


ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

Adaptando la sintaxis de los iteradores

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



Recordatorio: suma de elementos

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

Cambio de sintaxis

```
int suma_elems(const ListLinkedListDouble<int> &l) {  
    int suma = 0;  
    for (ListLinkedListDouble<int>::const_iterator it = l.cbegin();  
        it ≠ l.cend();  
        ++it) {  
        suma += *it;  
    }  
    return suma;  
}
```

Sobrecarga del operador *



Sobrecarga del operador *

- El operador * tiene dos significados en C++:
 - Multiplicación (binario). Ejemplo: `5 * x`
 - Indirección (unario). Ejemplo: `*y`

**Queremos sobrecargar
este**

Sobrecarga del operador *

Antes

```
template <typename U>
class gen_iterator {
public:

...

    U & elem() const {
        assert (current != head);
        return current->value;
    }

};
```

Ahora

```
template <typename U>
class gen_iterator {
public:

...

    U & operator*() const {
        assert (current != head);
        return current->value;
    }

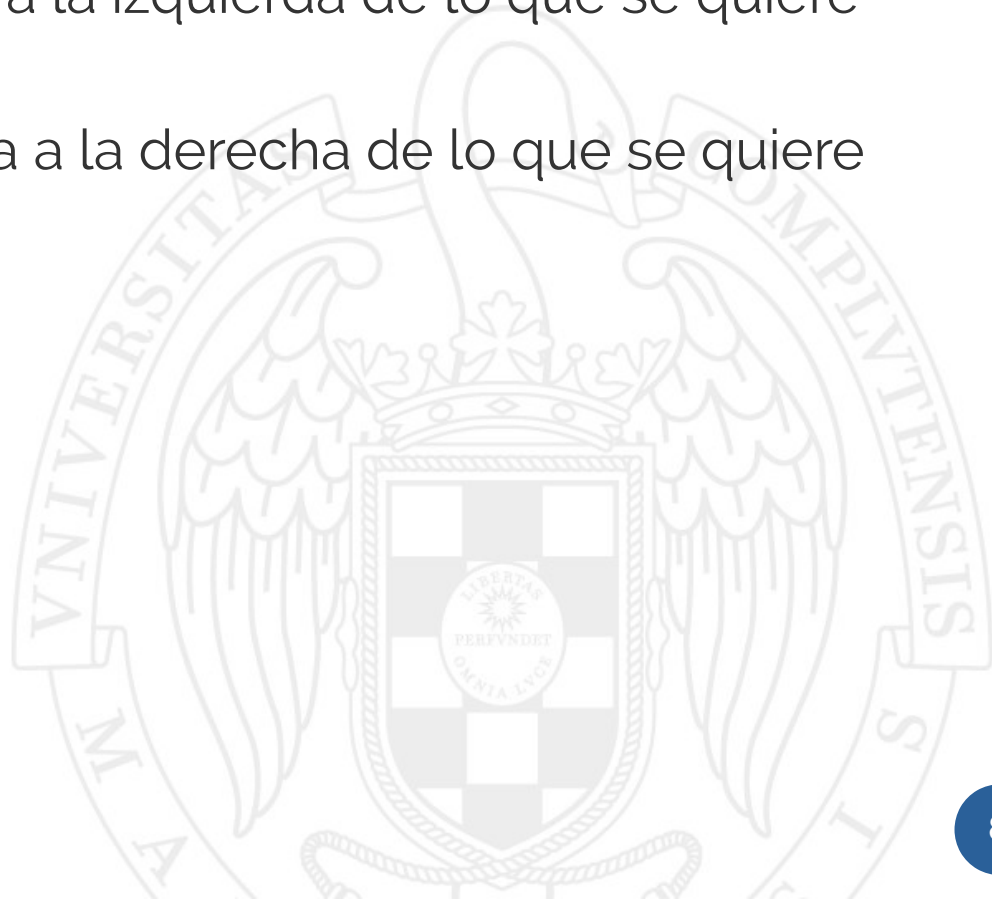
};
```

Sobrecarga del operador ++



Sobrecarga del operador ++

- El operador de incremento ++ también tiene dos significados en C++:
 - Preincremento, cuando se sitúa a la izquierda de lo que se quiere incrementar. Ejemplo: ++x
 - Postincremento, cuando se sitúa a la derecha de lo que se quiere incrementar. Ejemplo: x++



Preincremento vs postincremento

- Aplicados a tipos numéricos, ambos incrementan en una unidad el valor de la variable a la que se aplican.
- Pero:
 - El operador de preincremento devuelve el valor de la variable **tras** haberla incrementado.
 - El operador de postincremento devuelve el valor de la variable **antes de** haberla incrementado.

```
int x = 2;  
int z = ++x;  
// x vale 3, z vale 3
```

```
int x = 2;  
int z = x++;  
// x vale 3, z vale 2
```

¿Tanto nos interesa esto?

- En general no, porque casi siempre utilizamos las expresiones de incremento de manera aislada, sin asignar el valor resultante:

```
while (x < 10) {  
    // ...  
    x++;  
}
```

```
for (int i = 0; i < 10; i++) {  
    // ...  
}
```

- PERO... tenemos que conocer esta distinción a la hora de sobrecargar los operadores, para saber qué tenemos que devolver:
 - `it++` devuelve el iterador antes de haberlo avanzado.
 - `++it` devuelve el iterador tras haberlo avanzado.

Sobrecarga del operador ++

Antes

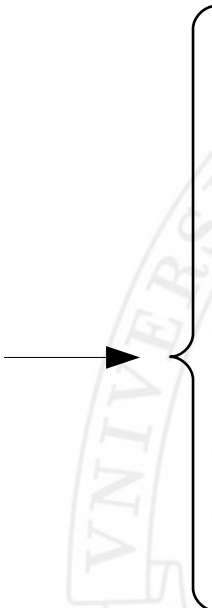
```
template <typename U>
class gen_iterator {
public:

    ...

    void advance() const {
        assert (current != head);
        current = current->next;
    }

};
```

Ahora



```
gen_iterator & operator++() {
    assert (current != head);
    current = current->next;
    return *this;
}

gen_iterator operator++(int) {
    assert (current != head);
    gen_iterator antes = *this;
    current = current->next;
    return antes;
}
```

Otro ejemplo

- Multiplicar todos los elementos de una lista por dos:

```
void mult_por_dos(ListLinkedDouble<int> &l) {  
    for (ListLinkedDouble<int>::iterator it = l.begin();  
        it != l.end();  
        ++it) {  
        *it = *it * 2;  
    }  
}
```

El especificador de tipo `auto` (C++11)



El especificador de tipo auto

- Cuando declaramos e inicializamos una variable, podemos indicar que su tipo es auto.
- En este caso, C++ infiere el tipo de la variable a partir del valor con el que se inicializa.

```
auto suma = 0;
```

equivale a

```
int suma = 0;
```

```
auto nombre = "";
```

equivale a

```
const char *nombre = "";
```

Ejemplo

- Antes de utilizar auto:

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



Ejemplo

- Después de utilizar auto:

```
int suma_elems(const ListLinkedListDouble<int> &l) {  
    int suma = 0;  
    for (auto it = l.cbegin(); it != l.cend(); ++it) {  
        suma += *it;  
    }  
    return suma;  
}
```



Bucles for basados en rango (C++11)



Bucles for basados en rango

- C++11 introduce una sintaxis nueva en los bucles for:

```
for (tipo variable : expresion) {  
    cuerpo  
}
```

equivale (casi) a:

```
for (auto it = expresion.begin(); it != expresion.end(); ++it) {  
    tipo variable = *it;  
    cuerpo  
}
```

<https://en.cppreference.com/w/cpp/language/range-for>

Bucles for basados en rango

```
int suma_elems(const ListLinkedList<int> &l) {  
    int suma = 0;  
    for (int x : l) {  
        suma += x;  
    }  
    return suma;  
}
```



Bucles for basados en rango

```
int suma_elems(const ListLinkedList<int> &l) {  
    int suma = 0;  
    for (int x : l) {  
        suma += x;  
    }  
    return suma;  
}
```

```
int suma_elems(const ListLinkedList<int> &l) {  
    int suma = 0;  
    for (auto it = l.begin(); it != l.end(); ++it) {  
        int x = *it;  
        suma += x;  
    }  
    return suma;  
}
```

Bucles for basados en rango

- Multiplicar todos los elementos de una lista por dos:

```
void mult_por_dos(ListLinkedListDouble<int> &l) {  
    for (int &x : l) {  
        x *= 2;  
    }  
}
```

