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 ListLinkedDouble<int> &l) {
  int suma = 0;
  for (ListLinkedDouble<int>::const_iterator it = l.cbegin();
       it != l.cend();
       it.advance()) {

    suma += it.elem();
  }
  return suma;
}
```

debemos aprender esta sintaxis rodeada.

Cambio de sintaxis

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

avanzar el operador usamos incremento de ese operador

En lugar de ser un puntero es un iterador.

Para ello vamos a necesitar SOBRECARGAR el operador * y ++
Para que puedan soportar iteradores.

Sobrecarga del operador *

Primero el operador asterisco.

Sobrecarga del operador *

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

Queremos sobrecargar este

Para ACCEDER AL VALOR APUNTADO POR UN PUNTERO. LO QUE QUEREMOS ES SOBRECARGARLO PARA QUE DEVUELVA EL VALOR APUNTADO POR EL ITERADOR.

Sobrecarga del operador *

Antes

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

  ...

  U & elem() const {
    assert (current ≠ head);
    return current→value;
  }

};
```

it.elem()

CAMBIAMOS EL NOMBRE

Ahora

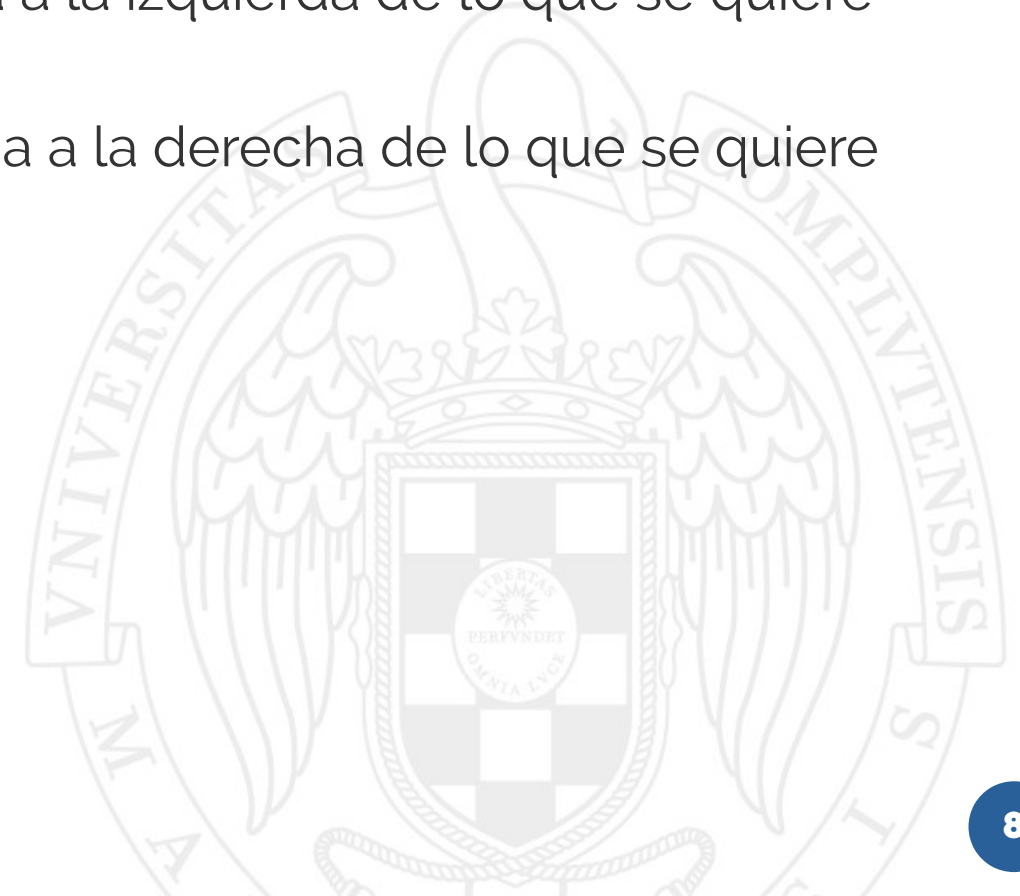
```
template <typename U>
class gen_iterator {
public:

  ...

  U & operator*() const {
    assert (current ≠ head);
    return current→value;
  }

};
```

*it

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
```

z tiene el valor de x DESPUÉS de ser incrementada.

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

z tiene el valor de x ANTES de ser incrementada. Antes de ser incrementada, x era 2

¿Tanto nos interesa esto?

- En geneneral 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;
   }

};
```

sustituimos advance

Ahora

operador de preincremento ++x

```
gen_iterator & operator++() {
  assert (current ≠ head);
  current = current→next;
  return *this;
}
```

iterador tras haber hecho el incremento

operador postincremento x++

```
gen_iterator operator++(int) {
  assert (current ≠ head);
  gen_iterator antes = *this;
  current = current→next;
  return antes;
}
```

Devuelve el iterador antes del incremento.

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 valor al que apunta el iterador es = al que apunta actualmente pero multiplicado *2

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

En teoría nos va a simplificar.

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.

al poner auto suma =0, lo reconoce como un entero.

```
auto suma = 0;
```

equivale a

```
int suma = 0;
```

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

equivale a

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

Ejemplo

- Antes de utilizar `auto`:

Él no suele ser muy devoto de usar auto

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

    suma += *it;
  }
  return suma;
}
```

EN ESTE CASO SI QUE PODRÍAMOS HACER EL AUTO.

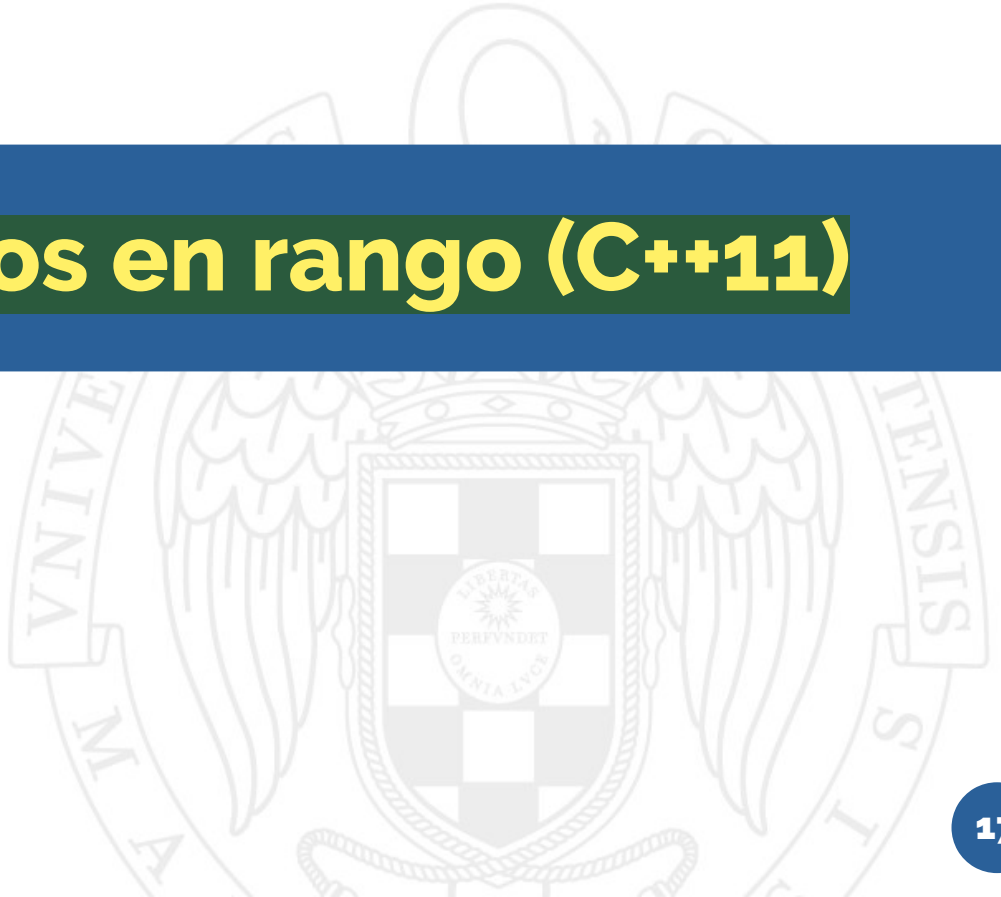
ESTO ES DEMASIADO LARGO.

Ejemplo

- Después de utilizar `auto`:

```
int suma_elems(const ListLinkedDouble<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
}
```

Esto lo hemos visto en TP2 Y 1 y lo hemos utilizado bastante.

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

Significa que recorra todos los elementos de "L", en cada iteración, x hace referencia al elemento actual

Bucles for basados en rango

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

```
int suma_elems(const ListLinkedDouble<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(ListLinkedDouble<int> &l) {
  for (int &x : l) {
    x *= 2;
  }
}
```

tipo referencia a int porque si no estaríamos modificando una copia pero no el elemento original de la lista.