


ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Iteradores constantes

Manuel Montenegro Montes  
Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid



# Ejemplo

- Volvemos al ejemplo de la suma de una lista de enteros:

```
int suma_elems(ListLinkedList<int> &l) {  
    int suma = 0;  
    for (ListLinkedList<int>::iterator it = l.begin();  
         it != l.end();  
         it.advance()) {  
        suma += it.elem();  
    }  
    return suma;  
}
```

# ¿Podemos pasar `l` por referencia constante?

- No, porque `begin()` y `end()` no son métodos constantes.

```
int suma_elems(const ListLinkedList<int> &l) {  
    int suma = 0;  
    for (ListLinkedList<int>::iterator it = l.begin();  
         it ≠ l.end();  
         it.advance()) {  
        suma += it.elem();  
    }  
    return suma;  
}
```

# ¿Y si `begin()` y `end()` fueran constantes?

```
template <typename T>
class ListLinkedList {
public:
    ...
    iterator begin() const;
    iterator end() const;
};
```

- Técnicamente, el compilador no se queja.
- Pero devuelven un `iterator`, a través del cual yo puedo modificar los elementos de la lista.

# ¿Por qué iterator puede modificar los elementos de la lista?

```
class iterator {  
public:  
    void advance();  
    T & elem();  
    bool operator==(const iterator &other) const;  
    bool operator!=(const iterator &other) const;  
    ...  
};
```

- Porque `elem()` devuelve una referencia al elemento apuntado por el iterator.
- A partir de esa referencia puedo cambiar el valor de ese elemento.

# ¿Y si elem() devolviera una referencia constante?

```
class iterator {  
public:  
    void advance();  
    const T & elem();  
    bool operator==(const iterator &other) const;  
    bool operator!=(const iterator &other) const;  
    ...  
};
```

# ¿Y si elem() devolviera una referencia constante?

- Ya no podría tener programas como este:

```
void multiplicar_por(ListLinkedListDouble<int> &l, int num) {  
    for (ListLinkedListDouble<int>::iterator it = l.begin();  
        it != l.end();  
        it.advance()) {  
        it.elem() = it.elem() * num;  
    }  
}
```

# Moraleja

- A veces quiero que los iteradores me devuelvan referencias no constantes, porque quiero utilizar un iterador para alterar los elementos de la lista.

```
void multiplicar_por(ListLinkedDouble<int> &l, int num) {  
    for (ListLinkedDouble<int>::iterator it = l.begin();  
        it != l.end();  
        it.advance()) {  
        it.elem() = it.elem() * num;  
    }  
}
```



# Moraleja

- A veces quiero que los iteradores me devuelvan referencias no constantes, porque quiero utilizar un iterador para alterar los elementos de la lista.
- Otras veces quiero iterar sobre una lista sin modificar sus elementos.

```
int suma_elems(const ListLinkedDouble<int> &l) {  
    int suma = 0;  
    for (ListLinkedDouble<int>::iterator it = l.begin();  
         it != l.end();  
         it.advance()) {  
  
        suma += it.elem();  
    }  
    return suma;  
}
```

# Moraleja

- A veces quiero que los iteradores me devuelvan referencias no constantes, porque quiero utilizar un iterador para alterar los elementos de la lista.
- Otras veces quiero iterar sobre una lista sin modificar sus elementos.
- Tengo que distinguir dos tipos de iteradores:
  - **Iteradores no constantes** (`iterator`)
  - **Iteradores constantes** (`const_iterator`)

# Iteradores constantes y no constantes

```
class iterator {  
public:  
    void advance();  
    T & elem();  
    bool operator==(const iterator &other) const;  
    bool operator!=(const iterator &other) const;  
    ...  
};
```

```
class const_iterator {  
public:  
    void advance();  
    const T & elem();  
    bool operator==(const iterator &other) const;  
    bool operator!=(const iterator &other) const;  
    ...  
};
```

Las implementaciones  
de ambas clases  
son exactamente  
iguales

# Iteradores constantes y no constantes

```
template <typename T>
class ListLinkedDouble {
public:
    ...
    iterator begin();
    iterator end();

    const_iterator cbegin() const;
    const_iterator cend() const;

};
```

Funciones no constantes.  
Devuelven iteradores que  
me permiten modificar la lista.

Funciones constantes.  
Garantizan que no puedo  
modificar la lista con el  
iterador que devuelvan.

# Consecuencias

- Podemos utilizar iteradores para modificar elementos de la lista.
- Para ello utilizamos la clase `iterator`, como siempre.

```
void multiplicar_por(ListLinkedDouble<int> &l, int num) {  
    for (ListLinkedDouble<int>::iterator it = l.begin();  
        it != l.end();  
        it.advance()) {  
        it.elem() = it.elem() * num;  
    }  
}
```

# Consecuencias

- Podemos utilizar iteradores para recorrer la lista sin modificarla, y así poder declarar el objeto correspondiente como constante.
- Para ello utilizamos la clase `const_iterator`, y los métodos `cbegin()` y `cend()`.

```
int suma_elems(const ListLinkedDouble<int> &l) {  
    int suma = 0;  
    for (ListLinkedDouble<int>::const_iterator it = l.cbegin();  
         it != l.cend();  
         it.advance()) {  
  
        suma += it.elem();  
    }  
    return suma;  
}
```

# Cuánta duplicación, ¿no?

```
class iterator {  
public:  
    void advance();  
    T & elem();  
    bool operator==(const iterator &other) const;  
    bool operator!=(const iterator &other) const;  
    ...  
};
```

```
class const_iterator {  
public:  
    void advance();  
    const T & elem();  
    bool operator==(const iterator &other) const;  
    bool operator!=(const iterator &other) const;  
    ...  
};
```

Sólo difieren en  
el tipo de retorno  
de elem()!

# Cuánta duplicación, ¿no?

- Podemos utilizar las plantillas de C++:

```
template <typename U>
class gen_iterator {
public:
    void advance();
    U & elem();
    bool operator==(const gen_iterator &other) const;
    bool operator!=(const gen_iterator &other) const;
    ...
};
```

En iteradores no constantes:  $U = T$

En iteradores constantes:  $U = \text{const } T$



# Cuánta duplicación, ¿no?

- Podemos utilizar las plantillas de C++:

```
template <typename U>
class gen_iterator {
public:
    void advance();
    U & elem();
    bool operator==(const gen_iterator &other) const;
    bool operator!=(const gen_iterator &other) const;
    ...
};
```

```
using iterator = gen_iterator<T>;
using const_iterator = gen_iterator<const T>;
```

# En resumen, tenemos

```
template <typename T>
class ListLinkedListDouble {
public:
    ...

    template <typename U>
    class gen_iterator { ... }

    using iterator = gen_iterator<T>;
    using const_iterator = gen_iterator<const T>;

    iterator begin();
    iterator end();

    const_iterator cbegin() const;
    const_iterator cend() const;

};
```

