;
}
```

```
//Operador ++ postfijo
```

```
Posicion Posicion::operator++(int){
    Posicion p(*this);
    ++(*this);
    return p;
}
```

```
//Operador -- prefijo
```

```
Posicion& Posicion::operator--(){
    puntero = puntero->anterior;
    return *this;
}
```

```
//Operador -- postfijo
```

```
Posicion Posicion::operator--(int){
    Posicion p(*this);
    --(*this);
    return p;
}
```

```
bool Posicion::operator==(const
Posicion& p) const{
    return (puntero==p.puntero);
}
```

```
bool Posicion::operator!=(const
Posicion& p) const{
    return (puntero!=p.puntero);
}
```

# Lista.cpp

```
Lista::Lista(){
    cabecera = new CeldaLista;           //Creamos la cabecera
    cabecera->siguiente = cabecera;       //Ajustamos punteros
    cabecera->anterior = cabecera;
}

Lista::Lista(const Lista& l){
    cabecera = new CeldaLista;           //Creamos la celda cabecera
    cabecera->siguiente = cabecera;
    cabecera->anterior = cabecera;

    CeldaLista* p = l.cabecera->siguiente; //Recorremos la lista y copiamos
    while(p!=l.cabecera){                  //Hasta "dar la vuelta completa"
        CeldaLista* q = new CeldaLista;    //Creamos una nueva celda
        q->elemento = p->elemento;           //Copiamos la información
        q->anterior = cabecera->anterior;   //Ajustamos punteros
        cabecera->anterior->siguiente = q;
        cabecera->anterior = q;
        q->siguiente = cabecera;
        p = p->siguiente;                  //Avanzamos en l
    }
}

Lista::~~Lista(){
    while(begin()!=end())                  //Mientras no esté vacía
        erase(begin());                   //Borramos la primera celda
    delete cabecera;                      //Borramos la cabecera
}
```

# Lista.cpp

```
Lista& Lista::operator=(const Lista &l){  
    Lista aux(l); //Usamos constructor de copia  
    swap(this->cabecera, aux.cabecera); //Intercambiamos punteros  
    return *this;  
    //Al salir se destruye aux, que tiene el antiguo contenido de *this  
}
```

```
void Lista::set(Posicion p, Tbase e){  
    p.puntero->elemento = e;  
}
```

```
Tbase Lista::get(Posicion p) const{  
    return p.puntero->elemento;  
}
```

```
Posicion Lista::insert(Posicion p, Tbase e){  
    CeldaLista* q = new CeldaLista; //Creamos la celda  
    q->elemento = e; //Almacenamos la información  
    q->anterior = p.puntero->anterior; //Ajustamos punteros  
    q->siguiente = p.puntero;  
    p.puntero->anterior = q;  
    q->anterior->siguiente = q;  
    p.puntero = q;  
    return p;  
}
```

# Lista.cpp

```
Posicion Lista::erase(Posicion p){
    assert(p!=end());
    CeldaLista* q = p.puntero;
    q->anterior->siguiente = q->siguiente;
    q->siguiente->anterior = q->anterior;
    p.puntero = q->siguiente;
    delete q;
    return p;
}

int Lista::size() const{
    int n = 0;
    for(Posicion p=begin(); p!= end(); p++)
        n++;
    return n;
}

Posicion Lista::begin() const{
    Posicion p;
    p.puntero = cabecera->siguiente;
    return p;
}

Posicion Lista::end() const{
    Posicion p;
    p.puntero = cabecera;
    return p;
}
```

Ejercicio propuesto:

- Desarrollar una clase Lista genérica con templates

# Listas

## Uso de una lista **STL**

```
#include <iostream>
#include <list>
using namespace std;
int main() {
    char dato;
    list<char> l, aux;
    cout << "Escriba una frase" << endl;
    while((dato=cin.get())!='\n')
        l.insert(l.end(), dato); //l.emplace_back(dato);
    cout << "La frase introducida es:" << endl;
    cout << l;
    cout << "La frase en minúsculas:" << endl;
    escribir_minuscula(l);
    if(localizar(l, ' ')==l.end())
        cout << "La frase no tiene espacios" << endl;
    else{
        cout << "La frase sin espacios:" << endl;
        aux = l;
        aux.remove(' ');
        cout << aux;
    }
    cout << "La frase al revés: " << endl;
    aux = l;
    aux.reverse();
    cout << aux;
    cout << (palindromo(l) ? "Es " : "No es ") << "un palíndromo" << endl;
    return 0;
}
```

# Listas

## Uso de una lista **STL**

```
void todo_minuscula(list<char>& l){
    list<char>::iterator p;
    for(p=l.begin(); p!=l.end(); p++)
        *p = tolower(*p);
}
```

```
ostream& operator<<(ostream & flujo, const list<char>& l){
    list<char>::const_iterator p;
    for(p=l.cbegin(); p!=l.cend(); p++)
        flujo << *p;
    flujo << endl;
    return flujo;
}
```

```
void escribir_minuscula(list<char> l){
    todo_minuscula(l);
    cout << l;
}
```

```
template <class T>
typename list<T>::iterator localizar(list<T> l, const T &c){
    typename list<T>::iterator p;
    for(p=l.begin(); p!=l.end(); p++)
        if(*p==c)
            return(p);
    return l.end();
}
```

# Listas

Uso de una lista  
**STL**

```
bool palindromo(const list<char>& l){  
    list<char> aux(l);  
    int n = l.size();  
    if(n<2)  
        return true;  
    aux.remove(' ');  
    todo_minuscula(aux);
```

```
list<char>::const_iterator p1, p2;  
p1 = aux.begin();  
p2 = aux.end();  
--p2;  
for(int i=0; i<n/2; i++){  
    if(*p1 != *p2)  
        return false;  
    ++p1;  
    --p2;  
}  
return true;  
}
```

# Listas. Ejercicios propuestos

- Ampliar la clase lista para que ofrezca las operaciones que ofrece la clase list de la STL:

```
Lista(const Posicion inicio, const Posicion final);  
void clear();  
void push_front(const Tbase & dato);  
void push_back(const Tbase & dato);  
void pop_front();  
void pop_back();  
Tbase & front();  
const Tbase & front() const;  
Tbase & back();  
const Tbase & back() const;  
void swap(Lista & l);
```

en su implementación con celdas doblemente enlazadas. Lo ideal sería hacerlo sobre la versión genérica (con templates) de esta implementación.



# Listas. Ejercicios propuestos

- Desarrollar iteradores para la implementación de la clase lista basada en celdas doblemente enlazadas, de forma que podamos usarlos como en la STL:

```
void todo_minuscula(Lista<char>& l){  
    for(typename Lista<char>::iterator it=l.begin(); it!=l.end(); ++it)  
        //l.set(it, tolower(l.get(it)));  
        *it = tolower(*it);  
}
```

```
template <class T>  
ostream& operator<<(ostream & flujo, const Lista<T>& l){  
    for(typename Lista<T>::const_iterator it=l.cbegin(); it!=l.cend(); ++it)  
        //cout << l.get(it);  
        cout << *it;  
    cout << endl;  
    return flujo;  
}
```

# Listas. Ejercicios propuestos

- Operaciones a implementar en el iterador (tanto en su versión const como en la no const):
  - Constructor por defecto
  - Constructor de copia
  - Operador de asignación (=)
  - Operador de indirección (\*)
  - Operadores de comparación (== y !=)
  - Operadores de incremento/decremento (++ y --), tanto prefijos como postfijos
  - (En la clase Lista) Métodos begin() y end(), cbegin() y cend()