

ESTRUCTURAS DE DATOS

INTRODUCCIÓN A LOS TIPOS ABSTRACTOS DE DATOS

# TADs: motivación

Manuel Montenegro Montes

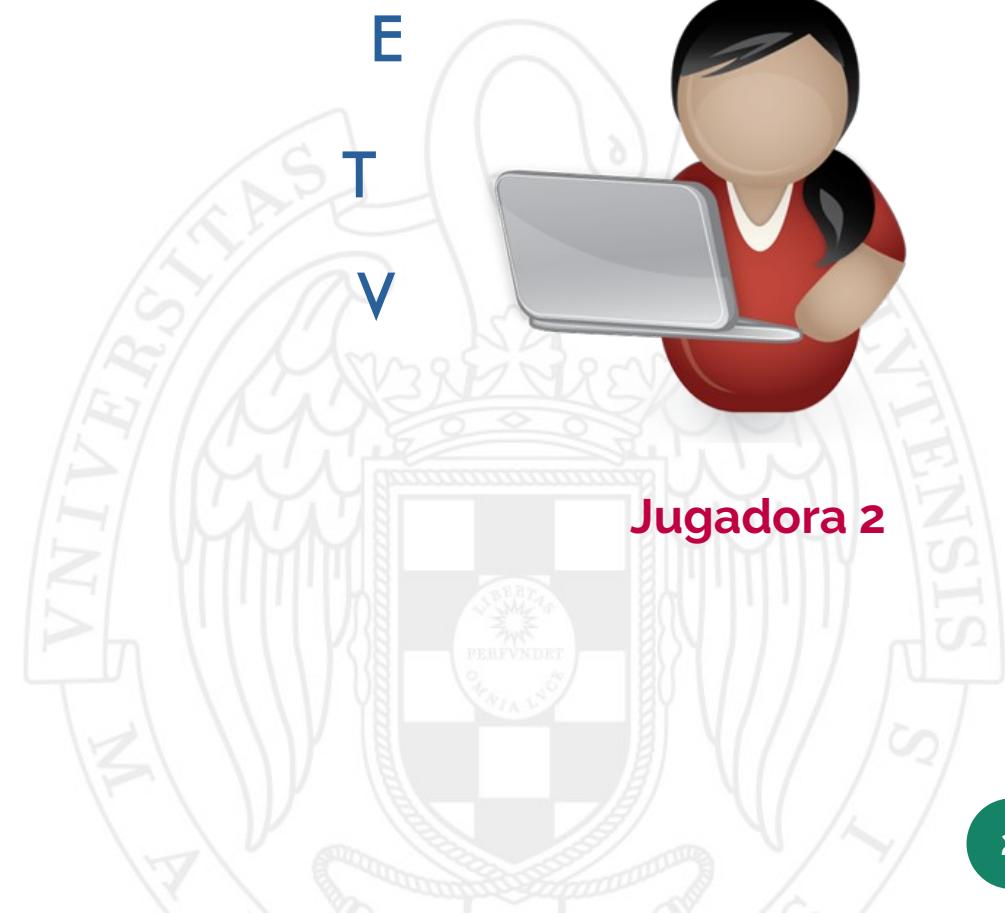
Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Un pequeño juego



Jugador 1

N  
S  
O  
T



Jugadora 2

# Un pequeño juego



N  
S  
O  
T



E  
T  
V

- Para saber si una letra se ha dicho antes, debemos almacenar el conjunto de letras nombradas hasta el momento.

# Tipo de datos ConjuntoChar

```
const int MAX_CHARS = 26;  
  
struct ConjuntoChar {  
    int num_chars;  
    char elementos[MAX_CHARS];  
};
```

- Suponemos que solo se admiten las letras mayúsculas del alfabeto inglés (A-Z).
  - Son un total de 26 letras.
- Guardamos las letras nombradas hasta el momento en el array **elementos**.
- Las primeras **num\_chars** posiciones tienen letras. El resto se consideran posiciones “vacías”.

# Función auxiliar: esta\_en\_conjunto

- Determina si el conjunto contiene la letra c pasada como parámetro.

```
bool esta_en_conjunto(char c, const ConjuntoChar &conjunto) {  
    int i = 0;  
    while (i < conjunto.num_chars && conjunto.elementos[i] != c) {  
        i++;  
    }  
    return conjunto.elementos[i] == c;  
}
```

# Implementación inicial del juego

```
int main() {
    int jugador_actual = 1;
    ConjuntoChar letras_nombradas;
    letras_nombradas.num_chars = 0;

    char letra_actual = preguntar_letra(jugador_actual);

    while (!esta_en_conjunto(letra_actual, letras_nombradas)) {
        letras_nombradas.elementos[letras_nombradas.num_chars] = letra_actual;
        letras_nombradas.num_chars++;

        jugador_actual = cambio_jugador(jugador_actual);
        letra_actual = preguntar_letra(jugador_actual);
    }

    std::cout << "Jugador " << jugador_actual << " ha perdido!" << std::endl;
    std::cout << "La letra repetida ha sido: " << letra_actual << std::endl;
    return 0;
}
```

# Funcionamiento



```
manuel@localhost:~/Trabajo/Docencia/ED/ejemplos/intro/juego_letras
manuel ... > ejemplos > intro > juego_letras > ./juegol
Jugador 1: B
Jugador 2: D
Jugador 1: A
Jugador 2: N
Jugador 1: X
Jugador 2: S
Jugador 1: H
Jugador 2: D
Jugador 2 ha perdido!
La letra repetida ha sido: D
manuel ... > ejemplos > intro > juego_letras >
```

# Cambios en la implementación

```
const int MAX_CHARS = 26;  
  
struct ConjuntoChar {  
    bool esta[MAX_CHARS];  
};
```

- Nuestro conjunto contiene un número limitado de letras.
- Podemos representar el contenido del conjunto como un array de booleanos.
  - Si la letra A está en el conjunto: `esta[0] = true`.
  - Si la letra B está en el conjunto: `esta[1] = true`.
  - ...

# Cambios en esta\_en\_conjunto

```
bool esta_en_conjunto(char c, const ConjuntoChar &conjunto) {  
    return esta[c - (int)'A'];  
}
```



# Implementación inicial del juego

```
int main() {  
    int jugador_actual = 1;  
    ConjuntoChar letras_nombradas;  
    letras_nombradas.num_chars = 0;   
  
    char letra_actual = preguntar_letra(jugador_actual);  
  
    while (!esta_en_conjunto(letra_actual, letras_nombradas)) {   
        letras_nombradas.elementos[letras_nombradas.num_chars] = letra_actual;   
        letras_nombradas.num_chars++;  
  
        jugador_actual = cambio_jugador(jugador_actual);  
        letra_actual = preguntar_letra(jugador_actual);  
    }  
  
    std::cout << "Jugador " << jugador_actual << " ha perdido!" << std::endl;  
    std::cout << "La letra repetida ha sido: " << letra_actual << std::endl;  
    return 0;  
}
```



# ¿Qué ha fallado?

- Cualquier cambio en el tipo de datos `ConjuntoChar` tiene que ser propagado hasta aquellos sitios en los que se utilicen dichos campos.
- La función `main()` menciona explícitamente los campos del tipo `ConjuntoChar`. Por tanto, se ve afectada por el cambio de la definición del tipo.
- Un cambio en la definición de un tipo de datos debe provocar el menor impacto posible en la implementación del resto del programa.
- ¿Cómo delimitamos las operaciones que pueden verse afectadas por este cambio?

**Abstracción mediante Tipos Abstractos de Datos (TADs)**

ESTRUCTURAS DE DATOS

INTRODUCCIÓN A LOS TIPOS ABSTRACTOS DE DATOS

# TADs: definición

Manuel Montenegro Montes

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

# ¿Qué hemos hecho mal?

```
int main() {  
    int jugador_actual = 1;  
    ConjuntoChar letras_nombradas;  
    letras_nombradas.num_chars = 0; // Linea 1  
  
    char letra_actual = preguntar_letra(jugador_actual);  
  
    while (!esta_en_conjunto(letra_actual, letras_nombradas)) {  
        letras_nombradas.elementos[letras_nombradas.num_chars] = letra_actual; // Linea 2  
        letras_nombradas.num_chars++;  
  
        jugador_actual = cambio_jugador(jugador_actual);  
        letra_actual = preguntar_letra(jugador_actual);  
    }  
  
    std::cout << "Jugador " << jugador_actual << " ha perdido!" << std::endl;  
    std::cout << "La letra repetida ha sido: " << letra_actual << std::endl;  
    return 0;  
}
```

La lógica del juego utiliza detalles relativos a la implementación de los conjuntos de caracteres

# ¿Qué hemos hecho mal?

```
int main() {  
    int jugador_actual = 1;  
    ConjuntoChar letras_nombradas;  
    letras_nombradas.num_chars = 0;  
  
    char letra_actual = preguntar_letra(jugador_actual);  
  
    while (!esta_en_conjunto(letra_actual, letras_nombradas)) {  
        letras_nombradas.elementos[letras_nombradas.num_chars] = letra_actual;  
        letras_nombradas.num_chars++;  
  
        jugador_actual = cambio_jugador(jugador_actual);  
        letra_actual = preguntar_letra(jugador_actual);  
    }  
  
    std::cout << "Jugador " << jugador_actual << " ha perdido!" << std::endl;  
    std::cout << "La letra repetida ha sido: " << letra_actual << std::endl;  
    return 0;  
}
```

Sin embargo, aquí sí lo hemos hecho bien...

# Abstrayendo los detalles



N  
S  
O  
T

Jugador 1



E  
T  
V

Jugadora 2

- El sitio en el que se guardan las letras nombradas hasta el momento se corresponde con la definición matemática de **conjunto**.

$$\text{LetrasNombradas} = \{N, E, S, O, T, V\}$$

# En un lenguaje ideal...

```
int main() {
    int jugador_actual = 1;
    LetrasNombradas = ∅;

    char letra_actual = preguntar_letra(jugador_actual);

    while (letra_actual ∈ LetrasNombradas) {
        LetrasNombradas = LetrasNombradas ∪ {letra_actual}

        jugador_actual = cambio_jugador(jugador_actual);
        letra_actual = preguntar_letra(jugador_actual);
    }

    std::cout << "Jugador " << jugador_actual << " ha perdido!" << std::endl;
    std::cout << "La letra repetida ha sido: " << letra_actual << std::endl;
    return 0;
}
```

# ¿Qué necesitamos de un conjunto?

- Obtener un conjunto vacío.

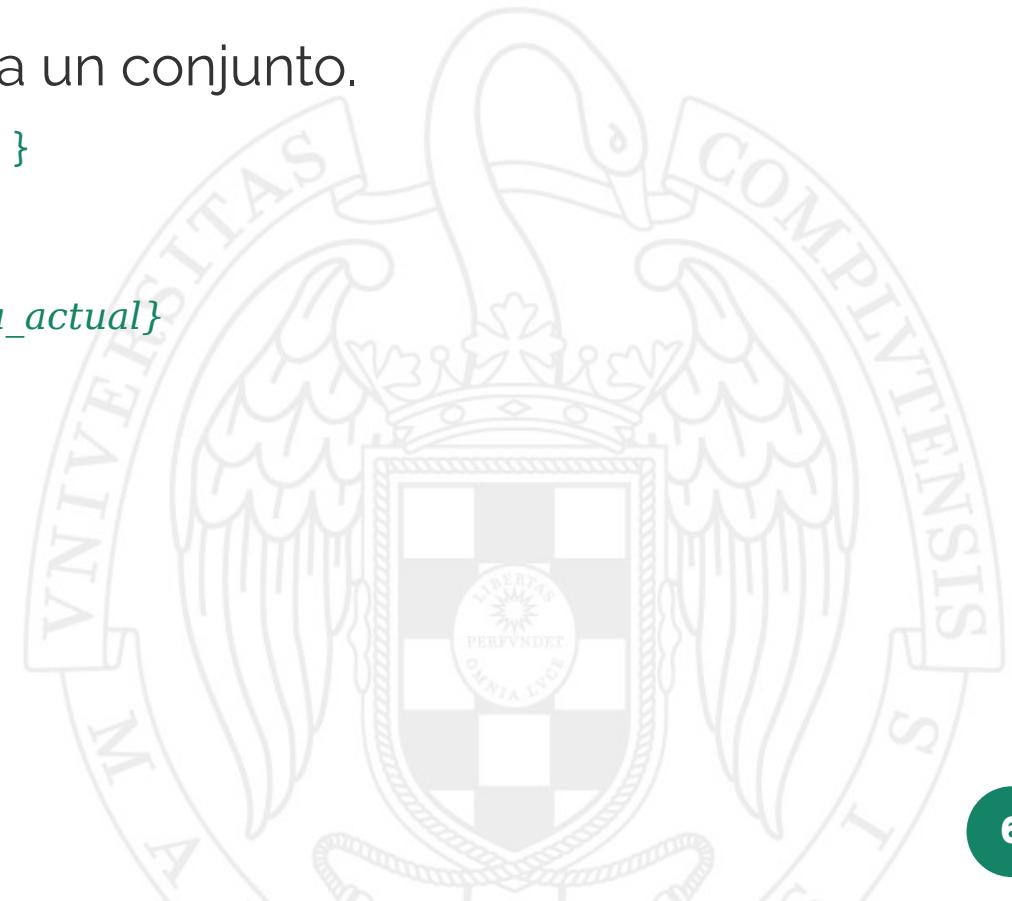
*LetrasNombradas =  $\emptyset$ ;*

- Saber si una letra pertenece (o no) a un conjunto.

*while (letra\_actual  $\notin$  LetrasNombradas) { ... }*

- Añadir una letra a un conjunto.

*LetrasNombradas = LetrasNombradas  $\cup$  {letra\_actual}*



# Tipo Abstracto de Datos: definición

- Un **tipo abstracto de datos** (TAD) es un tipo de datos asociado con:
  - Un **modelo** conceptual.
  - Un conjunto de **operaciones**, especificadas mediante ese modelo.



# En nuestro ejemplo

Tipo de datos: ConjuntoChar

- **Modelo:** conjuntos de letras, en el sentido matemático del término.

- **Operaciones:**  $[ \text{true} ]$

**vacio()**  $\rightarrow (C: \text{ConjuntoChar})$

$[ C = \emptyset ]$

$[ l \in \{A, \dots, Z\} ]$

**pertenece**(l: char, C: ConjuntoChar)  $\rightarrow (\text{está}: \text{bool})$

$[ \text{está} \Leftrightarrow l \in C ]$

$[ l \in \{A, \dots, Z\} ]$

**añadir**(l: char, C: ConjuntoChar)

$[ C = \text{old}(C) \cup \{l\} ]$

# Implementación del TAD

- Nuestro modelo conceptual admite varias representaciones en C++. Hemos propuesto dos:
- **Representación 1:** array de caracteres.

```
struct ConjuntoChar {  
    int num_chars;  
    char elementos[MAX_CHARS];  
};
```

- **Representación 2:** array de booleanos.

```
struct ConjuntoChar {  
    bool esta[MAX_CHARS];  
};
```

Cada representación implementa de manera distinta las operaciones mostradas anteriormente

La representación determina la eficiencia de las operaciones implementadas

# Representación 1

```
void vacio(ConjuntoChar &result) {  
    result.num_chars = 0;  
}  
  
void anyadir(char letra, ConjuntoChar &conjunto) {  
    assert (conjunto.num_chars < MAX_CHARS);  
    assert (letra ≥ 'A' && letra ≤ 'Z');  
    conjunto.elementos[conjunto.num_chars] = letra;  
    conjunto.num_chars++;  
}  
  
bool pertenece(char letra, const ConjuntoChar &conjunto) {  
    assert (letra ≥ 'A' && letra ≤ 'Z');  
    int i = 0;  
    while (i < conjunto.num_chars && conjunto.elementos[i] ≠ letra) {  
        i++;  
    }  
    return conjunto.elementos[i] = letra;  
}
```

# Representación 2

```
void vacio(ConjuntoChar &resultado) {
    for (int i = 0; i < MAX_CHARS; i++) {
        resultado.está[i] = false;
    }
}

void añadir(char letra, ConjuntoChar &conjunto) {
    assert (letra ≥ 'A' && letra ≤ 'Z');
    conjunto.está[letra - 'A'] = true;
}

bool pertenece(char c, const ConjuntoChar &conjunto) {
    assert (c ≥ 'A' && c ≤ 'Z');
    return conjunto.está[c - 'A'];
}
```



# Nuestro programa ideal...

```
int main() {
    int jugador_actual = 1;
    LetrasNombradas = ∅;

    char letra_actual = preguntar_letra(jugador_actual);

    while (letra_actual ∈ LetrasNombradas) {
        LetrasNombradas = LetrasNombradas ∪ {letra_actual}

        jugador_actual = cambio_jugador(jugador_actual);
        letra_actual = preguntar_letra(jugador_actual);
    }

    std::cout << "Jugador " << jugador_actual << " ha perdido!" << std::endl;
    std::cout << "La letra repetida ha sido: " << letra_actual << std::endl;
    return 0;
}
```

# ... y nuestro programa real

```
int main() {
    int jugador_actual = 1;
    ConjuntoChar letras_nombradas;
    vacio(letras_nombradas);

    char letra_actual = preguntar_letra(jugador_actual);

    while (!pertenece(letra_actual, letras_nombradas)) {
        anyadir(letra_actual, letras_nombradas);

        jugador_actual = cambio_jugador(jugador_actual);
        letra_actual = preguntar_letra(jugador_actual);
    }

    std::cout << "Jugador " << jugador_actual << " ha perdido!" << std::endl;
    std::cout << "La letra repetida ha sido: " << letra_actual << std::endl;
    return 0;
}
```

# ¿Qué hemos ganado?

## 1) Simplificar el desarrollo

No hemos de preocuparnos de cómo está implementado `ConjuntoChar`.

## 2) Reutilización

`ConjuntoChar` puede utilizarse en otros contextos.

## 3) Separación de responsabilidades

Podemos reemplazar una implementación de `ConjuntoChar` por otra sin alterar el resto del programa.

# Pero hay personas despistadas

```
int main() {  
    ...  
  
    std::cout << "Jugador " << jugador_actual << " ha perdido!" << std::endl;  
    std::cout << "La letra repetida ha sido: " << letra_actual << std::endl;  
    std::cout << "Se han nombrado " << letras_nombradas.num_chars  
        << " letras." << std::endl;  
    return 0;  
}
```



¿Existe algún mecanismo en el compilador de C++ que impida a las personas despistadas acceder a la representación interna de un TAD?

## Encapsulación mediante clases

ESTRUCTURAS DE DATOS

INTRODUCCIÓN A LOS TIPOS ABSTRACTOS DE DATOS

# TADs: definición

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

ESTRUCTURAS DE DATOS

INTRODUCCIÓN A LOS TIPOS ABSTRACTOS DE DATOS

# Encapsulación en TADs

Manuel Montenegro Montes

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

# TAD ConjuntoChar

- Representa conjuntos de caracteres en mayúsculas en el alfabeto inglés (A..Z).

- Operaciones:

[ *true* ]

**vacio()** → (C: ConjuntoChar)

[  $C = \emptyset$  ]

[  $l \in \{A, \dots, Z\}$  ]

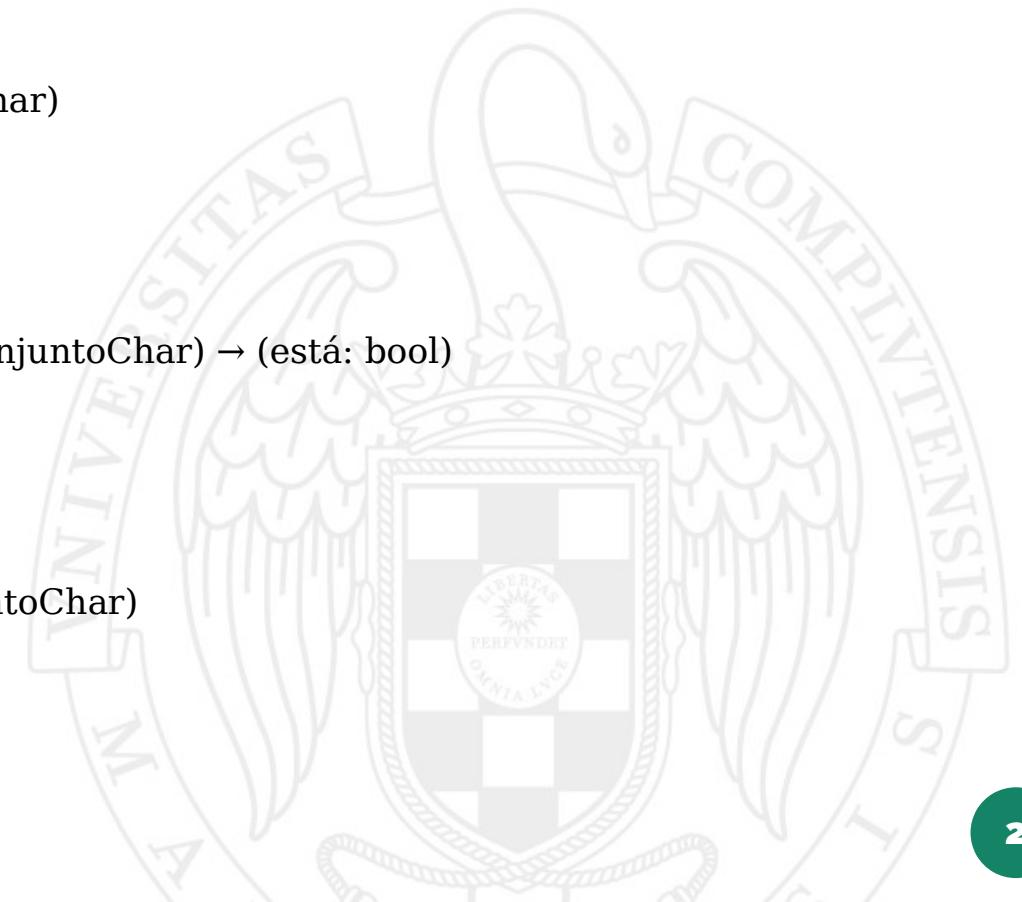
**pertenece**(l: char, C: ConjuntoChar) → (está: bool)

[  $\text{está} \Leftrightarrow l \in C$  ]

[  $l \in \{A, \dots, Z\}$  ]

**añadir**(l: char, C: ConjuntoChar)

[  $C = \text{old}(C) \cup \{l\}$  ]



# Implementación de TADs mediante clases

# TADs mediante clases

- Podemos definir tipos abstracto de datos utilizando las clases de C++.

```
class ConjuntoChar {  
public:  
  
    // operaciones públicas  
  
private:  
    // representación interna  
};
```



# TADs mediante clases

- Podemos definir tipos abstracto de datos utilizando las clases de C++.

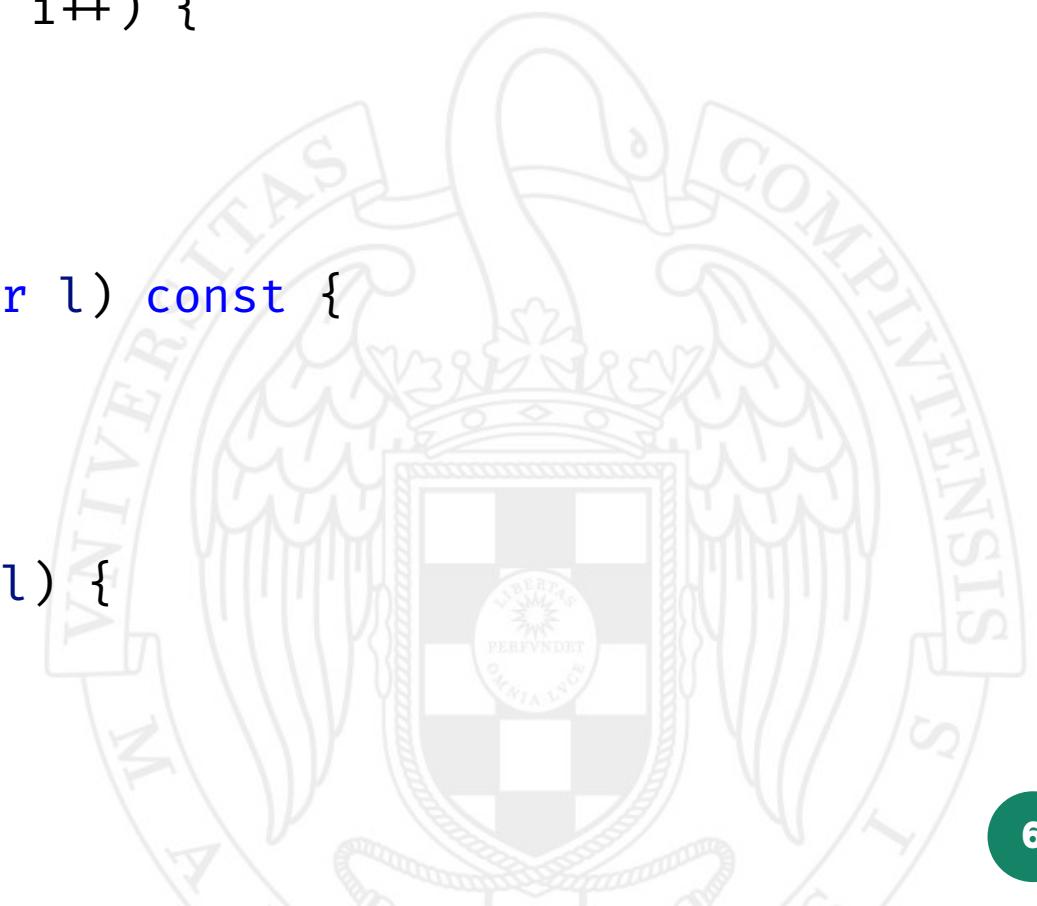
```
class ConjuntoChar {  
public:  
    ConjuntoChar(); → vacio() → (C: ConjuntoChar)  
    bool pertenece(char l) const; → pertenece(l: char, C: ConjuntoChar) → bool  
    void anyadir(char l); → añadir(l: char, C: ConjuntoChar)  
  
private:  
    bool esta[MAX_CHARS];  
};
```

# Implementación de las operaciones

```
ConjuntoChar::ConjuntoChar() {
    for (int i = 0; i < MAX_CHARS; i++) {
        esta[i] = false;
    }
}

bool ConjuntoChar::pertenece(char l) const {
    assert (l ≥ 'A' && l ≤ 'Z');
    return esta[l - (int)'A'];
}

void ConjuntoChar::anyadir(char l) {
    assert (l ≥ 'A' && l ≤ 'Z');
    esta[l - (int)'A'] = true;
}
```



# ¿Cómo comprobar las precondiciones?

[ true ]

**vacio()** → (C: ConjuntoChar)

[  $C = \emptyset$  ]

[  $l \in \{A, \dots, Z\}$  ]

**pertenecé(l: char, C: ConjuntoChar)** → (está: bool)

[  $\text{está} \Leftrightarrow l \in C$  ]

[  $l \in \{A, \dots, Z\}$  ]

**añadir(l: char, C: ConjuntoChar)**

[  $C = \text{old}(C) \cup \{l\}$  ]

- Más sencillo, pero menos flexible: la macro **assert**.
  - Se encuentra en el fichero de cabecera **<cassert>**.
  - Comprueba la condición pasada como parámetro. Si es falsa, el programa aborta.
- Más potente y flexible: manejo de excepciones en C++.

# Uso de TAD: lenguaje ideal

```
int main() {
    int jugador_actual = 1;
    LetrasNombradas = ∅;

    char letra_actual = preguntar_letra(jugador_actual);

    while (letra_actual ∈ LetrasNombradas) {
        LetrasNombradas = LetrasNombradas ∪ {letra_actual}

        jugador_actual = cambio_jugador(jugador_actual);
        letra_actual = preguntar_letra(jugador_actual);
    }

    std::cout << "Jugador " << jugador_actual << " ha perdido!" << std::endl;
    std::cout << "La letra repetida ha sido: " << letra_actual << std::endl;
    return 0;
}
```

# Uso de TAD: sin clases

```
int main() {
    int jugador_actual = 1;
    ConjuntoChar letras_nombradas = vacio();
    vacio(letras_nombradas);

    char letra_actual = preguntar_letra(jugador_actual);

    while (!pertenece(letra_actual, letras_nombradas)) {
        anyadir(letra_actual, letras_nombradas);

        jugador_actual = cambio_jugador(jugador_actual);
        letra_actual = preguntar_letra(jugador_actual);
    }

    std::cout << "Jugador " << jugador_actual << " ha perdido!" << std::endl;
    std::cout << "La letra repetida ha sido: " << letra_actual << std::endl;
    return 0;
}
```

# Uso de TAD: con clases

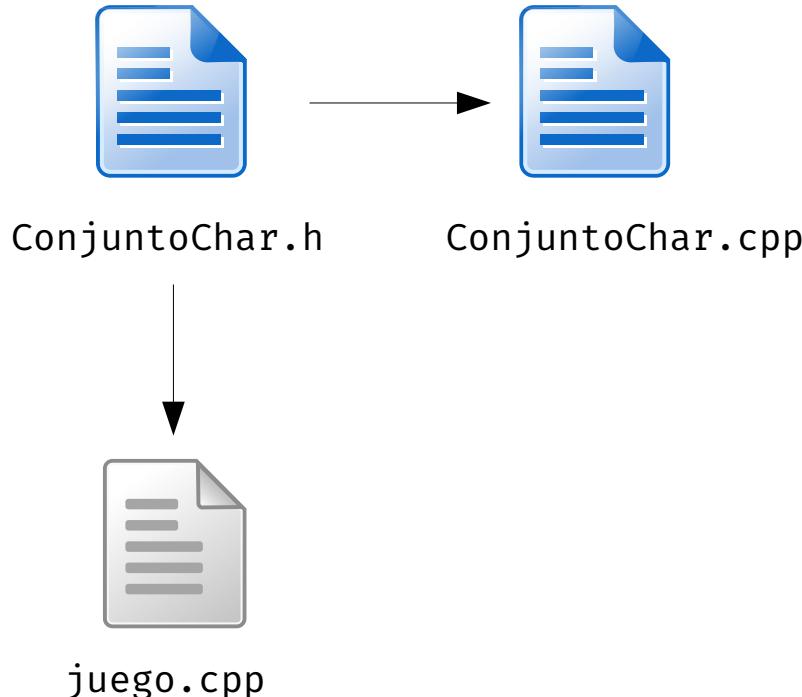
```
int main() {  
    int jugador_actual = 1;  
    ConjuntoChar letras_nombradas;  
  
    char letra_actual = preguntar_letra(jugador_actual);  
  
    while (!letras_nombradas.pertenece(letra_actual)) {  
        letras_nombradas.anyadir(letra_actual);  
  
        jugador_actual = cambio_jugador(jugador_actual);  
        letra_actual = preguntar_letra(jugador_actual);  
    }  
  
    std::cout << "Jugador " << jugador_actual << " ha perdido!" << std::endl;  
    std::cout << "La letra repetida ha sido: " << letra_actual << std::endl;  
    return 0;  
}
```



Encapsulación garantizada  
por el compilador

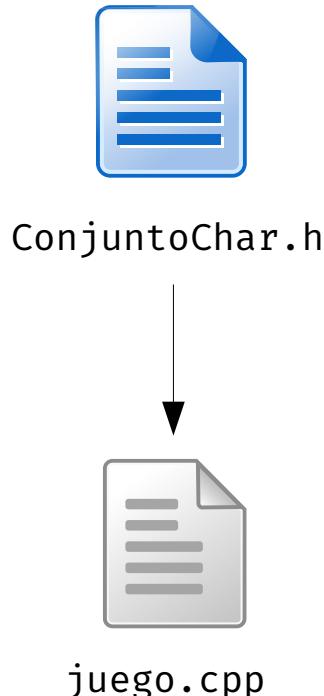
# Modularidad: ¿Cómo organizar el código?

# Alt. 1: interfaz/implementación separadas



- Ventajas:
  - Separación de aspectos de implementación.
  - La implementación puede compilarse por separado.
- Desventajas:
  - No puede utilizarse en combinación con plantillas de C++ (`template`).

# Alt. 2: interfaz e implementación juntas



- Inconvenientes:
  - Los detalles de implementación quedan expuestos.
  - Si cambiamos `ConjuntoChar`, hemos de recompilar `juego.cpp`
- Pero cuando utilicemos templates no tendremos más remedio que implementar las operaciones genéricas en el .h
  - ... por lo menos hasta C++20*

ESTRUCTURAS DE DATOS

INTRODUCCIÓN A LOS TIPOS ABSTRACTOS DE DATOS

# Modelo vs. representación

Manuel Montenegro Montes

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

# Modelo vs. representación

# TAD ConjuntoChar (representación 1)

## Modelo

Conjuntos de letras mayúsculas

$$\mathcal{P}(\{A..Z\})$$

$$\{A, D, Z\}$$

$$\{G, M\}$$

$$\emptyset$$

## Representación

```
class ConjuntoChar {  
    ...  
private:  
    int num_chars;  
    char elementos[MAX_CHARS];  
};
```

num\_chars: 3  
elementos: A|D|Z | | | ... | | |

num\_chars: 2  
elementos: G|M | | | ... | | |

num\_chars: 0  
elementos: | | | | | ... | | |

# TAD ConjuntoChar (representación 2)

## Modelo

Conjuntos de letras mayúsculas

$$\mathcal{P}(\{A..Z\})$$

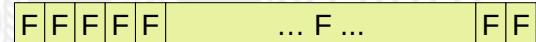
$$\{A, D, Z\}$$

$\emptyset$

## Representación

```
class ConjuntoChar {  
    ...  
private:  
    bool esta[MAX_CHARS];  
};
```

esta: 

esta: 

# TAD de números int

## Modelo

Elementos de  $\mathbb{Z}$

25

-7

## Representación

32 bits en complemento a 2

000...00011001

111...11111001

# TAD de números float

Modelo	Representación
Elementos de $\mathbb{Q}$	IEEE 754
$1.3423121$	<code>001111110101011101000011100010</code>
$-0.5$	<code>101111100000000000000000000000000000000</code>

# ¡Cuidado! La representación es relevante

¿Por qué nos interesa conocer la representación?

- **Eficiencia de las operaciones**
  - El coste en tiempo puede depender de la representación.
- **Coste en memoria de la representación**
  - Algunas representaciones necesitan más memoria.
- **Limitaciones de algunas representaciones**

# Limitaciones de algunas representaciones

- Enteros de 32 bits: -2147483648 a 2147483647.
- Coma flotante con float:

0.7  
0011111001100110011001100110011  
0.6999999881

```
float f = 7.0 / 10;  
std::cout << std::setprecision(10) << f << std::endl;
```

# Limitaciones de algunas representaciones

- Enteros de 32 bits: -2147483648 a 2147483647.
- Coma flotante con float:

0.7  
0011111001100110011001100110011  
0.6999999881

- TAD ConjuntoChar: capacidad máxima

# Función de abstracción

# Función de abstracción

- Fijada la representación de un TAD, la función de abstracción asociada a una representación que asocia cada instancia de la representación con el modelo que representa.
- Ejemplo: TAD `int`

$$000\dots00011001 \xrightarrow{f_{int}} 25$$

$$f_{int}(x_{31}x_{30}\dots x_0) = \begin{cases} \sum_{i=0}^{30} 2^i * x_i & \text{si } x_{31} = 0 \\ -(1 + \sum_{i=0}^{30} 2^i * \overline{x_i}) & \text{si } x_{31} = 1 \end{cases}$$

# Función de abstracción

- Fijada la representación de un TAD, la función de abstracción asociada a una representación que asocia cada instancia de la representación con el modelo que representa.
- Ejemplo: TAD ConjuntoChar

```
class ConjuntoChar {  
    ...  
private:  
    int num_chars;  
    char elementos[MAX_CHARS];  
};
```

$$f_{CCI} : \text{ConjuntoChar} \rightarrow \mathcal{P}(\{A..Z\})$$

$$f_{CCI}(x) = \{ x.elementos[i] \mid 0 \leq i < x.num\_chars \}$$

# Función de abstracción

- Fijada la representación de un TAD, la función de abstracción asociada a una representación que asocia cada instancia de la representación con el modelo que representa.
- Ejemplo: TAD ConjuntoChar

```
class ConjuntoChar {  
    ...  
private:  
    bool esta[MAX_CHARS];  
};
```

$$f_{CC2} : \text{ConjuntoChar} \rightarrow \mathcal{P}(\{A..Z\})$$

$$f_{CC2}(x) = \{ c \in \{A..Z\} \mid x.esta[\text{ord}(c) - \text{ord}('A')] = \text{true} \}$$

# Tipos de operaciones

# TAD = Modelo + Operaciones

- Las operaciones en un TAD se especifican en función de los modelos.

[ true ]

**vacio()** → (C: ConjuntoChar)

[  $C = \emptyset$  ]

[  $l \in \{A, \dots, Z\}$  ]

**pertenecce**(l: char, C: ConjuntoChar) → (está: bool)

[  $\text{está} \Leftrightarrow l \in C$  ]

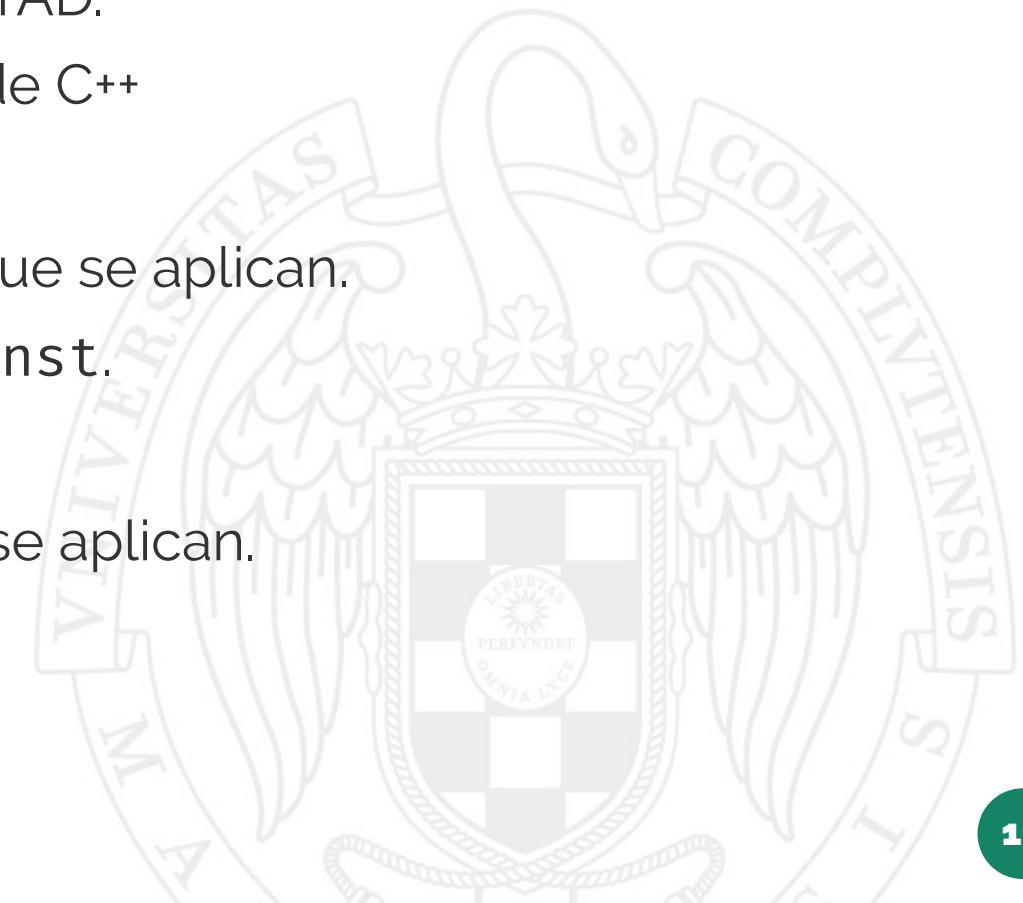
[  $l \in \{A, \dots, Z\}$  ]

**añadir**(l: char, C: ConjuntoChar)

[  $C = \text{old}(C) \cup \{l\}$  ]

# Tipos de operaciones

- Funciones **constructoras**
  - Crean una nueva instancia del TAD.
  - Equivalen a los constructores de C++
- Funciones **observadoras**
  - No modifican el TAD sobre el que se aplican.
  - En C++ llevan el modificador `const`.
- Funciones **mutadoras**
  - Modifican el TAD sobre el que se aplican.



# Ejemplo

[ true ]

**vacio()** → (C: ConjuntoChar)

[  $C = \emptyset$  ]

[  $l \in \{A, \dots, Z\}$  ]

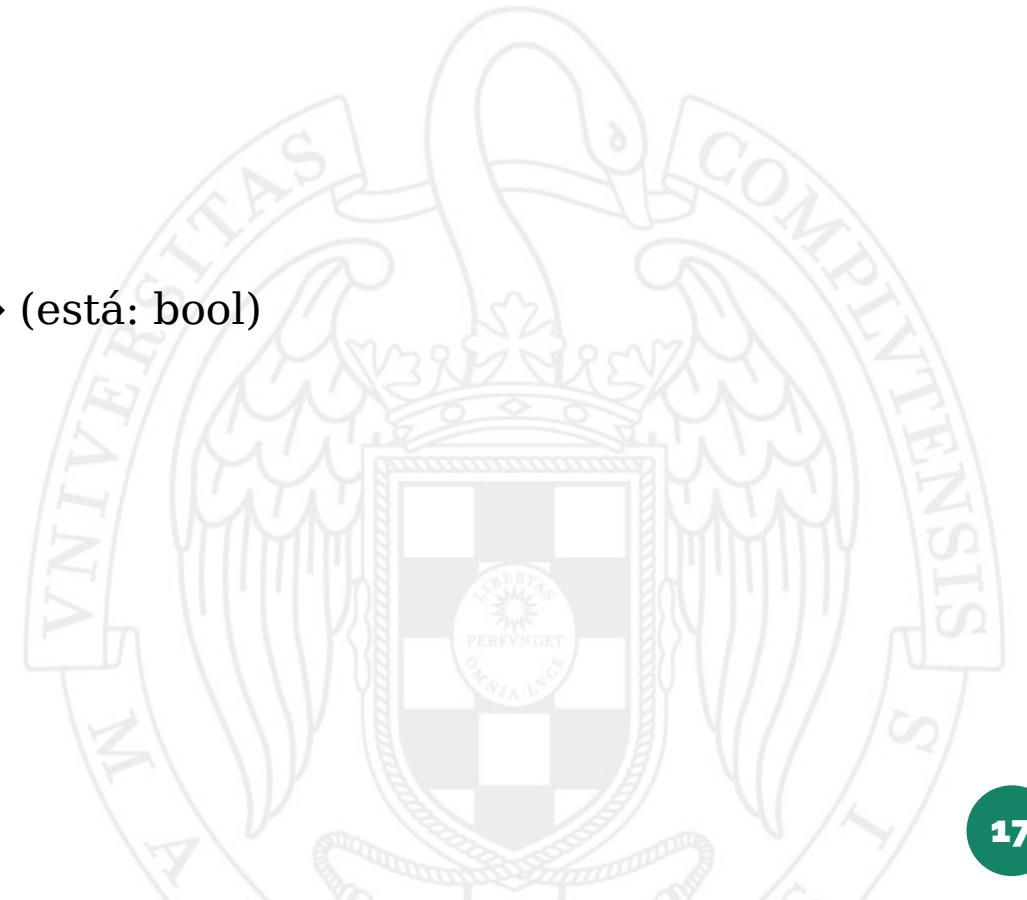
**pertenecce(l: char, C: ConjuntoChar)** → (está: bool)

[  $está \Leftrightarrow l \in C$  ]

[  $l \in \{A, \dots, Z\}$  ]

**añadir(l: char, C: ConjuntoChar)**

[  $C = old(C) \cup \{l\}$  ]



# Invariante de la representación

# Instancias no válidas

- No todas las instancias de una representación denotan un modelo.

```
class ConjuntoChar {  
    ...  
private:  
    int num_chars;  
    char elementos[MAX_CHARS];  
};
```

num\_chars: 2

elementos: 9 M | | | ... | | |

num\_chars: -3

elementos: B M | | | ... | | |

$f_{CCI}$

???

$f_{CCI}$

???

# Invariante de representación

- Un **invariante de representación** es una fórmula lógica que especifica cuándo una instancia es válida.

```
class ConjuntoChar {  
    ...  
private:  
    int num_chars;  
    char elementos[MAX_CHARS];  
};
```

$$I_{CCI}(x) =$$

$$0 \leq x.num\_chars \leq MAX\_CHARS \wedge$$

$$\forall i: 0 \leq i < x.num\_chars \Rightarrow x.elementos[i] \in \{A..Z\}$$

# Invariante de representación

- Un **invariante de representación** (o invariante de clase) es una fórmula lógica que especifica cuándo una instancia es válida.

```
class ConjuntoChar {  
    ...  
    private:  
        bool esta[MAX_CHARS];  
};
```

$$I_{CC2}(x) = \text{true}$$

# Invariantes y operaciones

- Las operaciones constructoras deben producir una instancia que cumpla el invariante.
- Las operaciones consultoras pueden asumir que la instancia cumple el invariante.
- Las operaciones mutadoras pueden asumir que la instancia cumple el invariante, y han de preservarlo al final de su ejecución.

[ *true* ]

**vacio()** → (C: ConjuntoChar)

[  $C = \emptyset$  ]

[  $l \in \{A, \dots, Z\}$  ]

**pertenece(l: char, C: ConjuntoChar)** → (está: bool)

[  $está \Leftrightarrow l \in C$  ]

[  $l \in \{A, \dots, Z\}$  ]

**añadir(l: char, C: ConjuntoChar)**

[  $C = old(C) \cup \{l\}$  ]

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# EL TAD Lista

Manuel Montenegro Montes

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

# El TAD Lista

- Una **lista** es una colección de elementos, con las siguientes características:
  - Tiene longitud finita.
  - Los elementos siguen un orden secuencial:
    - Existe un primer elemento y un último elemento.
    - Todos los elementos en la lista tienen un predecesor (excepto el primero) y un sucesor (excepto el último).
  - Se permiten elementos duplicados en la lista.

# Modelo del TAD Lista

- Las listas representan **secuencias** de elementos de la siguiente forma:

$$[ x_0, x_1, x_2, \dots, x_{n-1} ]$$

## Primer elemento

Longitud de la lista: *n*

## Último elemento

- Ejemplos:

[ 3, 1, 6, 14, 5 ]

[ 1, 3, 5, 6, 14 ]

[ 25, 25, 7, 5, 7, 7 ]

[ Marta, David, Gerardo ]

[ true ]

$$[ [1, 0, 0], [0, 1, 0], [0] ]$$

[ ]

## Lista vacía (longitud 0)

# Operaciones sobre el TAD Lista

- **Constructoras:**
  - Crear una lista vacía (*create\_empty*).
- **Mutadoras:**
  - Añadir un elemento al principio de la lista (*push\_front*).
  - Añadir un elemento al final de la lista (*push\_back*).
  - Eliminar el elemento del principio de la lista (*pop\_front*).
  - Eliminar el elemento del final de la lista (*pop\_back*).
- **Observadoras:**
  - Obtener el tamaño de la lista (*size*).
  - Comprobar si la lista es vacía (*empty*).
  - Acceder al primer elemento de la lista (*front*).
  - Acceder al último elemento de la lista (*back*).
  - Acceder a un elemento que ocupa una posición determinada (*at*).

# Operaciones constructoras

{ true }

***create\_empty()*** → (L: List)

{ L = [] }



# Operaciones mutadoras

{  $L = [x_0, \dots, x_{n-1}]$  }

**push\_front**( $x$ : elem,  $L$ : List)

{  $L = [x, x_0, \dots, x_{n-1}]$  }

{  $L = [x_0, x_1, \dots, x_{n-1}], n \geq 1$  }

**pop\_front**( $L$ : List)

{  $L = [x_1, \dots, x_{n-1}]$  }

{  $L = [x_0, \dots, x_{n-1}]$  }

**push\_back**( $x$ : elem,  $L$ : List)

{  $L = [x_0, \dots, x_{n-1}, x]$  }

{  $L = [x_0, \dots, x_{n-2}, x_{n-1}], n \geq 1$  }

**pop\_back**( $L$ : List)

{  $L = [x_0, \dots, x_{n-2}]$  }

# Operaciones observadoras

$\{ L = [x_0, \dots, x_{n-1}] \}$

**size**( $L: List$ )  $\rightarrow$  ( $tamaño: int$ )  
 $\{ tamaño = n \}$

$\{ L = [x_0, \dots, x_{n-1}] \wedge n \geq 1 \}$

**front**( $L: List$ )  $\rightarrow$  ( $e: elem$ )  
 $\{ e = x_0 \}$

$\{ L = [x_0, \dots, x_{n-1}] \wedge 0 \leq i < n \}$

**at**( $i: int, L: List$ )  $\rightarrow$  ( $e: elem$ )  
 $\{ e = x_i \}$

$\{ L = [x_0, \dots, x_{n-1}] \}$

**empty**( $L: List$ )  $\rightarrow$  ( $b: bool$ )  
 $\{ b \Leftrightarrow n = 0 \}$

$\{ L = [x_0, \dots, x_{n-1}] \wedge n \geq 1 \}$

**back**( $L: List$ )  $\rightarrow$  ( $e: elem$ )  
 $\{ e = x_{n-1} \}$

# Operaciones adicionales

- Algunas implementaciones permiten las siguientes operaciones:
  - Mostrar una lista por pantalla.
  - Insertar/eliminar en una posición determinada.
  - Concatenar dos listas.
  - Invertir el orden de los elementos de una lista.
  - Recorrer una lista.

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Implementación del TAD Lista mediante arrays

Manuel Montenegro Montes

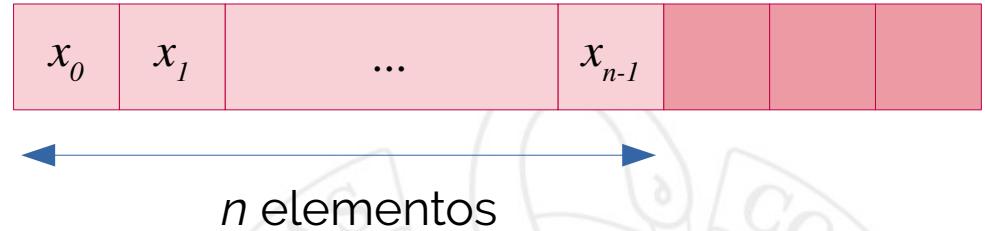
Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Recordatorio: operaciones del TAD Lista

- **Constructoras:**
  - Crear una lista vacía: ***create\_empty()*** →  $L: List$
- **Mutadoras:**
  - Añadir un elemento al principio de la lista: ***push\_front(x: elem, L: List)***.
  - Añadir un elemento al final de la lista: ***push\_back(x: elem, L: List)***.
  - Eliminar el elemento del principio de la lista: ***pop\_front(L: List)***.
  - Eliminar el elemento del final de la lista: ***pop\_back(L: List)***.
- **Observadoras:**
  - Obtener el tamaño de la lista: ***size(L: List)*** →  $tam: int$ .
  - Comprobar si la lista es vacía ***empty(L: List)*** →  $b: bool$ .
  - Acceder al primer elemento de la lista ***front(L: List)*** →  $e: elem$ .
  - Acceder al último elemento de la lista ***back(L: List)*** →  $e: elem$ .
  - Acceder a un elemento que ocupa una posición determinada ***at(idx: int, L: List)*** →  $e: elem$ .

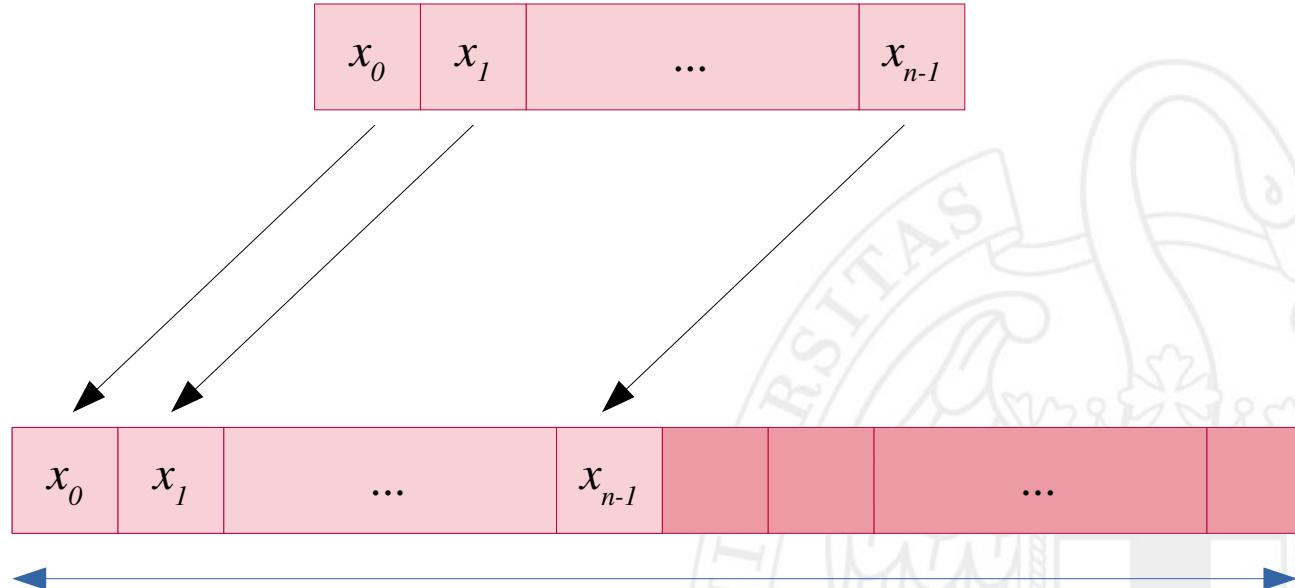
# Implementación mediante arrays

$[x_0, x_1, \dots, x_{n-1}]$



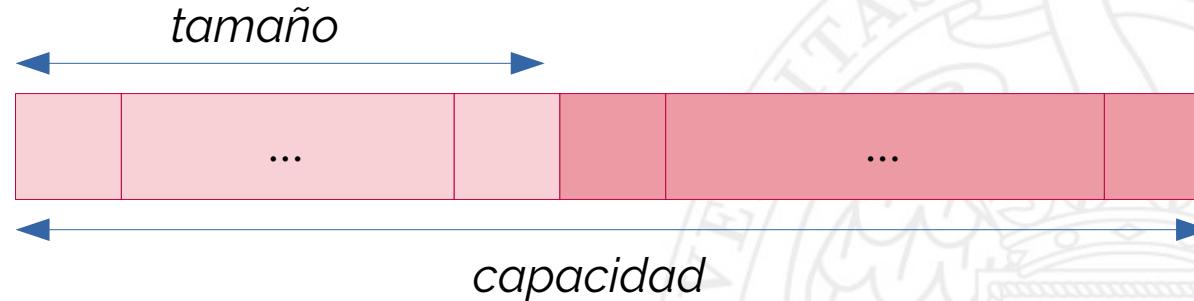
```
class ListArray {  
public:  
    ...  
private:  
    int num_elems;  
    std::string elems[MAX_CAPACITY];  
};
```

# ¿Y si el array se llena?



# Tamaño vs. capacidad

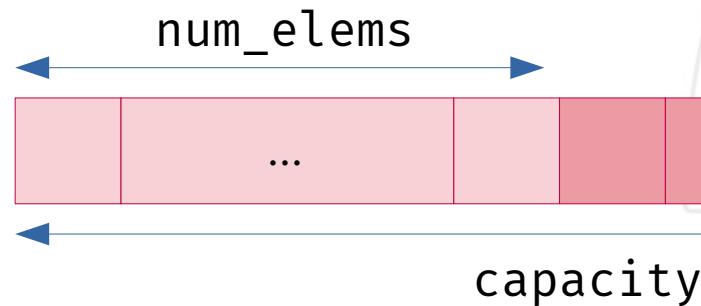
- **Capacidad de una lista:** Tamaño del array que contiene los elementos.
- **Tamaño de una lista:** Número de posiciones ocupadas por elementos.
  - Siempre se cumple  $\text{tamaño} \leq \text{capacidad}$ .



# Implementación con tamaño y capacidad

```
class ListArray {  
public:  
    ...  
private:  
    int num_elems;  
    int capacity;  
    std::string *elems;  
};
```

Array reservado dinámicamente



# Invariante y modelo de representación

```
class ListArray {  
public:  
    ...  
private:  
    int num_elems;  
    int capacity;  
    std::string *elems;  
};
```

- Invariante de la representación:

$$I(x) = 0 \leq x.\text{num\_elems} \leq x.\text{capacity}$$

- Función de abstracción:

$$f(x) = [x.\text{elems}[0], x.\text{elems}[1], \dots, x.\text{elems}[x.\text{num\_elems} - 1]]$$



# Creación y destrucción de una lista

```
const int DEFAULT_CAPACITY = 10;

class ListArray {

public:
    ListArray(int initial_capacity)
        : num_elems(0), capacity(initial_capacity), elems(new std::string[capacity]) { }

    ListArray()
        : ListArray(DEFAULT_CAPACITY) { }

    ~ListArray() { delete[] elems; }

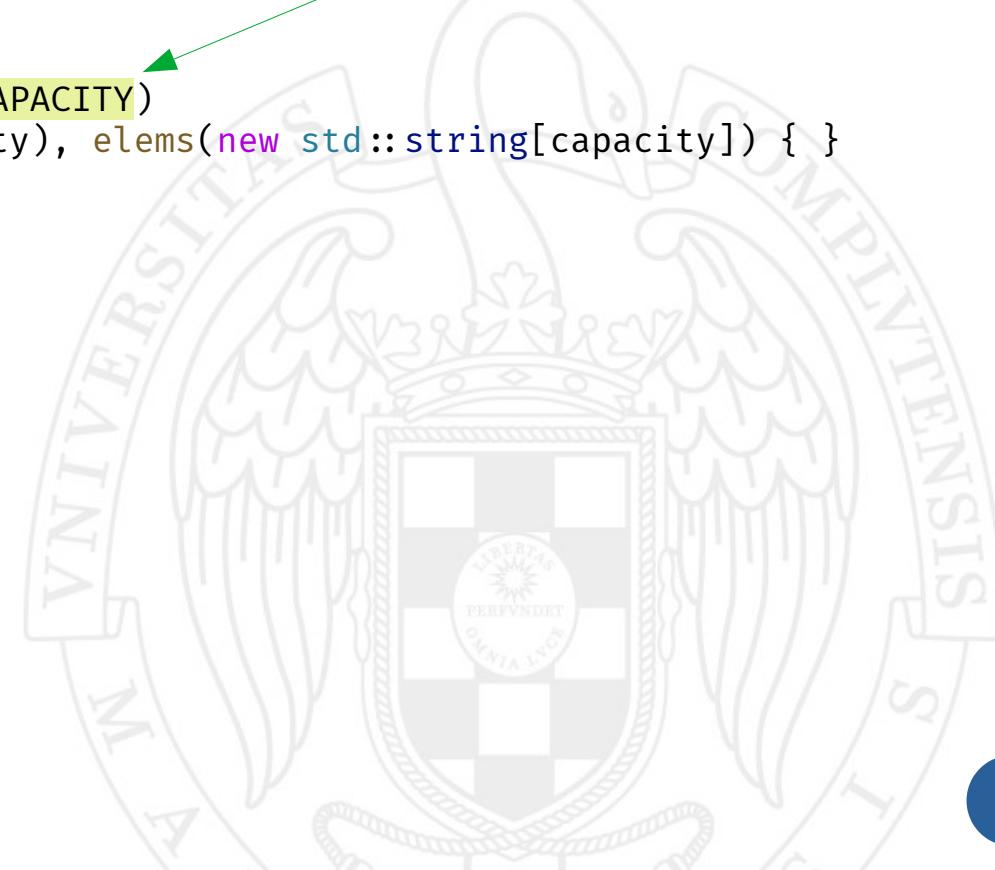
    ...

private:
    int num_elems;
    int capacity;
    std::string *elems;
};
```

# Creación y destrucción de una lista

```
const int DEFAULT_CAPACITY = 10;  
  
class ListArray {  
  
public:  
    ListArray(int initial_capacity = DEFAULT_CAPACITY)  
        : num_elems(0), capacity(initial_capacity), elems(new std::string[capacity]) {}  
  
    ListArray()  
        : ListArray(DEFAULT_CAPACITY) {}  
  
    ~ListArray() { delete[] elems; }  
  
    ...  
  
private:  
    int num_elems;  
    int capacity;  
    std::string *elems;  
};
```

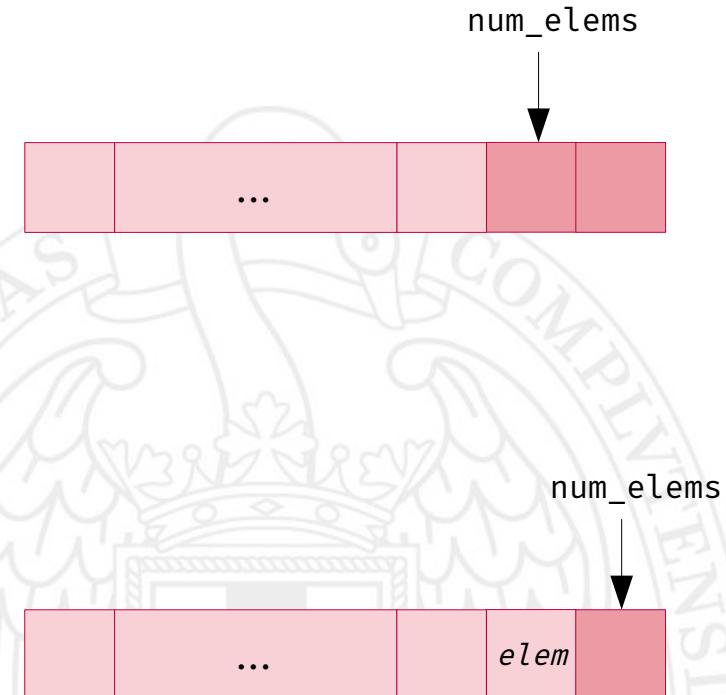
Valores por defecto para parámetros



# Añadir elementos al final de una lista

```
class ListArray {  
public:  
    void push_back(const std::string &elem);  
    ...  
private:  
    int num_elems;  
    int capacity;  
    std::string *elems;  
};
```

```
void ListArray::push_back(const std::string &elem) {  
    if (num_elems == capacity) {  
        resize_array(capacity * 2);  
    }  
  
    elems[num_elems] = elem;  
    num_elems++;  
}
```



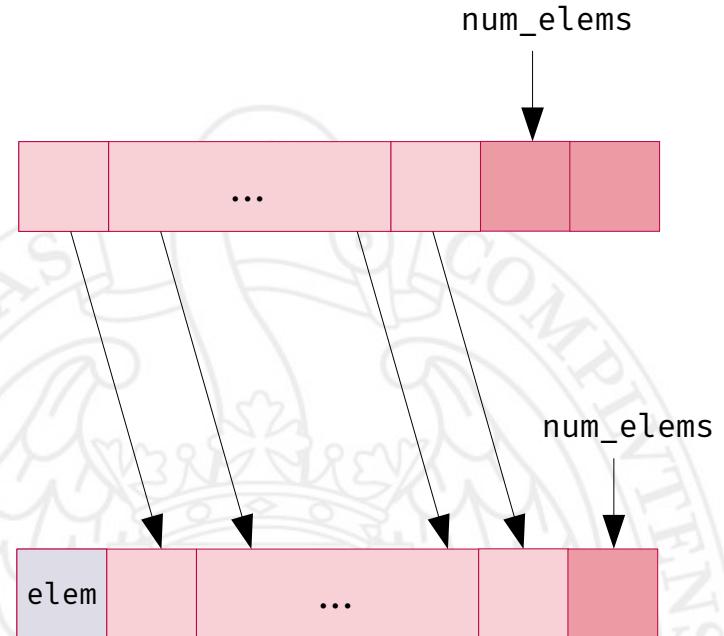
# Redimensionar el array

```
class ListArray {  
public:  
    ...  
private:  
    int num_elems;  
    int capacity;  
    std::string *elems;  
  
    void resize_array(int new_capacity);  
};  
  
void ListArray::resize_array(int new_capacity) {  
    std::string *new_elems = new std::string[new_capacity];  
    for (int i = 0; i < num_elems; i++) {  
        new_elems[i] = elems[i];  
    }  
  
    delete[] elems;  
    elems = new_elems;  
    capacity = new_capacity;  
}
```

# Añadir elementos al principio de una lista

```
class ListArray {  
public:  
    void push_front(const std::string &elem);  
    ...  
};
```

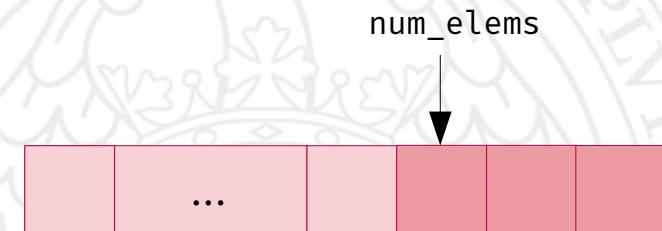
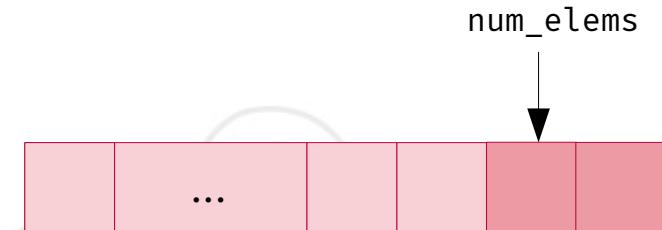
```
void ListArray::push_front(const std::string &elem) {  
    if (num_elems == capacity) {  
        resize_array(capacity * 2);  
    }  
  
    for (int i = num_elems - 1; i ≥ 0; i--) {  
        elems[i + 1] = elems[i];  
    }  
    elems[0] = elem;  
    num_elems++;  
}
```



# Eliminar elementos del final de una lista

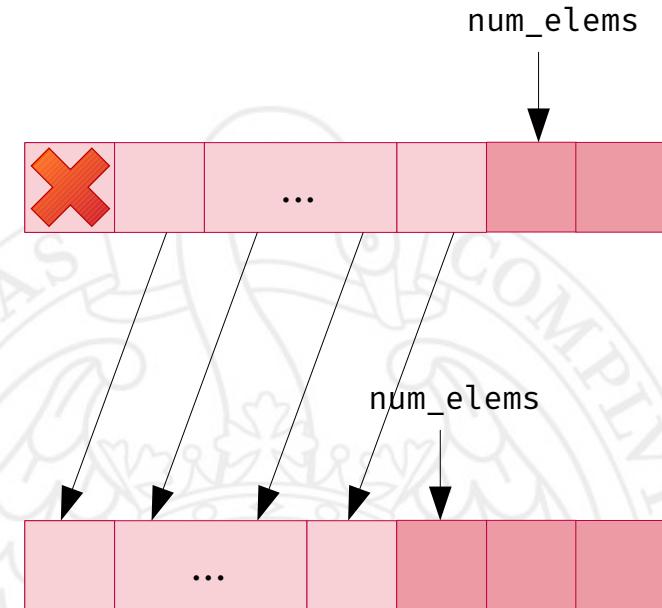
```
class ListArray {  
public:  
    void pop_back();  
    ...  
};
```

```
void ListArray::pop_back() {  
    assert (num_elems > 0);  
    num_elems--;  
}
```



# Eliminar elementos del principio de una lista

```
class ListArray {  
public:  
    void pop_front();  
    ...  
};  
  
void ListArray::pop_front() {  
    assert (num_elems > 0);  
  
    for (int i = 1; i < num_elems; i++) {  
        elems[i - 1] = elems[i];  
    }  
    num_elems--;  
}
```



# Funciones observadoras (1)

```
class ListArray {  
public:  
  
    int size() const {  
        return num_elems;  
    }  
  
    bool empty() const {  
        return num_elems == 0;  
    }  
  
    std::string front() const {  
        assert (num_elems > 0);  
        return elems[0];  
    }  
    ...  
};
```



# Funciones observadoras (2)

```
class ListArray {  
public:  
  
    std::string back() const {  
        assert (num_elems > 0);  
        return elems[num_elems - 1];  
    }  
  
    std::string at(int index) const {  
        assert (0 <= index && index < num_elems);  
        return elems[index];  
    }  
    ...  
};
```



# Mostrar el contenido de una lista

```
class ListArray {
public:
    void display() const;
    ...
};

void ListArray::display() const {
    std::cout << "[";
    if (num_elems > 0) {
        std::cout << elems[0];
        for (int i = 1; i < num_elems; i++) {
            std::cout << ", " << elems[i];
        }
    }
    std::cout << "]";
}
```



# Prueba de ejecución

```
int main() {
    ListArray l;
    l.push_back("David");
    l.push_back("Maria");
    l.push_back("Elvira");
    l.display(); std::cout << std::endl;

    std::cout << "Elemento 1: " << l.at(1) << std::endl;

    l.pop_front();
    l.display(); std::cout << std::endl;

    return 0;
}
```

[David, Maria, Elvira]

Maria

[Maria, Elvira]

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Modificación de listas mediante referencias

Manuel Montenegro Montes

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

# Comportamiento en memoria de at()

```
class ListArray {
public:

    std::string at(int index) const {
        assert (0 <= index && index < num_elems);
        return elems[index];
    }
    ...

private:
    int num_elems;
    int capacity;
    std::string *elems;
};
```

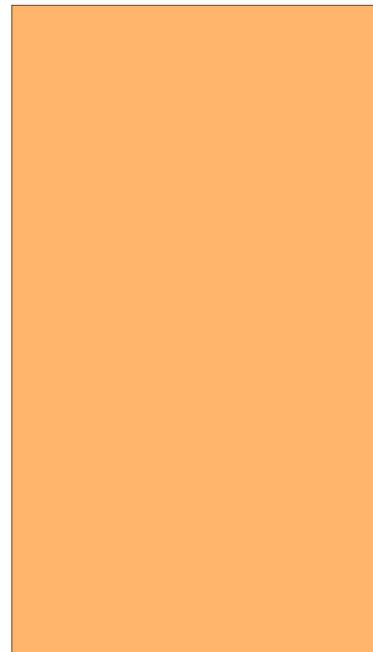


# Comportamiento en memoria de `at()`

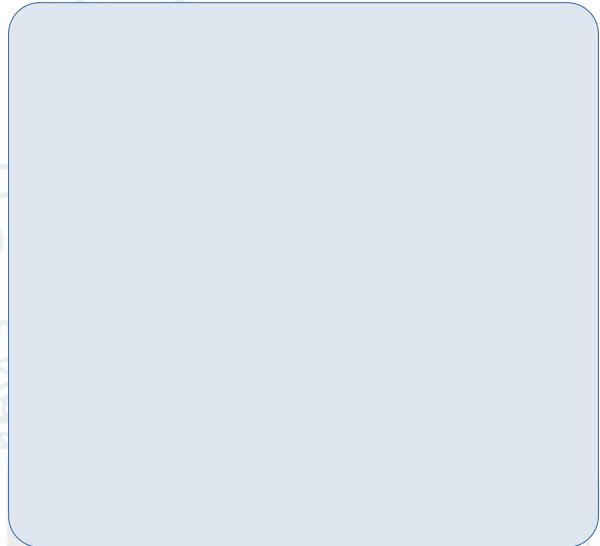
```
ListArray l;  
l.push_back("David");  
l.push_back("Maria");  
l.push_back("Elvira");  
  
std::string m = l.at(1);  
  
m = "Manuel"  
l.display();
```

[David, Maria, Elvira]

Pila



Heap



# Comportamiento en memoria de at()

```
class ListArray {
public:

    std::string & at(int index) const {
        assert (0 <= index && index < num_elems);
        return elems[index];
    }
    ...

private:
    int num_elems;
    int capacity;
    std::string *elems;
};
```

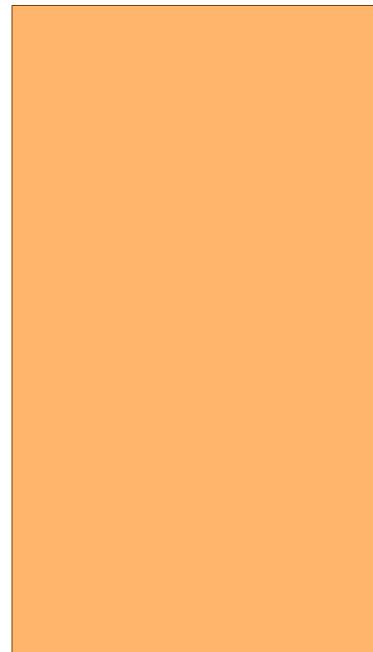


# ¿Y si `at()` devolviese una referencia?

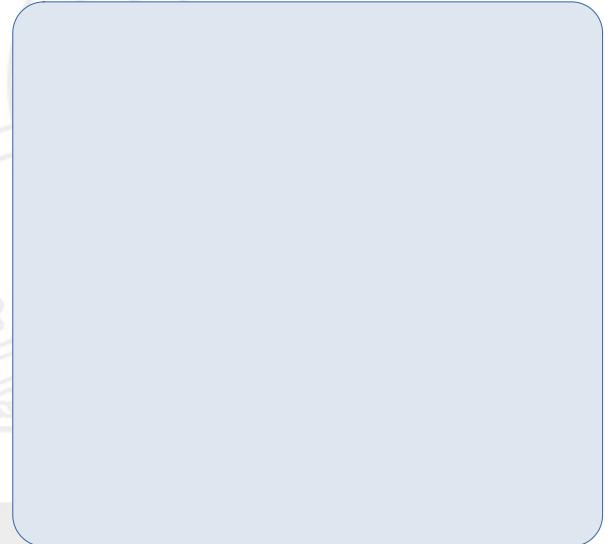
```
ListArray l;  
l.push_back("David");  
l.push_back("Maria");  
l.push_back("Elvira");  
  
std::string &m = l.at(1);  
m = "Manuel"  
  
l.display();
```

[David, Manuel, Elvira]

Pila



Heap



# ¿Y si `at()` devolviese una referencia?

```
ListArray l;  
l.push_back("David");  
l.push_back("Maria");  
l.push_back("Elvira");
```

`std::string &m = l.at(1);` ] equivale a ➔ `l.at(1) = "Manuel";`

```
l.display();
```

# Consecuencias

- Haciendo que `at()` devuelva una referencia al elemento del array permitimos la posibilidad de **actualizar** elementos de la lista, sin necesidad de necesitar un método específico para ello.
- Pero, a cambio, la función ha dejado de ser `const`. ☹
- Por ejemplo, la siguiente función dejaría de ser aceptada por el compilador:

```
int contar_caracteres(const ListArray &l) {  
    int suma = 0;  
    for (int i = 0; i < l.size(); i++) {  
        suma += l.at(i).length();  
    }  
    return suma;  
}
```

No puede llamarse a `at()`, porque `l` es una referencia constante.



# Solución: dos versiones para at()

```
class ListArray {
public:

    const std::string & at(int index) const {
        assert (0 <= index && index < num_elems);
        return elems[index];
    }

    std::string & at(int index) {
        assert (0 <= index && index < num_elems);
        return elems[index];
    }

    ...
};

};
```

Versión constante

Versión no constante

# Solución: dos versiones para at()

```
int contar_caracteres(const ListArray &l) {  
    int suma = 0;  
    for (int i = 0; i < l.size(); i++) {  
        suma += l.at(i).length();  
    }  
    return suma;  
}
```

Se llama a la versión  
constante de at()

```
ListArray l;  
...  
l.at(1) = "Manuel";
```

Se llama a la versión  
no constante de at()

# Referencias en front() y back()

```
const std::string & front() const {
    assert (num_elems > 0);
    return elems[0];
}

std::string & front() {
    assert (num_elems > 0);
    return elems[0];
}

const std::string & back() const {
    assert (num_elems > 0);
    return elems[num_elems - 1];
}

std::string & back() {
    assert (num_elems > 0);
    return elems[num_elems - 1];
}
```



# ¡Cuidado con las referencias!

```
int main() {
    ListArray l(3);
    l.push_back("Javier");
    l.push_back("Simona");
    l.push_back("Jerry");

    std::string &primero = l.front();

    l.push_back("David");

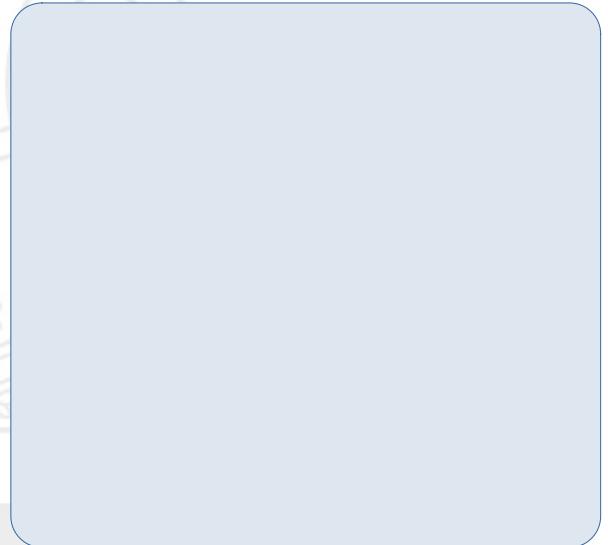
    primero = "Javier Francisco";

    return 0;
}
```

Pila



Heap



# ¡Cuidado con las referencias!

- Si se obtiene una referencia a un elemento de la lista, debe hacerse uso de esa referencia (para leer o modificar el valor apuntado por la referencia) **antes** de añadir o eliminar otros elementos de la lista.

```
l.front() = "Javier Francisco"; 
```

```
std::string &primero = l.front();   
...  
primero = "Javier Francisco";
```



ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Implementación del TAD Lista mediante listas enlazadas

Manuel Montenegro Montes

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

# Recordatorio: operaciones del TAD Lista

- **Constructoras:**
  - Crear una lista vacía: ***create\_empty()*** →  $L: List$
- **Mutadoras:**
  - Añadir un elemento al principio de la lista: ***push\_front(x: elem, L: List)***.
  - Añadir un elemento al final de la lista: ***push\_back(x: elem, L: List)***.
  - Eliminar el elemento del principio de la lista: ***pop\_front(L: List)***.
  - Eliminar el elemento del final de la lista: ***pop\_back(L: List)***.
- **Observadoras:**
  - Obtener el tamaño de la lista: ***size(L: List)*** →  $tam: int$ .
  - Comprobar si la lista es vacía ***empty(L: List)*** →  $b: bool$ .
  - Acceder al primer elemento de la lista ***front(L: List)*** →  $e: elem$ .
  - Acceder al último elemento de la lista ***back(L: List)*** →  $e: elem$ .
  - Acceder a un elemento que ocupa una posición determinada ***at(idx: int, L: List)*** →  $e: elem$ .

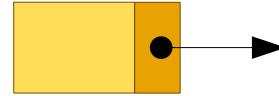
# ¿Qué es una lista enlazada?

- Secuencia de **nodos**, en la que cada nodo contiene:
  - Un campo con información arbitraria.
  - Un puntero al siguiente nodo de la secuencia.
- En este caso, decimos que son **listas enlazadas simples**.



# Definición de un nodo

```
struct Node {  
    std::string value;  
    Node *next;  
};
```



- Cuando un nodo no tiene sucesor, su campo next contiene el puntero nulo (`nullptr` en C++).

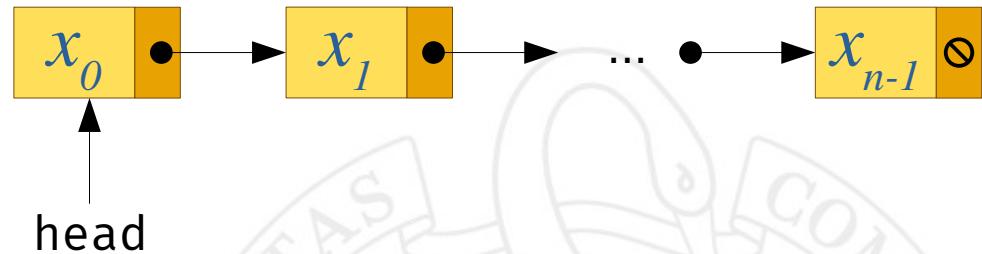


# Ejemplo

```
struct Node {  
    std::string value;  
    Node *next;  
};  
  
Node *tres = new Node { "Tres", nullptr };  
Node *dos = new Node { "Dos", tres };  
Node *uno = new Node { "Uno", dos };
```

# El TAD Lista mediante listas enlazadas

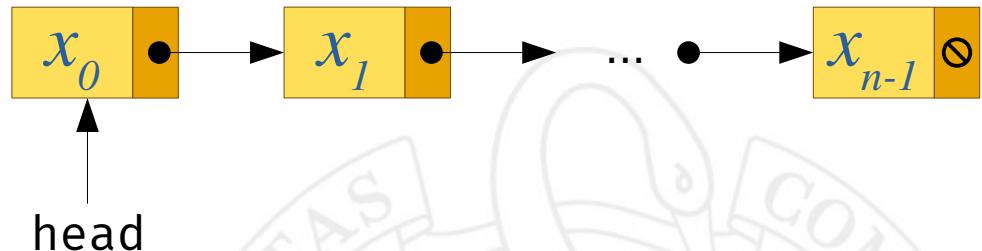
[  $x_0, x_1, \dots, x_{n-1}$  ]



```
class ListLinkedSingle {  
public:  
    ...  
private:  
    struct Node { ... };  
    Node *head;  
};
```

# El TAD Lista mediante listas enlazadas

[  $x_0, x_1, \dots, x_{n-1}$  ]



- Invariante de representación:  
 $I(x) = \text{true}$
- Función de abstracción:  
 $f(x) = [ x.\text{head} \rightarrow \text{value}, x.\text{head} \rightarrow \text{next} \rightarrow \text{value}, x.\text{head} \rightarrow \text{next} \rightarrow \text{next} \rightarrow \text{value}, \dots ]$

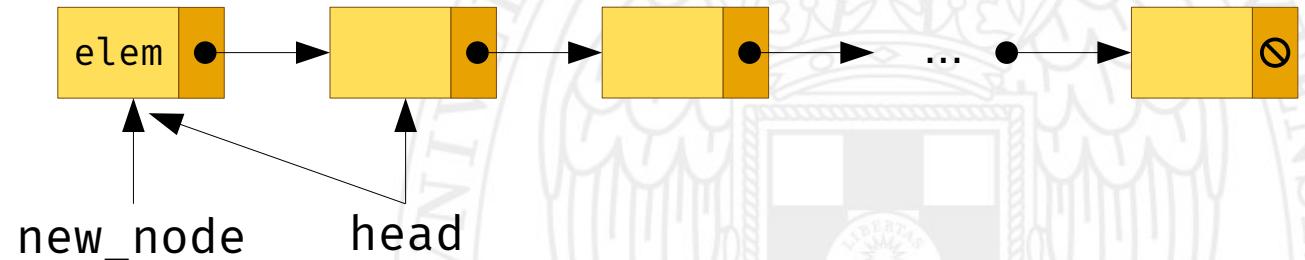
# Inicializar lista

```
class ListLinkedSingle {  
public:  
    ListLinkedSingle(): head(nullptr) { }  
    ...  
  
private:  
    struct Node { ... };  
    Node *head;  
};
```



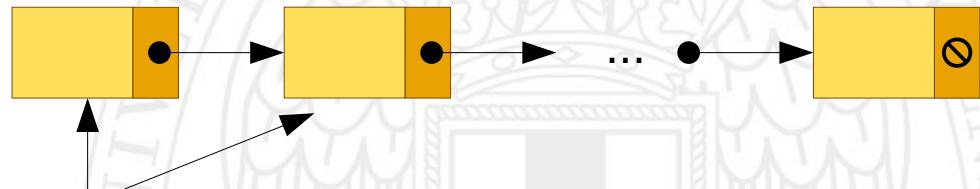
# Añadir un elemento al principio de la lista

```
class ListLinkedSingle {  
public:  
    ...  
    void push_front(const std::string &elem) {  
        Node *new_node = new Node { elem, head };  
        head = new_node;  
    }  
  
private:  
    struct Node { ... };  
    Node *head;  
};
```



# Eliminar un elemento del principio de la lista

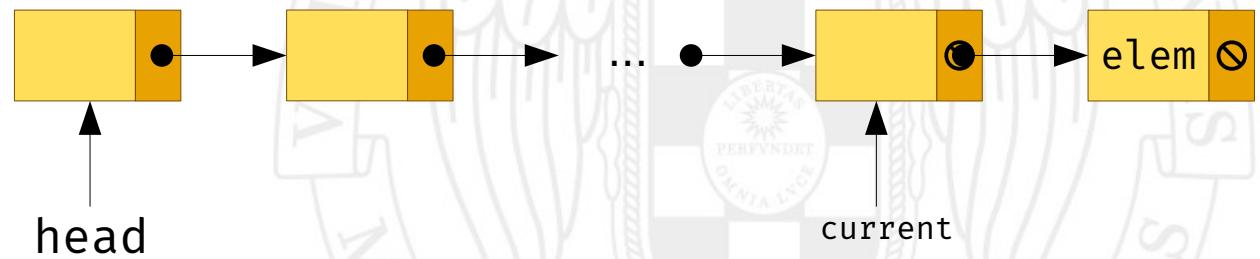
```
class ListLinkedSingle {  
public:  
    ...  
    void pop_front() {  
        assert (head != nullptr);  
        Node *old_head = head;  
        head = head->next;  
        delete old_head;  
    }  
  
private:  
    struct Node { ... };  
    Node *head;  
};
```



head

# Añadir un elemento al final de la lista

```
class ListLinkedSingle {  
public:  
    ...  
    void push_back(const std::string &elem);  
    ...  
};  
  
void ListLinkedSingle::push_back(const std::string &elem) {  
    Node *new_node = new Node { elem, nullptr };  
    if (head == nullptr) {  
        head = new_node;  
    } else {  
        Node *current = head;  
        while (current->next != nullptr) {  
            current = current->next;  
        }  
        current->next = new_node;  
    }  
}
```



# Refactorizando: obtener el último nodo

```
ListLinkedSingle::Node * ListLinkedSingle::last_node() const {
    assert (head != nullptr);
    Node *current = head;
    while (current->next != nullptr) {
        current = current->next;
    }
    return current;
}

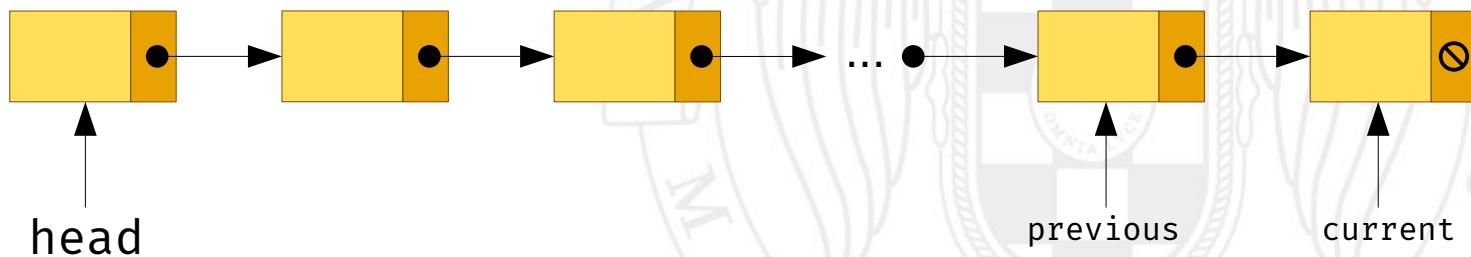
void ListLinkedSingle::push_back(const std::string &elem) {
    Node *new_node = new Node { elem, nullptr };
    if (head == nullptr) {
        head = new_node;
    } else {
        last_node()->next = new_node;
    }
}
```

# Eliminar un elemento del final de la lista

```
void ListLinkedSingle::pop_back() {
    assert (head != nullptr);
    if (head->next == nullptr) {
        delete head;
        head = nullptr;
    } else {
        Node *previous = head;
        Node *current = head->next;

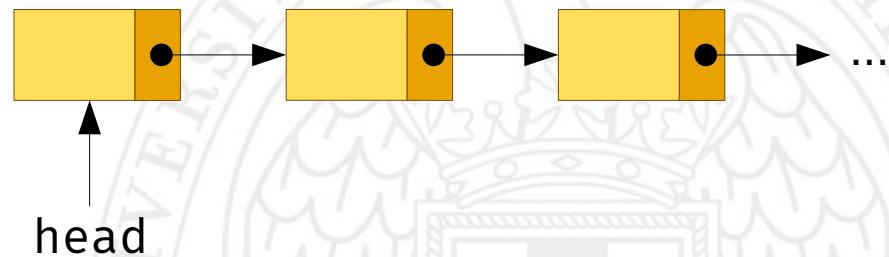
        while (current->next != nullptr) {
            previous = current;
            current = current->next;
        }

        delete current;
        previous->next = nullptr;
    }
}
```



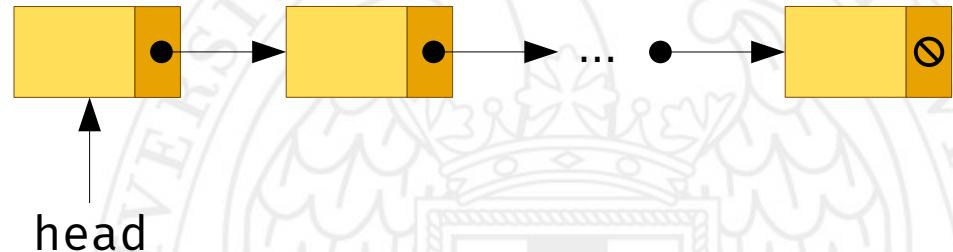
# Acceder al primer elemento de la lista

```
class ListLinkedSingle {  
public:  
    ...  
    const std::string & front() const {  
        assert (head != nullptr);  
        return head->value;  
    }  
  
    std::string & front() { ... }  
  
private:  
    struct Node { ... };  
    Node *head;  
};
```



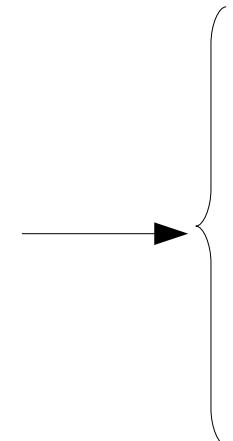
# Acceder al último elemento de la lista

```
class ListLinkedSingle {  
public:  
    ...  
    const std::string & back() const {  
        return last_node()→value;  
    }  
  
    std::string & back() { ... }  
  
private:  
    struct Node { ... };  
    Node *head;  
  
    Node *last_node() const;  
};
```



# Acceder al elemento $n$ -ésimo de la lista

```
class ListLinkedSingle {  
public:  
    ...  
    const std::string & at(int index) const {  
        Node *result_node = nth_node(index);  
        assert (result_node != nullptr);  
        return result_node->value;  
    }  
  
    std::string & at(int index) { ... }  
  
private:  
    struct Node { ... };  
    Node *head;  
  
    Node *last_node() const;  
    Node *nth_node(int n) const;  
};
```



```
Node * ListLinkedSingle::nth_node(int n) const {  
    assert (0 <= n);  
    int current_index = 0;  
    Node *current = head;  
  
    while (current_index < n && current != nullptr) {  
        current_index++;  
        current = current->next;  
    }  
  
    return current;  
}
```

# Obtener el tamaño de una lista

```
class ListLinkedSingle {  
public:  
    int size() const;  
  
    bool empty() const {  
        return head == nullptr;  
    };  
    ...  
};  
  
int ListLinkedSingle::size() const {  
    int num_nodes = 0;  
  
    Node *current = head;  
    while (current != nullptr) {  
        num_nodes++;  
        current = current->next;  
    }  
  
    return num_nodes;  
}
```



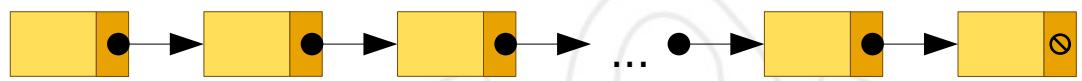
# Mostrar una lista por pantalla

```
void ListLinkedSingle::display(std::ostream &out) const {
    std::cout << "[";
    if (head != nullptr) {
        out << head->value;
        Node *current = head->next;
        while (current != nullptr) {
            out << ", " << current->value;
            current = current->next;
        }
    }
    out << "]";
}
```



# Destrucción de una lista

```
class ListLinkedSingle {  
public:  
    ...  
    ~ListLinkedSingle() {  
        delete_list(head);  
    }  
  
private:  
    ...  
    void delete_list(Node *start_node);  
}  
  
void ListLinkedSingle::delete_list(Node *start_node) {  
    if (start_node != nullptr) {  
        delete_list(start_node->next);  
        delete start_node;  
    }  
}
```



ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Constructores de copia en el TAD Lista

Manuel Montenegro Montes

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

# Implementaciones del TAD Lista

- Mediante vectores.
- Mediante listas enlazadas simples.



# Implementación mediante vectores

# Implementación mediante vectores

- El constructor de copia por defecto para la clase `ListArray` no nos sirve.

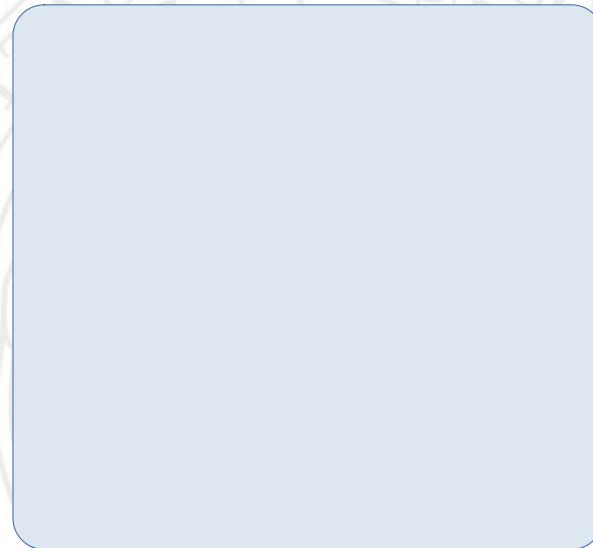
```
ListArray l1;  
l1.push_back("David");  
l1.push_back("Maria");  
l1.push_back("Eugenio");
```

```
ListArray l2 = l1;  
l2.front() = "Pepe";
```

Pila

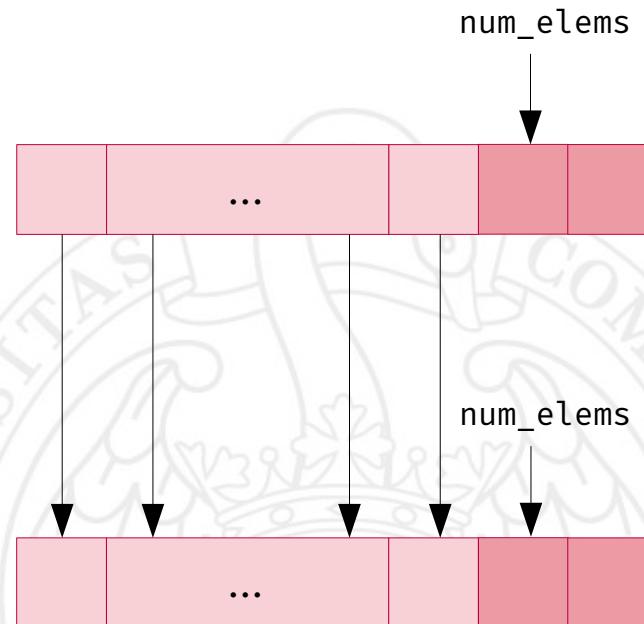


Heap



# Constructor de copia para ListArray

```
class ListArray {  
public:  
    ...  
    ListArray(const ListArray &other);  
    ...  
};  
  
ListArray::ListArray(const ListArray &other)  
: num_elems(other.num_elems),  
 capacity(other.capacity),  
 elems(new std::string[other.capacity])  
{  
    for (int i = 0; i < num_elems; i++) {  
        elems[i] = other.elems[i];  
    }  
}
```

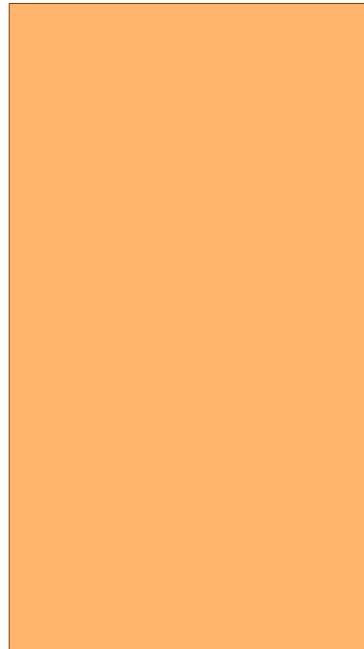


# Ejemplo

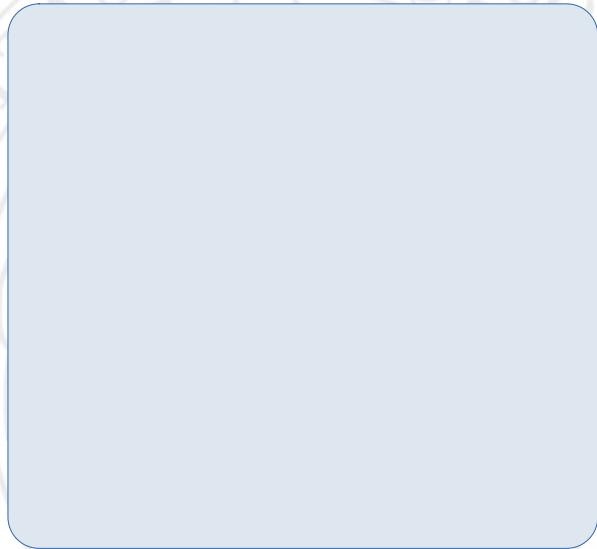
- Con el constructor de copia, l1 y l2 pueden ser modificadas de manera independiente.

```
ListArray l1;  
l1.push_back("David");  
l1.push_back("Maria");  
l1.push_back("Eugenio");  
  
ListArray l2 = l1;  
l2.front() = "Pepe";
```

Pila



Heap



# Implementación mediante listas enlazadas

# Implementación mediante listas enlazadas

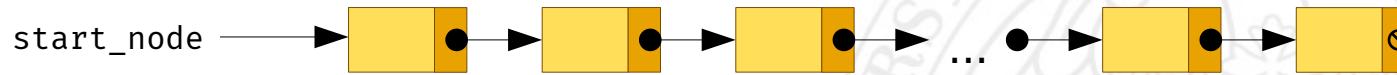
- Creamos una función auxiliar (`copy_nodes`) para producir una copia de una lista enlazada de nodos.
- El constructor de copia inicializa la cabeza creando una copia de la lista enlazada original.

```
class ListLinkedSingle {  
public:  
    ListLinkedSingle(const ListLinkedSingle &other)  
        : head(copy_nodes(other.head)) { }  
  
private:  
    Node *head;  
  
...  
    Node *copy_nodes(Node *start_node) const;  
};
```



# Implementación recursiva de copy\_nodes

```
ListLinkedSingle::Node * ListLinkedSingle::copy_nodes(Node *start_node) const {  
    if (start_node != nullptr) {  
        Node *result = new Node { start_node->value, copy_nodes(start_node->next) };  
        return result;  
    } else {  
        return nullptr;  
    }  
}
```



ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Nodos fantasma

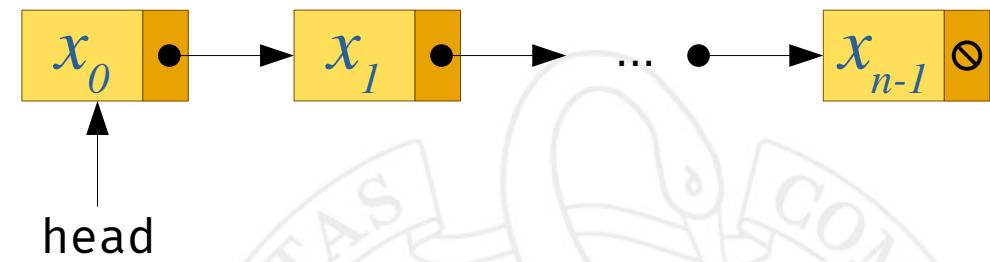
Manuel Montenegro Montes

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

# Recordatorio

$[x_0, x_1, \dots, x_{n-1}]$

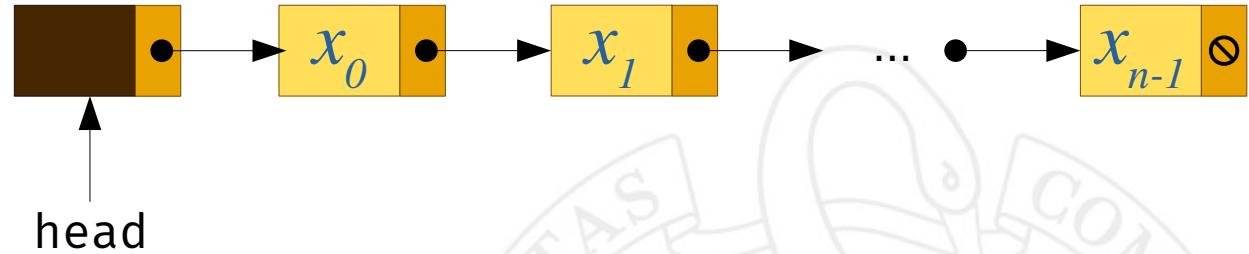
$[]$



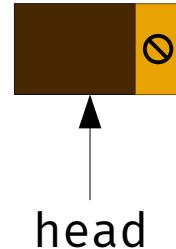
`head = nullptr`

# Introduciendo un nodo fantasma

$[x_0, x_1, \dots, x_{n-1}]$



$[]$



# Nodo fantasma

- Es un nodo que se sitúa siempre al principio de la lista enlazada de nodos.
- La información que contiene (esto es, su atributo `value`) es irrelevante.
- El atributo `head` de la lista apunta siempre a este nodo fantasma.  
    ⇒ **head nunca va a tomar el valor `nullptr`.**

**Consecuencia:** simplificación de la implementación de algunos métodos.

# Interfaz del TAD Lista

```
class ListLinkedSingle {  
public:  
    ListLinkedSingle();  
    ListLinkedSingle(const ListLinkedSingle &other);  
    ~ListLinkedSingle();  
  
    void push_front(const std::string &elem);  
    void push_back(const std::string &elem);  
    void pop_front();  
    void pop_back();  
    int size() const;  
    bool empty() const;  
    const std::string & front() const;  
    std::string & front();  
    const std::string & back() const;  
    std::string & back();  
    const std::string & at(int index) const;  
    std::string & at(int index);  
    void display() const;  
private:  
    ...  
};
```

```
ListLinkedSingle() {  
    head = new Node;  
    head->next = nullptr;  
}
```



# Cambios en la implementación

```
class ListLinkedSingle {  
public:  
    ListLinkedSingle();  
    ListLinkedSingle(const ListLinkedSingle &other);  
    ~ListLinkedSingle();  
  
    void push_front(const std::string &elem);  
    void push_back(const std::string &elem);  
    void pop_front();  
    void pop_back();  
    int size() const;  
    bool empty() const;  
    const std::string & front() const;  
    std::string & front();  
    const std::string & back() const;  
    std::string & back();  
    const std::string & at(int index) const;  
    std::string & at(int index);  
    void display() const;  
private:  
    ...  
};
```

- El constructor de copia y el destructor no cambian con la incorporación de nodos fantasma.
- Tampoco varían los métodos privados asociados:

`delete_list()`  
`copy_nodes()`

# Cambios en la implementación

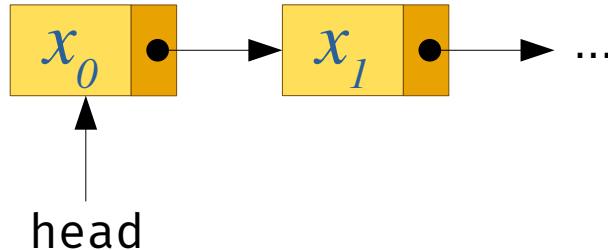
```
class ListLinkedSingle {  
public:  
    ListLinkedSingle();  
    ListLinkedSingle(const ListLinkedSingle &other);  
    ~ListLinkedSingle();  
  
    void push_front(const std::string &elem);  
    void push_back(const std::string &elem);  
    void pop_front();  
    void pop_back();  
    int size() const;  
    bool empty() const;  
    const std::string & front() const;  
    std::string & front();  
    const std::string & back() const;  
    std::string & back();  
    const std::string & at(int index) const;  
    std::string & at(int index);  
    void display() const;  
private:  
    ...  
};
```

- La mayoría de operaciones requieren cambios triviales.
- Por ejemplo:  
`assert(head != nullptr)`  
se transforma en  
`assert(head->next != nullptr)`
- Las operaciones de iteración comienzan en `head->next` en lugar de en `head`.

# Ejemplo: método front()

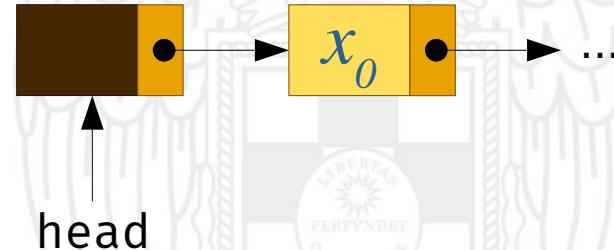
Antes

```
std::string & front() {  
    assert (head != nullptr);  
    return head->value;  
}
```



Después

```
std::string & front() {  
    assert (head->next != nullptr);  
    return head->next->value;  
}
```



# Ejemplo: método nth\_node()

## Antes

```
Node *nth_node(int n) const {
    assert (0 ≤ n);
    int current_index = 0;
    Node *current = head;

    while (current_index < n
           && current ≠ nullptr) {
        current_index++;
        current = current→next;
    }
    return current;
}
```

## Después

```
Node *nth_node(int n) const {
    assert (0 ≤ n);
    int current_index = 0;
    Node *current = head→next;

    while (current_index < n
           && current ≠ nullptr) {
        current_index++;
        current = current→next;
    }
    return current;
}
```

# Cambios en la implementación

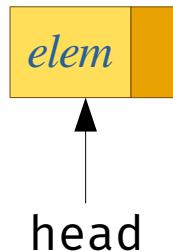
```
class ListLinkedSingle {  
public:  
    ListLinkedSingle();  
    ListLinkedSingle(const ListLinkedSingle &other);  
    ~ListLinkedSingle();  
  
    void push_front(const std::string &elem);  
    void push_back(const std::string &elem);  
    void pop_front();  
    void pop_back();  
    int size() const;  
    bool empty() const;  
    const std::string & front() const;  
    std::string & front();  
    const std::string & back() const;  
    std::string & back();  
    const std::string & at(int index) const;  
    std::string & at(int index);  
    void display() const;  
private:  
    ...  
};
```

- La implementación de las operaciones `push_back()` y `pop_back()` se simplifican, ya que no tienen que comprobar si la lista es vacía o no.

# Cambios en push\_back()

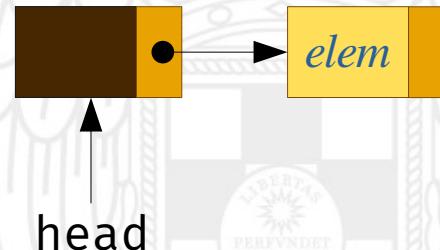
## Antes

```
void push_back(const std::string &elem) {  
    Node *new_node = new Node { elem, nullptr };  
    if (head == nullptr) {  
        head = new_node;  
    } else {  
        last_node()→next = new_node;  
    }  
}
```



## Después

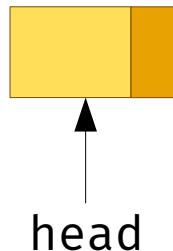
```
void push_back(const std::string &elem) {  
    Node *new_node = new Node { elem, nullptr };  
    last_node()→next = new_node;  
}
```



# Cambios en pop\_back()

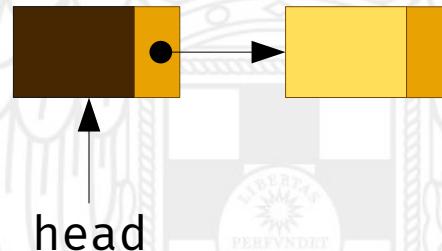
## Antes

```
void pop_back() {
    assert (head != nullptr);
    if (head->next = nullptr) {
        delete head;
        head = nullptr;
    } else {
        // borrar último nodo
    }
}
```



## Después

```
void pop_back() {
    assert (head->next != nullptr);
    // borrar último nodo
}
```



# Conclusiones

## Ventajas

- Simplificación en las implementaciones.

## Desventajas

- Un nodo extra en memoria.
- La inicialización del nodo fantasma requiere un constructor por defecto.



ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

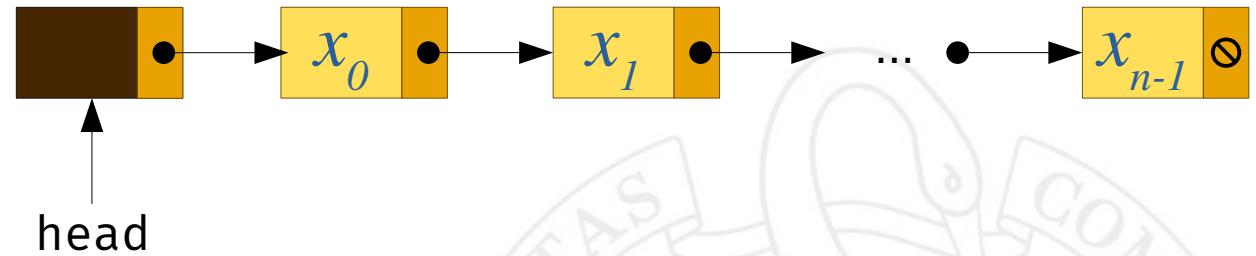
# Listas doblemente enlazadas (1)

Manuel Montenegro Montes

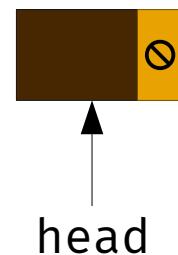
Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Recordatorio: listas enlazadas simples

$[x_0, x_1, \dots, x_{n-1}]$

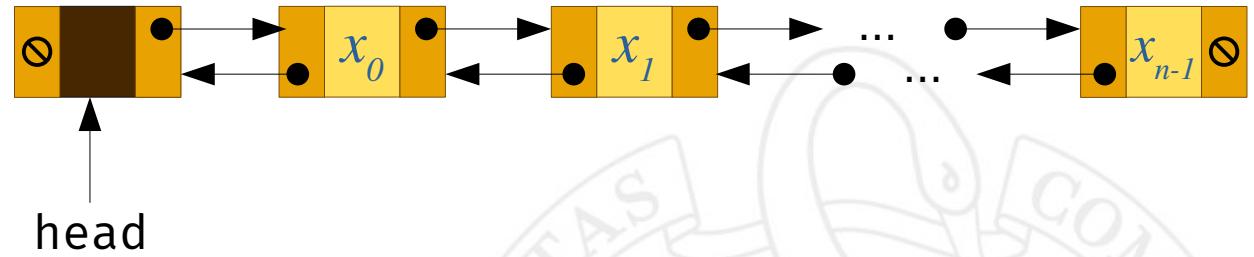


$[]$

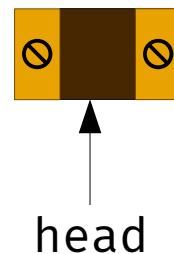


# Listas doblemente enlazadas

$[ x_0, x_1, \dots, x_{n-1} ]$



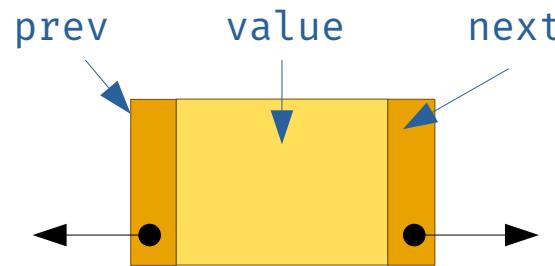
$[]$



# Listas enlazadas dobles

```
struct Node {  
    std::string value;  
    Node *next;  
    Node *prev;  
};
```

- Cada nodo tiene dos punteros:
  - next**: Nodo siguiente en la lista enlazada.
  - prev**: Nodo anterior en la lista enlazada.



# Implementación: ListLinkedDouble

```
class ListLinkedDouble {
public:
    ListLinkedDouble();
    ListLinkedDouble(const ListLinkedDouble &other);
    ~ListLinkedDouble();

    void push_front(const std::string &elem);
    void push_back(const std::string &elem);
    void pop_front();
    void pop_back();
    int size() const;
    bool empty() const;
    const std::string & front() const;
    std::string & front();
    const std::string & back() const;
    std::string & back();
    const std::string & at(int index) const;
    std::string & at(int index);
    void display() const;
private:
    ...
    Node *head;
};
```



# Constructores y destructor

```
ListLinkedDouble() {  
    head = new Node;  
    head->next = nullptr;  
    head->prev = nullptr;  
}
```

```
~ListLinkedDouble() {  
    delete_list(head);  
}
```

```
ListLinkedDouble(const ListLinkedDouble &other)  
: head(copy_nodes(other.head)) { }
```

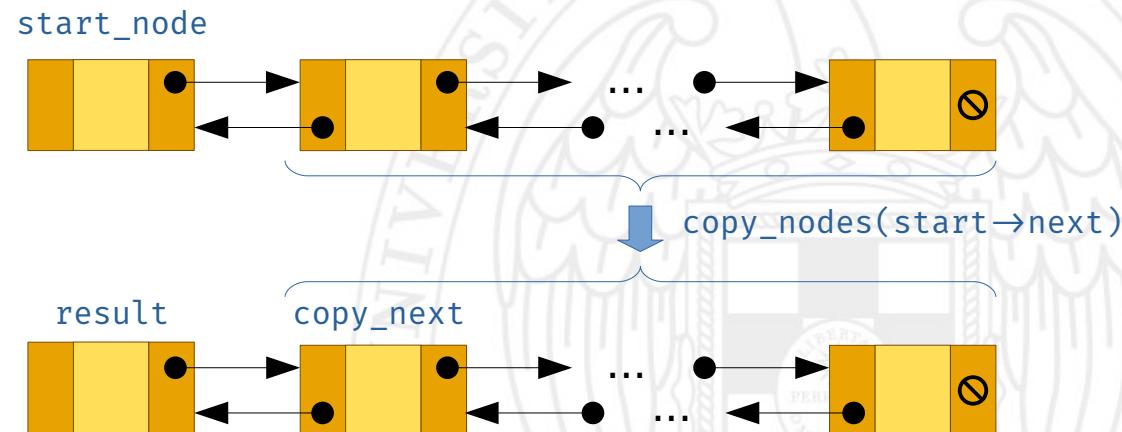


head



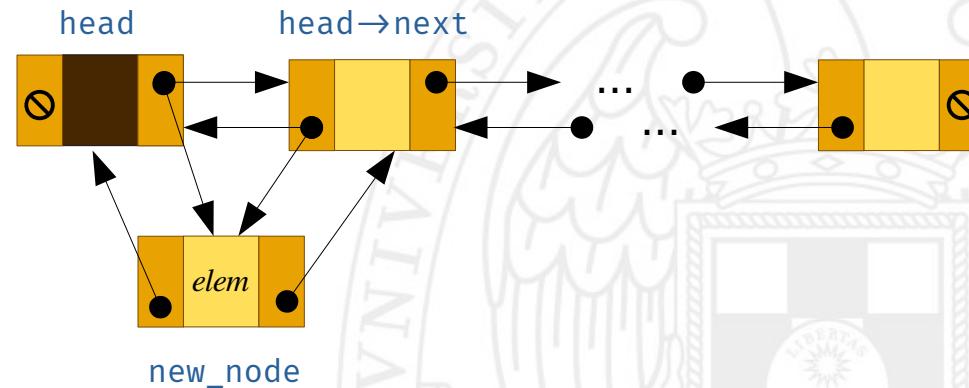
# Copia de una cadena de nodos

```
Node * ListLinkedDouble::copy_nodes(Node *start_node) const {
    if (start_node != nullptr) {
        Node *copy_next = copy_nodes(start_node->next);
        Node *result = new Node { start_node->value, copy_next, nullptr };
        if (copy_next != nullptr) {
            copy_next->prev = result;
        }
        return result;
    } else {
        return nullptr;
    }
}
```



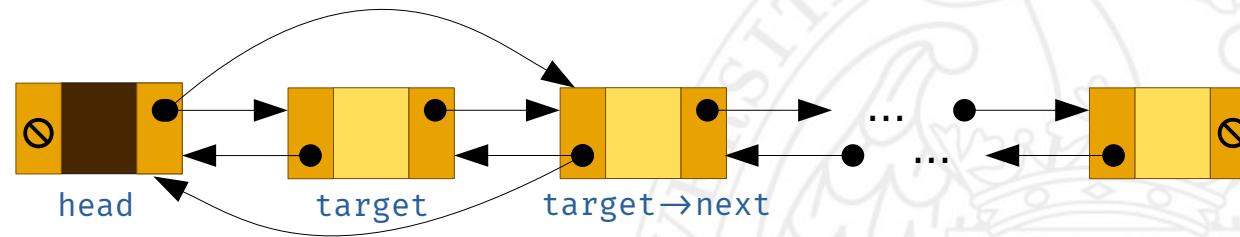
# Insertar al principio de la cadena

```
void push_front(const std::string &elem) {  
    Node *new_node = new Node { elem, head->next, head };  
    if (head->next != nullptr) {  
        head->next->prev = new_node;  
    }  
    head->next = new_node;  
}
```



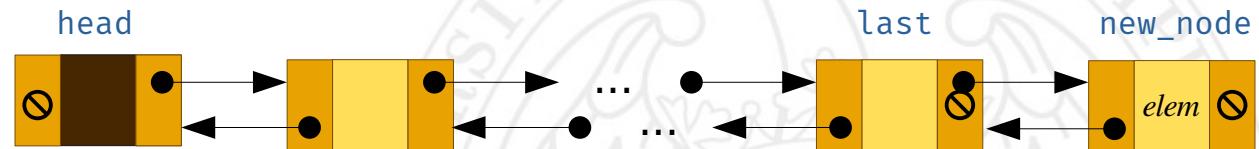
# Eliminar al principio de la cadena

```
void pop_front() {
    assert (head->next != nullptr);
    Node *target = head->next;
    head->next = target->next;
    if (target->next != nullptr) {
        target->next->prev = head;
    }
    delete target;
}
```



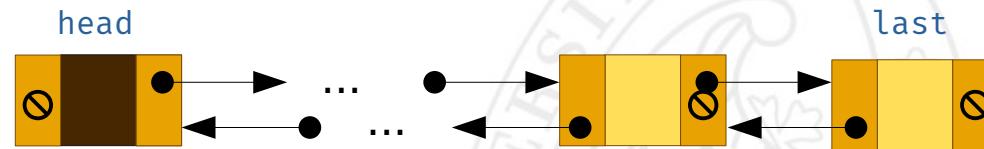
# Insertar al final de la cadena

```
void push_back(const std::string &elem) {
    Node *last = last_node();
    Node *new_node = new Node { elem, nullptr, last };
    last->next = new_node;
}
```



# Eliminar al final de la cadena

```
void pop_back() {
    assert (head->next != nullptr);
    Node *last = last_node();
    last->prev->next = nullptr;
    delete last;
}
```



# ¿Mejoras en el coste?

Operación	Listas enlazadas simples	Listas doblemente enlazadas
Creación	$O(1)$	$O(1)$
Copia	$O(n)$	$O(n)$
push_back	$O(n)$	$O(n)$
push_front	$O(1)$	$O(1)$
pop_back	$O(n)$	$O(n)$
pop_front	$O(1)$	$O(1)$
back	$O(n)$	$O(n)$
front	$O(1)$	$O(1)$
display	$O(n)$	$O(n)$
at(index)	$O(index)$	$O(index)$
size	$O(n)$	$O(n)$
empty	$O(1)$	$O(1)$

$n$  = número de elementos de la lista de entrada

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

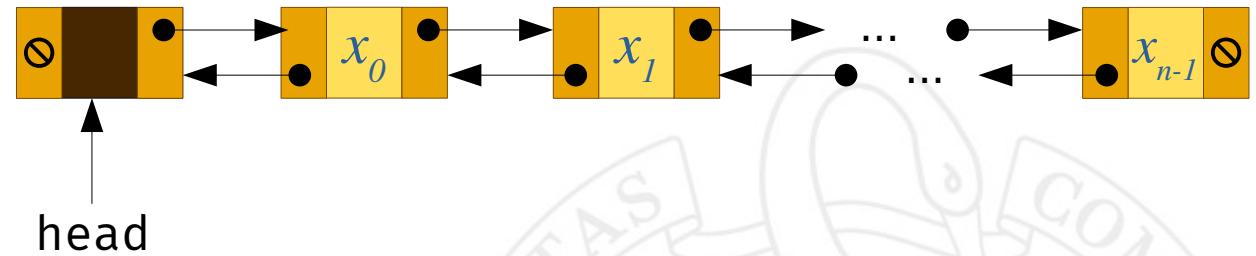
# Listas doblemente enlazadas (2)

Manuel Montenegro Montes

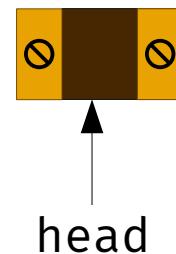
Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Listas doblemente enlazadas

$[ x_0, x_1, \dots, x_{n-1} ]$

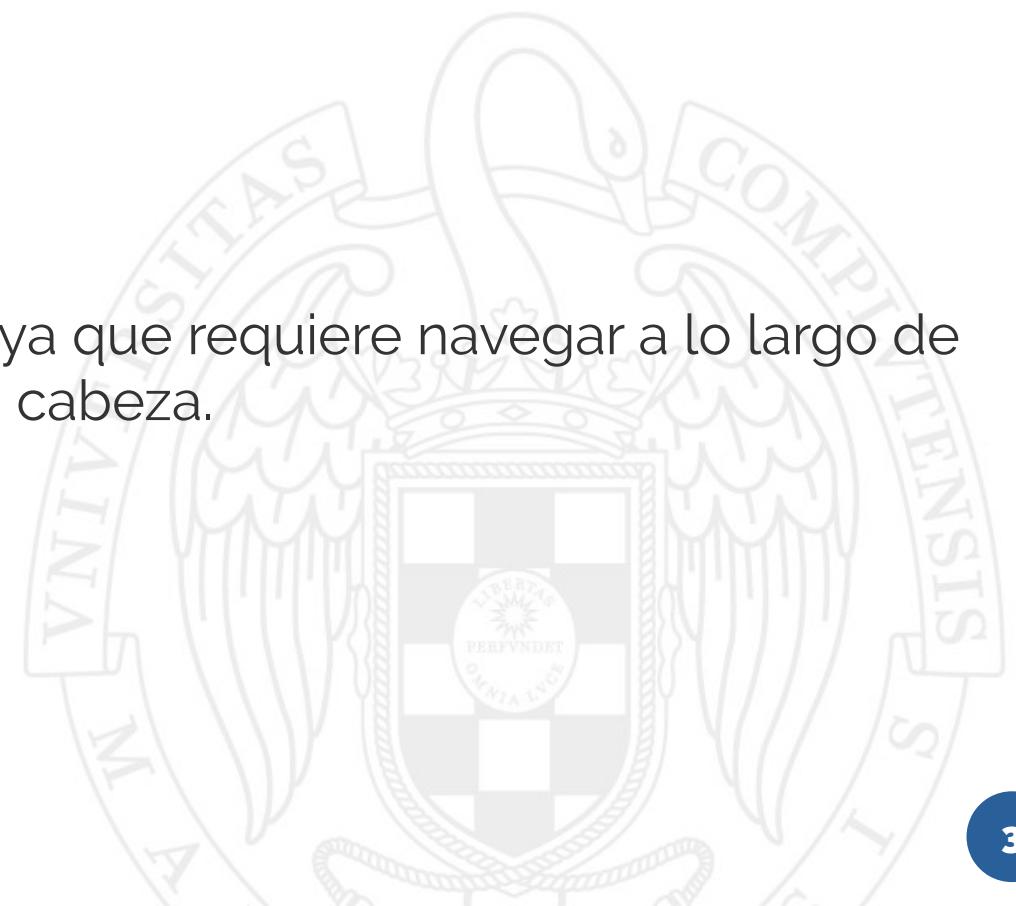


$[]$



# Mejorando algunas operaciones

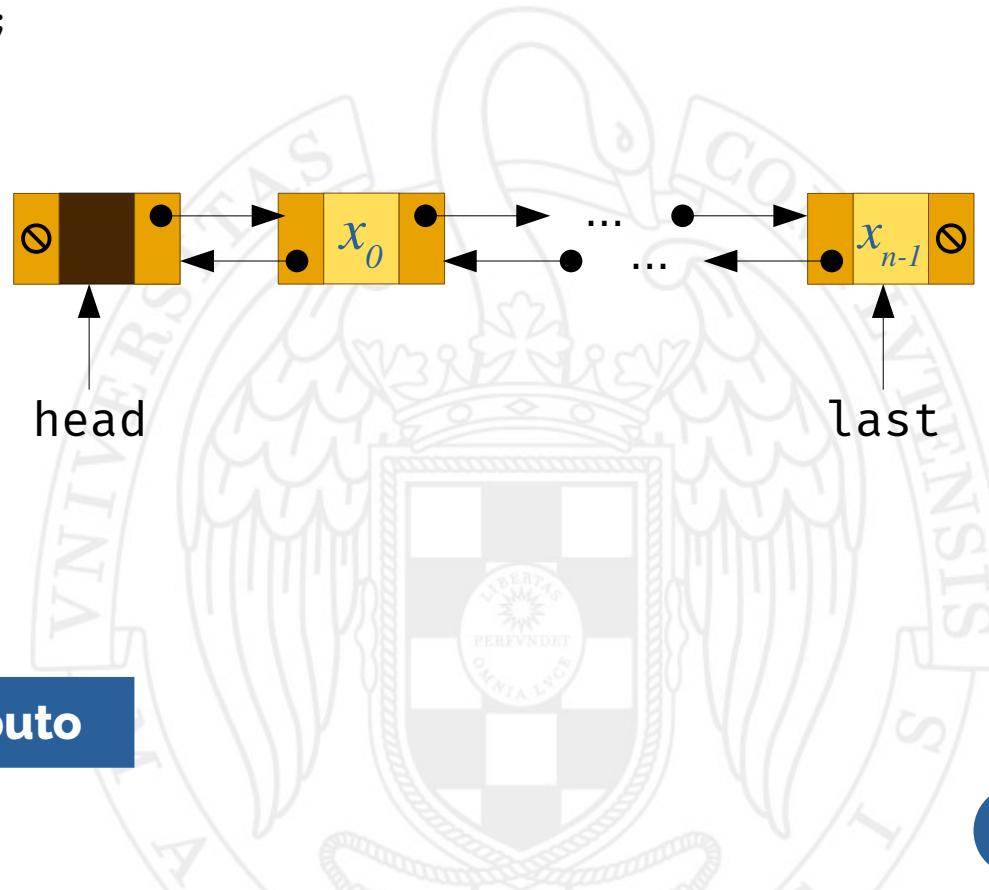
- Las siguientes operaciones requieren situarnos en el último nodo de la lista:
  - `push_back()`
  - `pop_back()`
  - `back()`
- Esto hace que tengan coste lineal, ya que requiere navegar a lo largo de toda la cadena, partiendo del nodo cabeza.
- ¿Podemos mejorar esto?



# Añadiendo un nuevo atributo

```
class ListLinkedDouble {  
public:  
    ListLinkedDouble();  
    ListLinkedDouble(const ListLinkedDouble &other);  
    ~ListLinkedDouble();  
  
    void push_front(const std::string &elem);  
    void push_back(const std::string &elem);  
    void pop_front();  
    void pop_back();  
    int size() const;  
    bool empty() const;  
    const std::string & front() const;  
    std::string & front();  
    const std::string & back() const;  
    std::string & back();  
    const std::string & at(int index) const;  
    std::string & at(int index);  
    void display() const;  
private:  
    ...  
    Node *head, *last;
```

Nuevo atributo

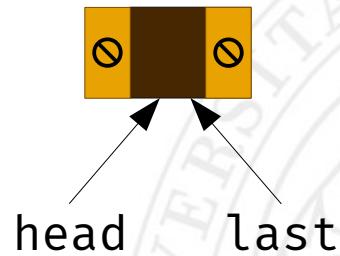


# Añadiendo un nuevo atributo

- **Ventajas:**
  - La operación privada `last_node()` pasa a tener coste constante, ya que se limita a devolver el atributo `last`.
  - De hecho, podemos eliminar la función `last_node()`.
- **Desventajas:**
  - Tenemos que actualizar el atributo `last` cada vez que añadamos un nodo.

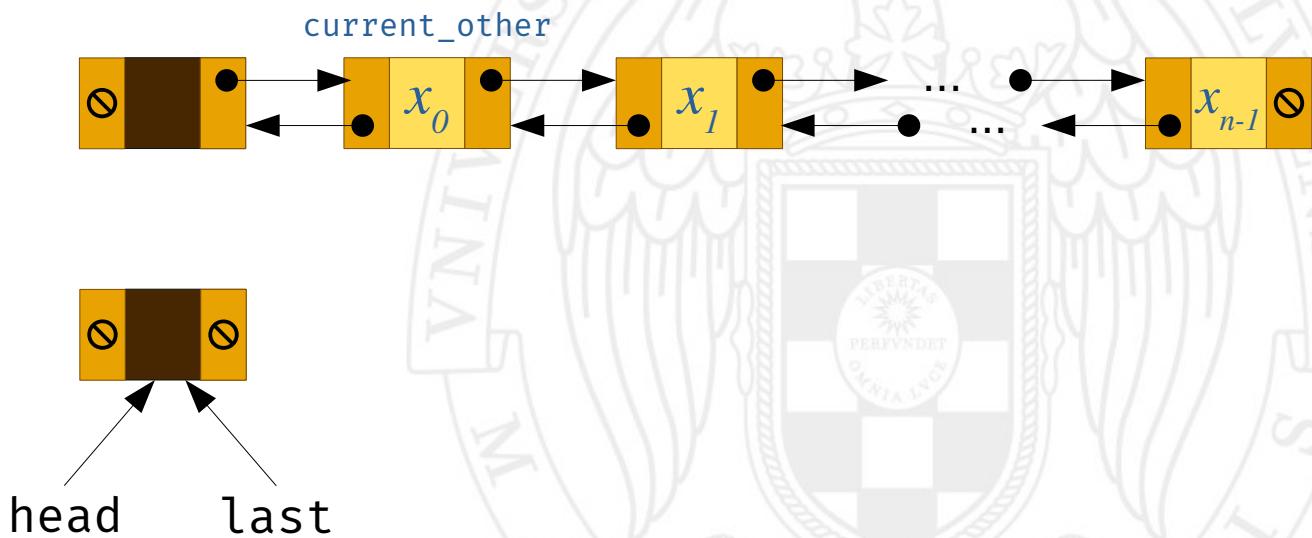
# Creación de una cadena de nodos

```
ListLinkedDouble() {  
    head = new Node;  
    head→next = nullptr;  
    head→prev = nullptr;  
    last = head;  
}
```



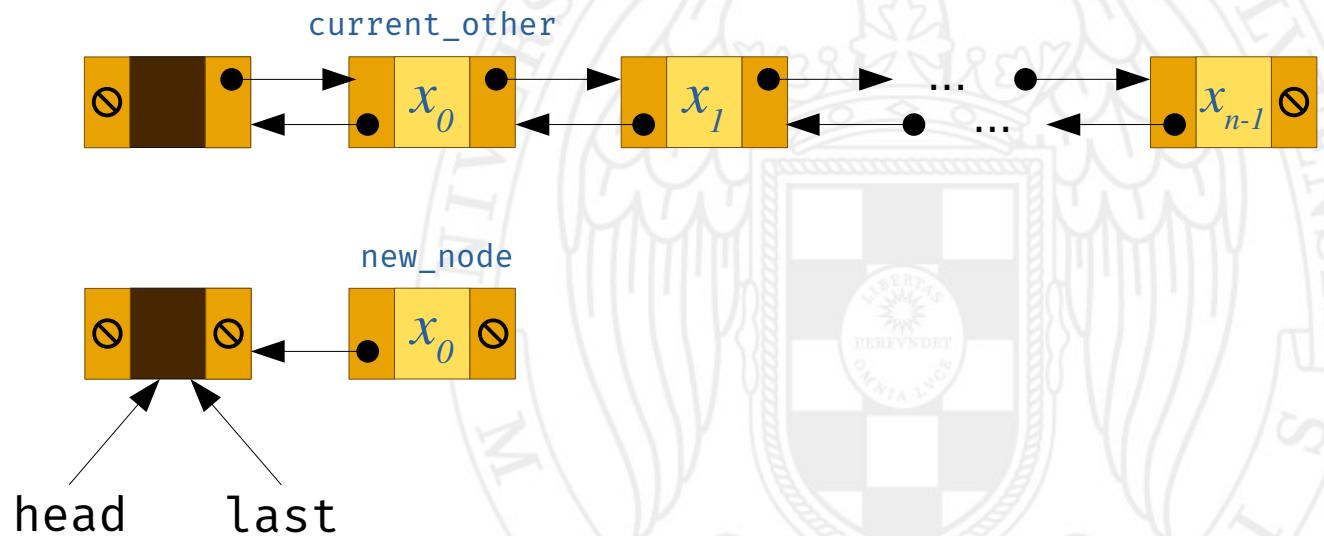
# Copia de una cadena de nodos

```
ListLinkedDouble(const ListLinkedDouble &other): ListLinkedDouble() {  
    Node *current_other = other.head->next;  
  
    while (current_other != nullptr) {  
        Node *new_node = new Node { current_other->value, nullptr, last };  
        last->next = new_node;  
  
        last = new_node;  
        current_other = current_other->next;  
    }  
}
```



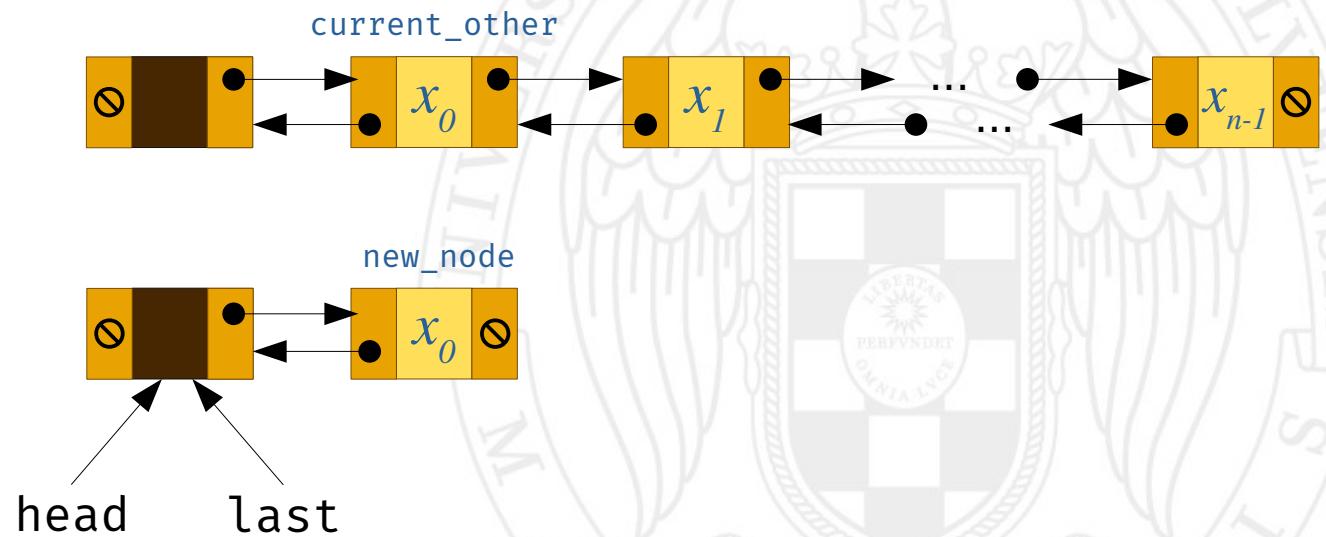
# Copia de una cadena de nodos

```
ListLinkedDouble(const ListLinkedDouble &other): ListLinkedDouble() {  
    Node *current_other = other.head->next;  
  
    while (current_other != nullptr) {  
        Node *new_node = new Node { current_other->value, nullptr, last };  
        last->next = new_node;  
        last = new_node;  
        current_other = current_other->next;  
    }  
}
```



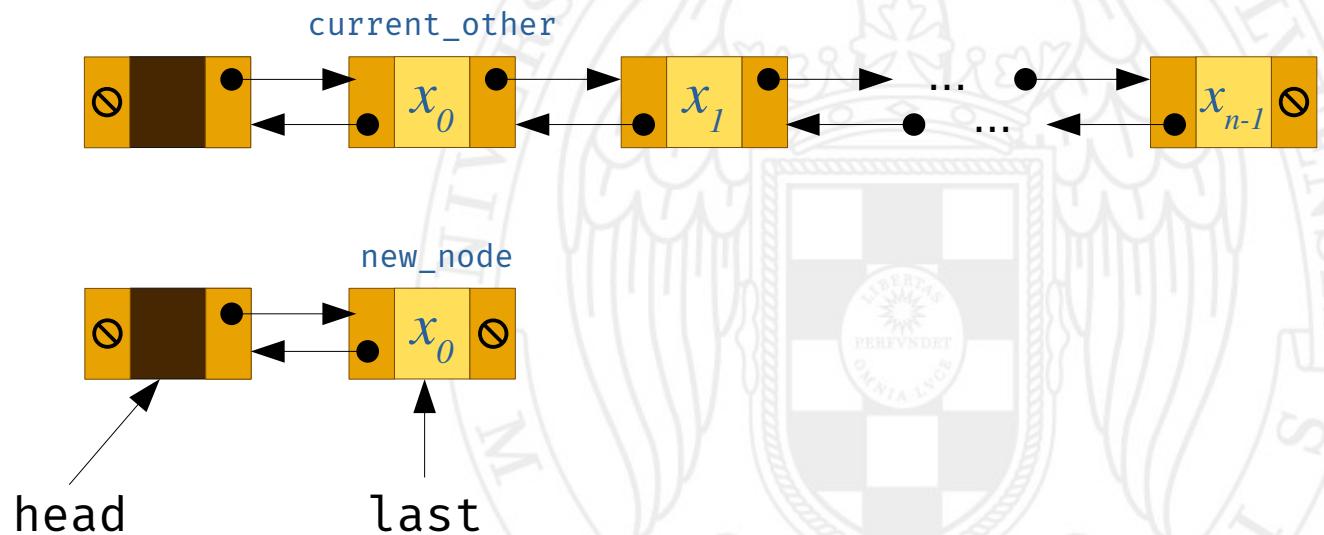
# Copia de una cadena de nodos

```
ListLinkedDouble(const ListLinkedDouble &other): ListLinkedDouble() {  
    Node *current_other = other.head->next;  
  
    while (current_other != nullptr) {  
        Node *new_node = new Node { current_other->value, nullptr, last };  
        last->next = new_node;  
        last = new_node;  
        current_other = current_other->next;  
    }  
}
```



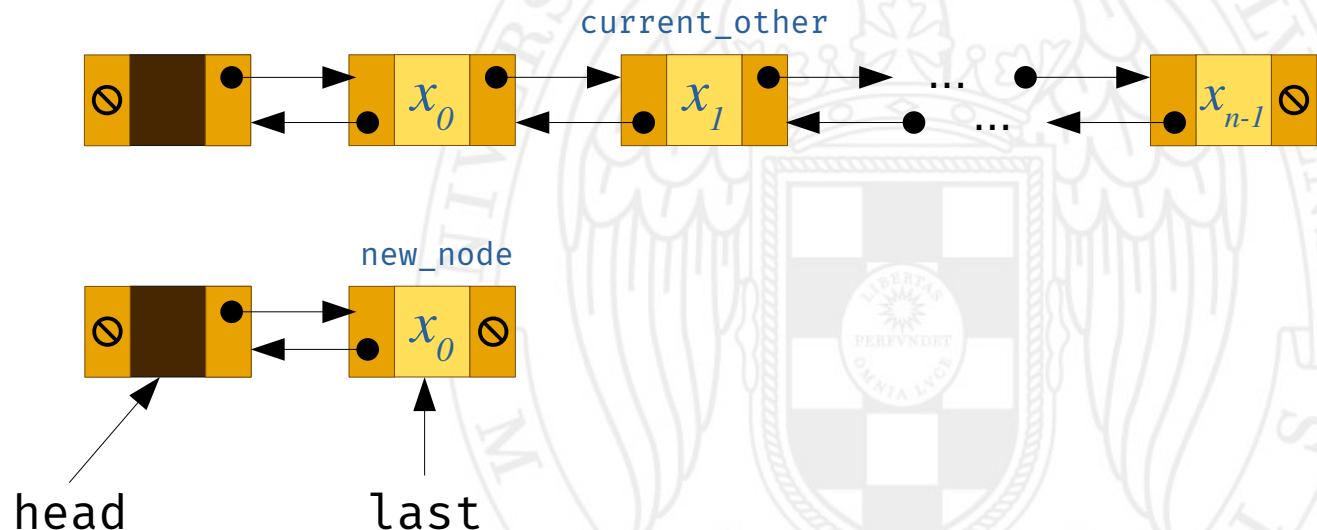
# Copia de una cadena de nodos

```
ListLinkedDouble(const ListLinkedDouble &other): ListLinkedDouble() {  
    Node *current_other = other.head->next;  
  
    while (current_other != nullptr) {  
        Node *new_node = new Node { current_other->value, nullptr, last };  
        last->next = new_node;  
        last = new_node;  
        current_other = current_other->next;  
    }  
}
```



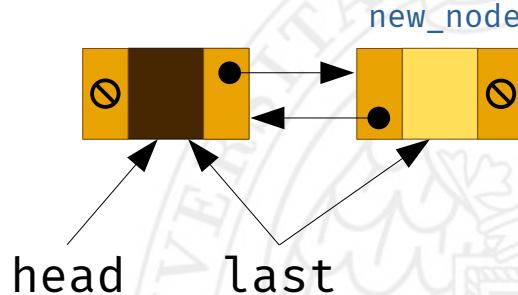
# Copia de una cadena de nodos

```
ListLinkedDouble(const ListLinkedDouble &other): ListLinkedDouble() {  
    Node *current_other = other.head->next;  
  
    while (current_other != nullptr) {  
        Node *new_node = new Node { current_other->value, nullptr, last };  
        last->next = new_node;  
        last = new_node;  
        current_other = current_other->next;  
    }  
}
```



# Añadir al principio de la lista

```
void push_front(const std::string &elem) {  
    Node *new_node = new Node { elem, head->next, head };  
    if (head->next != nullptr) {  
        head->next->prev = new_node;  
    }  
    head->next = new_node;  
    if (last == head) {  
        last = new_node;  
    }  
}
```

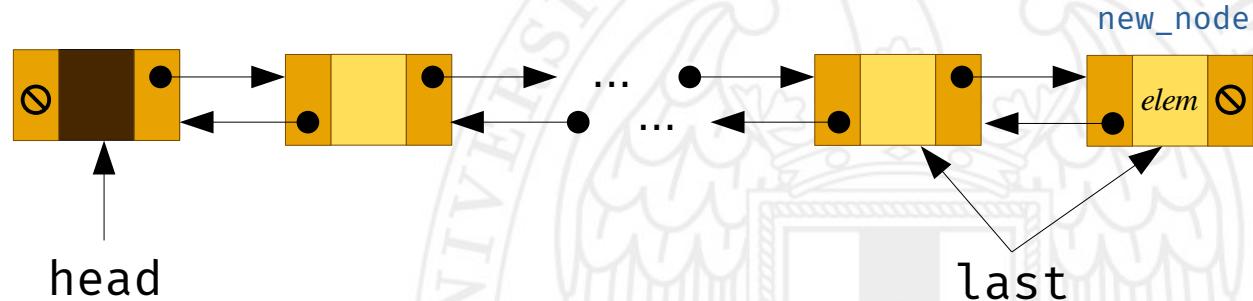


# Eliminar al principio de la lista

```
void pop_front() {
    assert (head->next != nullptr);
    Node *target = head->next;
    head->next = target->next;
    if (target->next != nullptr) {
        target->next->prev = head;
    }
    if (last == target) {
        last = head;
    }
    delete target;
}
```

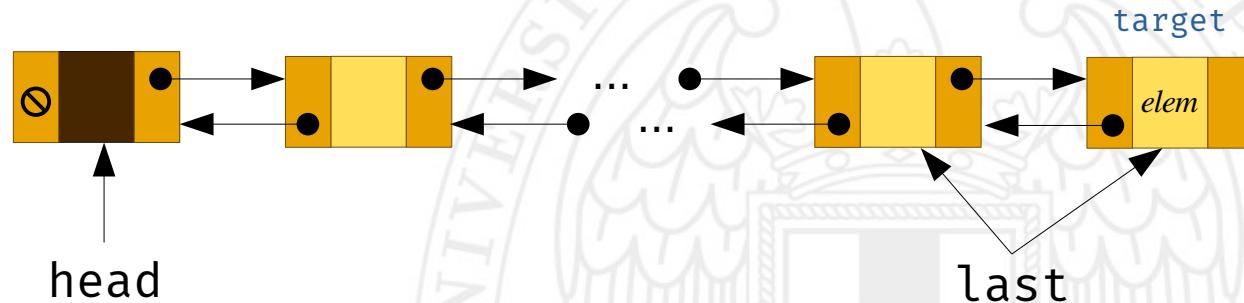
# Añadir al final de la lista

```
void push_back(const std::string &elem) {  
    Node *new_node = new Node { elem, nullptr, last };  
    last->next = new_node;  
    last = new_node;  
}
```



# Eliminar del final de la lista

```
void pop_back() {
    assert (head->next != nullptr);
    Node *target = last;
    target->prev->next = nullptr;
    last = target->prev;
    delete target;
}
```



# ¿Mejoras en el coste?

Operación	Listas enlazadas simples	Listas doblemente enlazadas
Creación	$O(1)$	$O(1)$
Copia	$O(n)$	$O(n)$
push_back	$O(n)$	$O(1)$
push_front	$O(1)$	$O(1)$
pop_back	$O(n)$	$O(1)$
pop_front	$O(1)$	$O(1)$
back	$O(n)$	$O(1)$
front	$O(1)$	$O(1)$
display	$O(n)$	$O(n)$
at(index)	$O(index)$	$O(index)$
size	$O(n)$	$O(n)$
empty	$O(1)$	$O(1)$

$n$  = número de elementos de la lista de entrada

# ¿Podemos mejorar size( )?

- Sí. Para ello añadimos un nuevo atributo num\_elems a la clase que mantenga el número de elementos en la lista.
- La función size( ) devuelve el valor de este atributo.
- Actualizamos este elemento al añadir/quitar elementos de la lista.

```
class ListLinkedDouble {  
public:  
    ...  
    int size() const { return num_elems; }  
private:  
    ...  
    Node *head, *last;  
    int num_elems;  
};
```

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

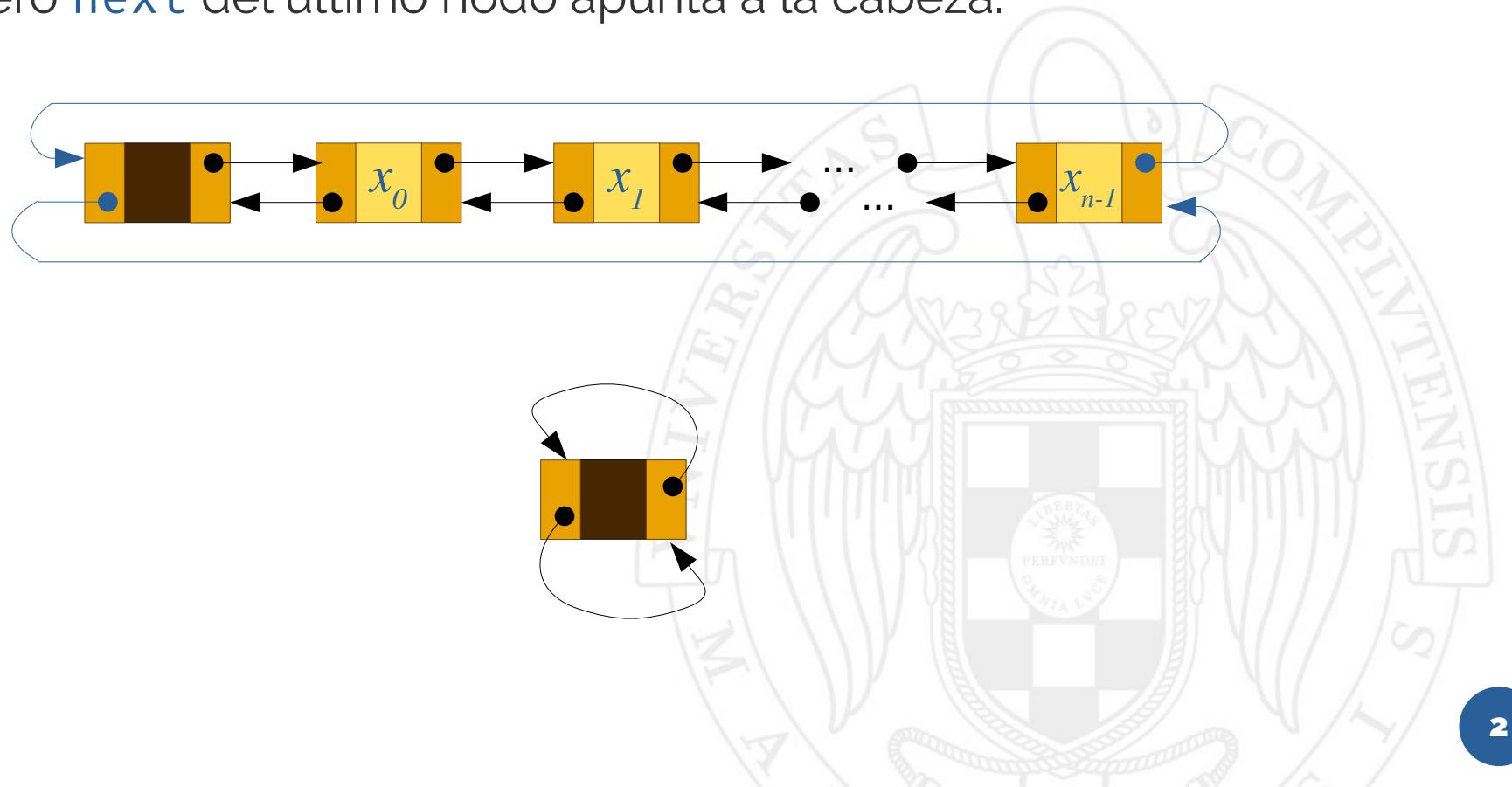
# Listas enlazadas circulares

Manuel Montenegro Montes

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

# Listas doblemente enlazadas circulares

- El puntero `prev` de la cabeza apunta al último nodo.
- El puntero `next` del último nodo apunta a la cabeza.



# Consecuencias

- No hay punteros nulos en la cadena.
- No es necesario un atributo `last` en la clase `ListLinkedDouble` que apunte al último nodo.
  - En su lugar: `head→prev`.
- Se simplifican algunas operaciones.
- ¡Cuidado al iterar sobre los nodos!

```
current = head→next;  
while (current ≠ nullptr) { !  
    ...  
    current = current→next;  
}
```



```
current = head→next;  
while (current ≠ head) { ✓  
    ...  
    current = current→next;  
}
```

# Eliminamos atributo last

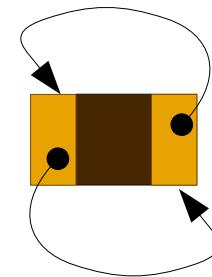
```
class ListLinkedDouble {  
public:  
    ListLinkedDouble();  
    ListLinkedDouble(const ListLinkedDouble &other);  
    ~ListLinkedDouble();  
  
    void push_front(const std::string &elem);  
    void push_back(const std::string &elem);  
    void pop_front();  
    void pop_back();  
    int size() const;  
    bool empty() const;  
    const std::string & front() const;  
    std::string & front();  
    const std::string & back() const;  
    std::string & back();  
    const std::string & at(int index) const;  
    std::string & at(int index);  
    void display() const;  
private:  
    ...  
    Node *head, *last;  
    int num_elems;  
};
```

← Eliminar ~~\*last~~



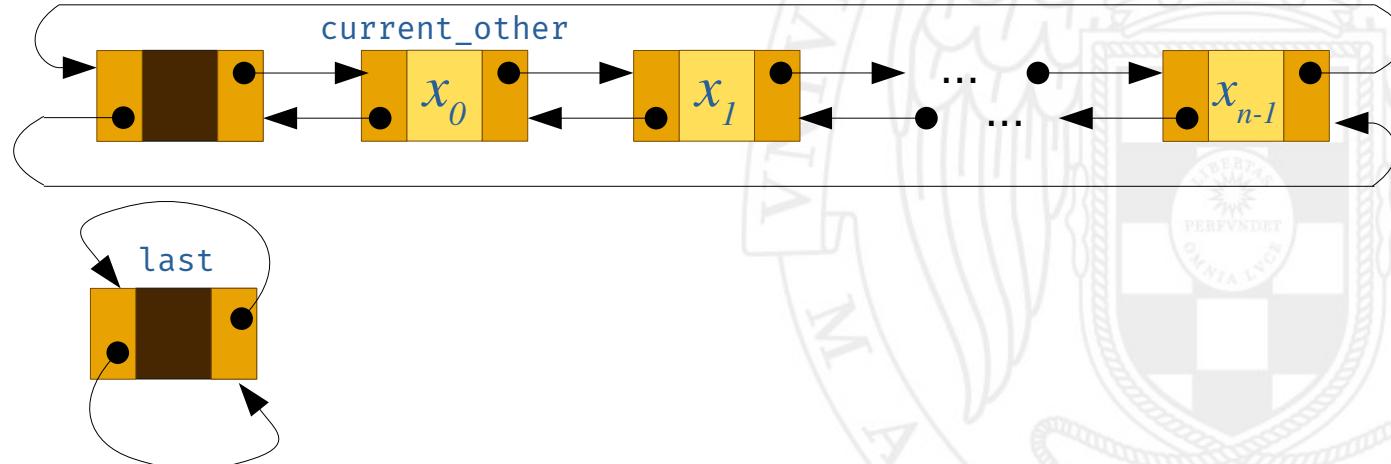
# Creación de una lista

```
ListLinkedDouble(): num_elems(0) {  
    head = new Node;  
    head→next = head;  
    head→prev = head;  
}
```



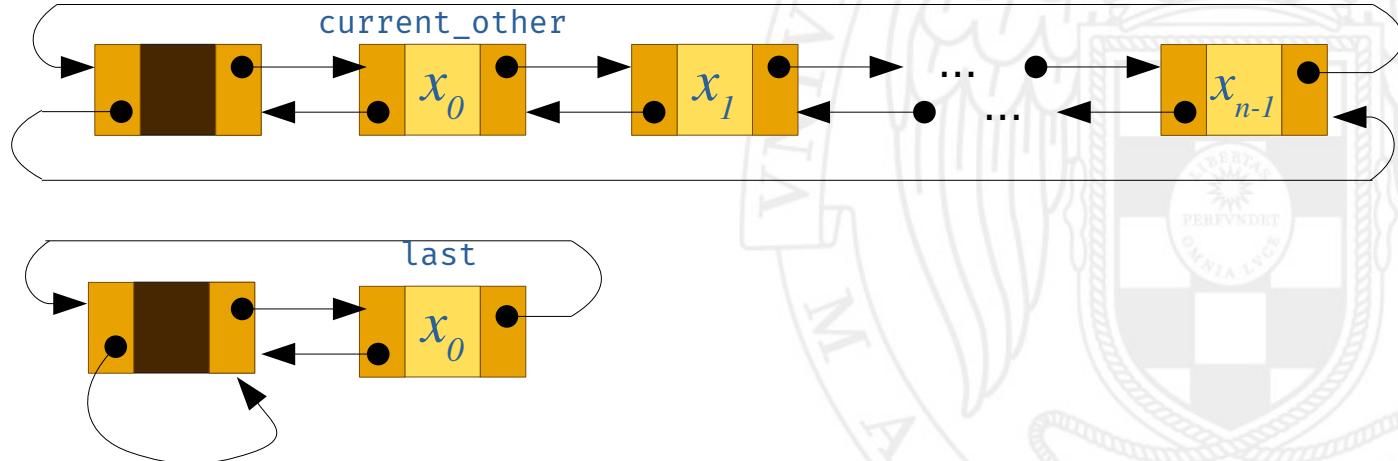
# Copia de una cadena de nodos

```
ListLinkedDouble(const ListLinkedDouble &other): ListLinkedDouble() {  
    Node *current_other = other.head->next;  
    Node *last = head;  
  
    while (current_other != other.head) {  
        Node *new_node = new Node { current_other->value, head, last };  
        last->next = new_node;  
        last = new_node;  
        current_other = current_other->next;  
    }  
    head->prev = last;  
    num_elems = other.num_elems;  
}
```



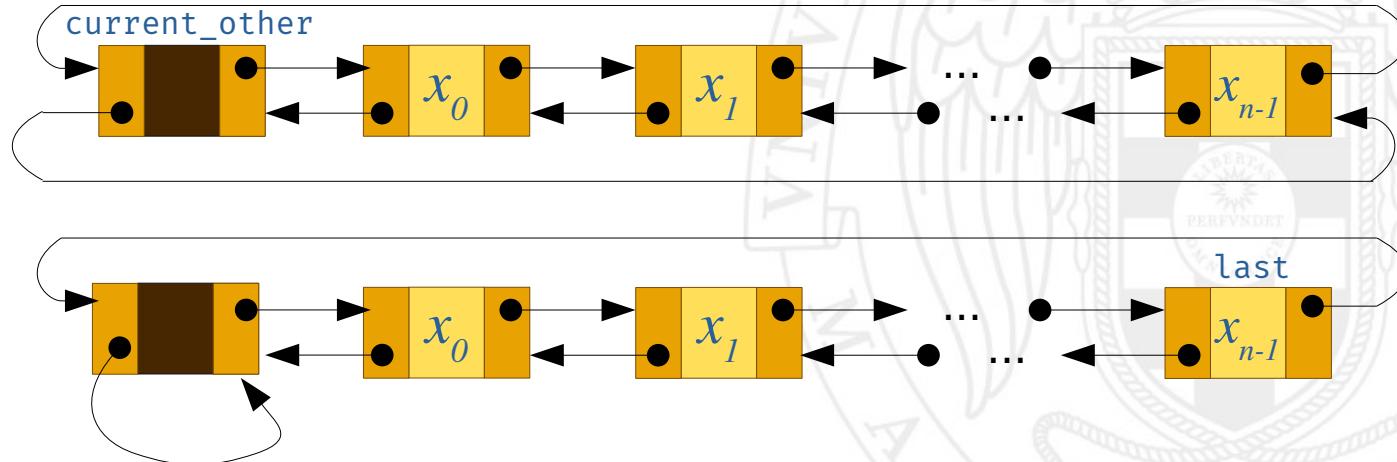
# Copia de una cadena de nodos

```
ListLinkedDouble(const ListLinkedDouble &other): ListLinkedDouble() {  
    Node *current_other = other.head->next;  
    Node *last = head;  
  
    while (current_other != other.head) {  
        Node *new_node = new Node { current_other->value, head, last };  
        last->next = new_node;  
        last = new_node;  
        current_other = current_other->next;  
    }  
    head->prev = last;  
    num_elems = other.num_elems;  
}
```



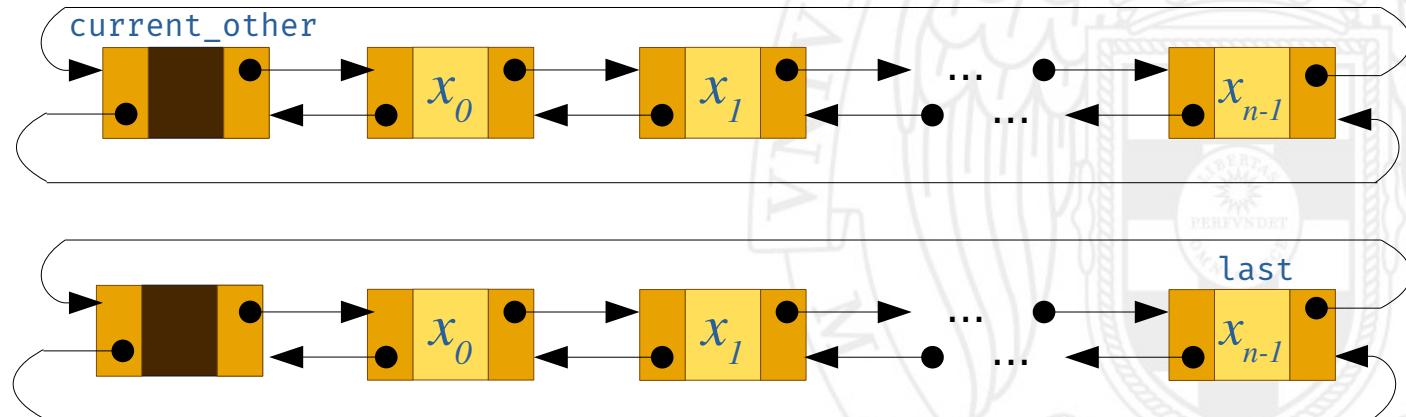
# Copia de una cadena de nodos

```
ListLinkedDouble(const ListLinkedDouble &other): ListLinkedDouble() {  
    Node *current_other = other.head->next;  
    Node *last = head;  
  
    while (current_other != other.head) {  
        Node *new_node = new Node { current_other->value, head, last };  
        last->next = new_node;  
        last = new_node;  
        current_other = current_other->next;  
    }  
    head->prev = last;  
    num_elems = other.num_elems;  
}
```



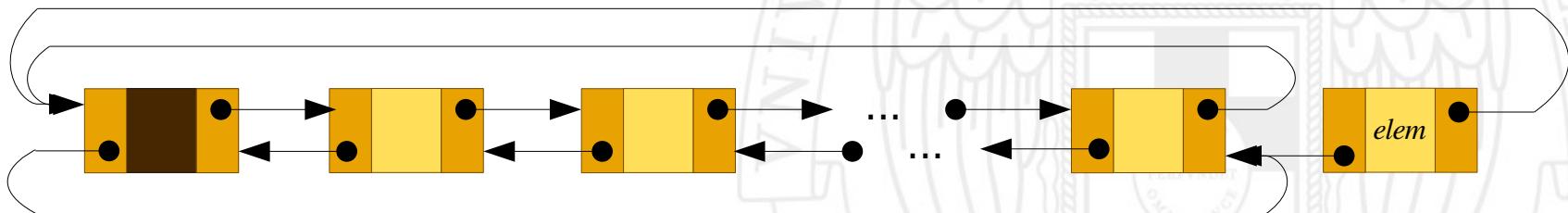
# Copia de una cadena de nodos

```
ListLinkedDouble(const ListLinkedDouble &other): ListLinkedDouble() {  
    Node *current_other = other.head->next;  
    Node *last = head;  
  
    while (current_other != other.head) {  
        Node *new_node = new Node { current_other->value, head, last };  
        last->next = new_node;  
        last = new_node;  
        current_other = current_other->next;  
    }  
    head->prev = last;  
    num_elems = other.num_elems;  
}
```



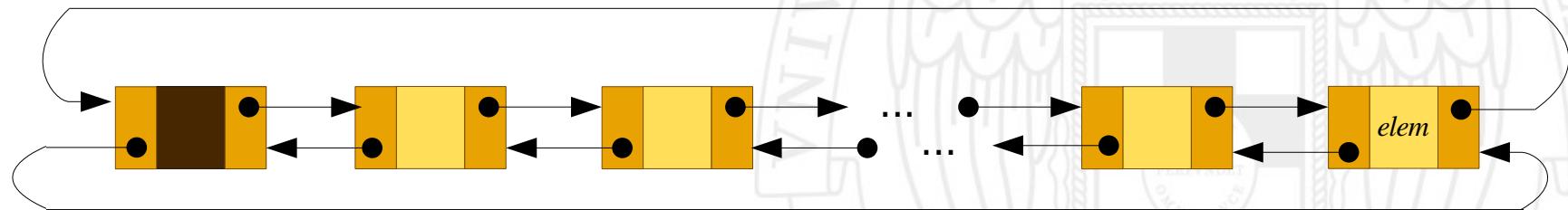
# Añadir elementos

```
void push_front(const std::string &elem) {  
    Node *new_node = new Node { elem, head->next, head };  
    head->next->prev = new_node;  
    head->next = new_node;  
    num_elems++;  
}  
  
void push_back(const std::string &elem) {  
    Node *new_node = new Node { elem, head, head->prev };  
    head->prev->next = new_node;  
    head->prev = new_node;  
    num_elems++;  
}
```



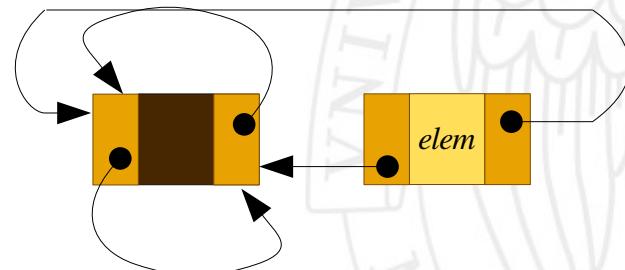
# Añadir elementos

```
void push_front(const std::string &elem) {  
    Node *new_node = new Node { elem, head->next, head };  
    head->next->prev = new_node;  
    head->next = new_node;  
    num_elems++;  
}  
  
void push_back(const std::string &elem) {  
    Node *new_node = new Node { elem, head, head->prev };  
    head->prev->next = new_node;  
    head->prev = new_node;  
    num_elems++;  
}
```



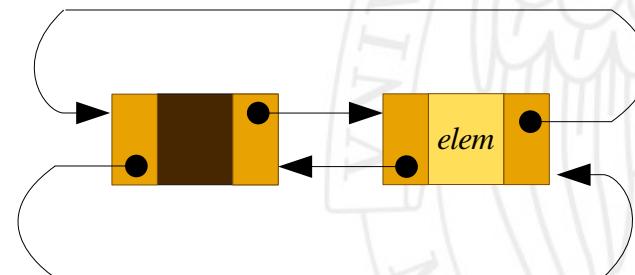
# Añadir elementos

```
void push_front(const std::string &elem) {  
    Node *new_node = new Node { elem, head->next, head };  
    head->next->prev = new_node;  
    head->next = new_node;  
    num_elems++;  
}  
  
void push_back(const std::string &elem) {  
    Node *new_node = new Node { elem, head, head->prev };  
    head->prev->next = new_node;  
    head->prev = new_node;  
    num_elems++;  
}
```



# Añadir elementos

```
void push_front(const std::string &elem) {  
    Node *new_node = new Node { elem, head->next, head };  
    head->next->prev = new_node;  
    head->next = new_node;  
    num_elems++;  
}  
  
void push_back(const std::string &elem) {  
    Node *new_node = new Node { elem, head, head->prev };  
    head->prev->next = new_node;  
    head->prev = new_node;  
    num_elems++;  
}
```



# Eliminar elementos

```
void pop_front() {
    assert (num_elems > 0);
    Node *target = head->next;
    head->next = target->next;
    target->next->prev = head;
    delete target;
    num_elems--;
}

void pop_back() {
    assert (num_elems > 0);
    Node *target = head->prev;
    target->prev->next = head;
    head->prev = target->prev;
    delete target;
    num_elems--;
}
```



# Coste de las operaciones

Operación	Coste en tiempo
Creación	$O(1)$
Copia	$O(n)$
push_back	$O(1)$
push_front	$O(1)$
pop_back	$O(1)$
pop_front	$O(1)$
back	$O(1)$
front	$O(1)$
display	$O(n)$
at(index)	$O(index)$
size	$O(1)$
empty	$O(1)$

$n$  = número de elementos de la lista de entrada

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Sobrecargando operadores en el TAD Lista

Manuel Montenegro Montes

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

# Sobrecarga del operador <<

# Generalizando el método `display()`

```
class ListArray {
public:
    void display() const;
    ...
};

void ListArray::display() const {
    std::cout << "[";
    if (num_elems > 0) {
        std::cout << elems[0];
        for (int i = 1; i < num_elems; i++) {
            std::cout << ", " << elems[i];
        }
    }
    std::cout << "]";
}
```



# Generalizando el método `display()`

```
class ListArray {  
public:  
    void display(std::ostream &out) const;  
    ...  
};  
  
void ListArray::display(std::ostream &out) const {  
    out << "[";  
    if (num_elems > 0) {  
        out << elems[0];  
        for (int i = 1; i < num_elems; i++) {  
            out << ", " << elems[i];  
        }  
    }  
    out << "]";  
}
```



# Sobrecargando el operador <<

```
class ListArray {  
public:  
    void display(std::ostream &out) const;  
    ...  
};  
  
std::ostream & operator<<(std::ostream &out, const ListArray &l) {  
    l.display(out);  
    return out;  
}
```



# Ejemplo

```
ListArray l1;  
l1.push_back("David");  
l1.push_back("Maria");  
l1.push_back("Elvira");  
  
ListArray l2 = l1;  
l2.at(1) = "Manuel";  
  
std::cout << l1 << " " << l2 << std::endl;
```

[David, Maria, Elvira] [David, Manuel, Elvira]

# Sobrecarga del operador [ ]

# Acceso a elementos de una lista

- El método `at(i)` nos permitía acceder a la posición  $i$ -ésima de una lista.

```
std::cout << l.at(1);
l.at(2) = "Francisco";
```

- Resultaría más intuitiva una notación similar a la de los arrays.

```
std::cout << l[1];
l[2] = "Francisco";
```

- Es posible habilitar esta notación sobrecargando el operador `[ ]`.
  - Para ello hay que definir un método llamado `operator[]` en la clase lista.
  - La expresión `l[i]` equivale a `l.operator[](i)`

# Sobrecarga del operador []

```
class ListArray {  
public:  
    ...  
    const std::string & at(int index) const {  
        assert (0 <= index && index < num_elems);  
        return elems[index];  
    }  
  
    std::string & at(int index) {  
        assert (0 <= index && index < num_elems);  
        return elems[index];  
    }  
  
    ...  
};
```



# Sobrecarga del operador []

```
class ListArray {  
public:  
    ...  
    const std::string & operator[](int index) const {  
        assert (0 <= index && index < num_elems);  
        return elems[index];  
    }  
  
    std::string & operator[](int index) {  
        assert (0 <= index && index < num_elems);  
        return elems[index];  
    }  
    ...  
};
```



# Ejemplo

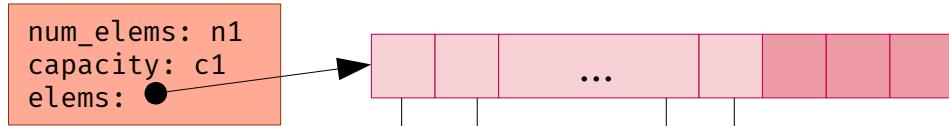
```
ListArray l;  
l.push_back("David");  
l.push_back("Maria");  
l.push_back("Elvira");  
l[2] = "Enriqueta";  
std::cout << l << std::endl;
```

[David, Maria, Enriqueta]

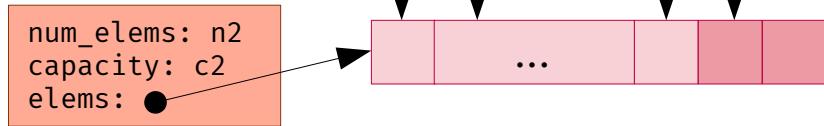
# Sobrecarga del operador de asignación

# Listas mediante arrays

`l1`



`l2`

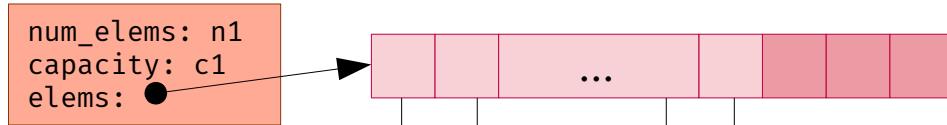


Al hacer la asignación `l2 = l1`.

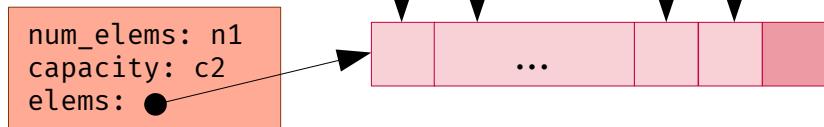
- 1) Si los elementos de `l1` caben en `l2`:
  - 1) Copiarlos de `l1` a `l2`.
  - 2) Copiar el atributo `num_elems`.

# Listas mediante arrays

`l1`



`l2`

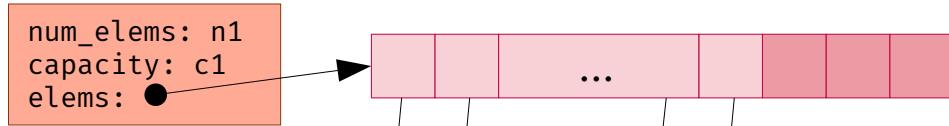


Al hacer la asignación `l2 = l1`.

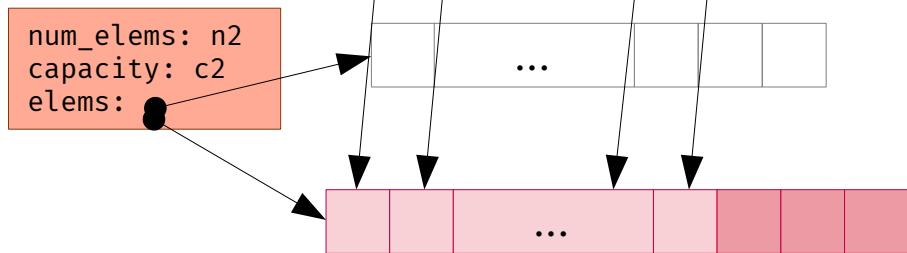
- 1) Si los elementos de `l1` caben en `l2`:
  - 1) Copiarlos de `l1` a `l2`.
  - 2) Copiar el atributo `num_elems`.

# Listas mediante arrays

`l1`



`l2`

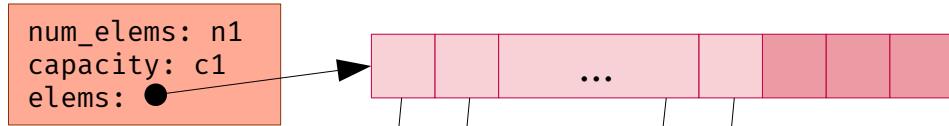


Al hacer la asignación `l2 = l1`.

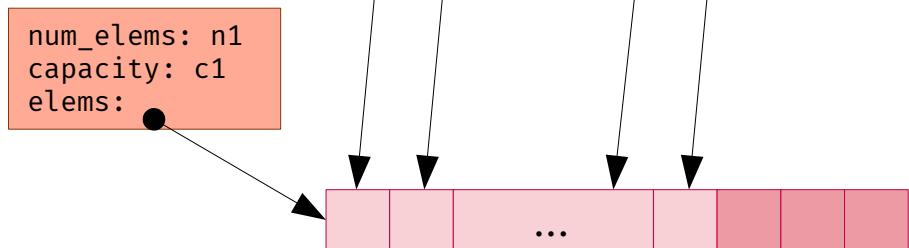
- 1) Si los elementos de `l1` caben en `l2`:
  - 1) Copiarlos de `l1` a `l2`.
  - 2) Copiar el atributo `num_elems`.
- 2) En caso contrario:
  - 1) Desechar `l2.elems` y reemplazarlo por otro array con la misma capacidad que el de `l1`.
  - 2) Copiar el atributo `capacity`.
  - 3) Copiar los elementos del array `elems` de `l1` a `l2`.
  - 4) Copiar el atributo `num_elems`.

# Listas mediante arrays

`l1`



`l2`



Al hacer la asignación `l2 = l1`.

- 1) Si los elementos de `l1` caben en `l2`:
  - 1) Copiarlos de `l1` a `l2`.
  - 2) Copiar el atributo `num_elems`.
- 2) En caso contrario:
  - 1) Desechar `l2.elems` y reemplazarlo por otro array con la misma capacidad que el de `l2`.
  - 2) Copiar el atributo `capacity`.
  - 3) Copiar los elementos del array `elems` de `l1` a `l2`.
  - 4) Copiar el atributo `num_elems`.

# Listas mediante arrays

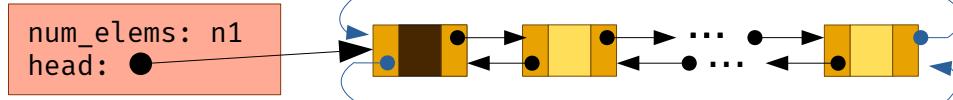
```
ListArray & operator=(const ListArray &other) {  
  
    if (this != &other) {  
        if (capacity < other.num_elems) {  
            delete[] elems;  
            elems = new std::string[other.capacity];  
            capacity = other.capacity;  
        }  
        num_elems = other.num_elems;  
        for (int i = 0; i < num_elems; i++) {  
            elems[i] = other.elems[i];  
        }  
    }  
    return *this;  
}
```

Al hacer la asignación `l2 = l1`.

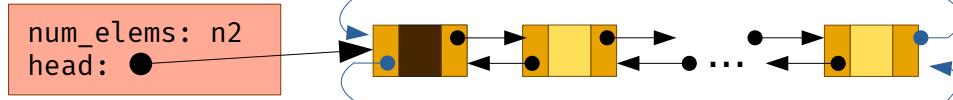
- 1) Si los elementos de `l1` caben en `l2`:
  - 1) Copiarlos de `l1` a `l2`.
  - 2) Copiar el atributo `num_elems`.
- 2) En caso contrario:
  - 1) Desechar `l2.elems` y reemplazarlo por otro array con la misma capacidad que el de `l2`.
  - 2) Copiar el atributo `capacity`.
  - 3) Copiar los elementos del array `elems` de `l1` a `l2`.
  - 4) Copiar el atributo `num_elems`.

# Listas doblemente enlazadas circulares

`l1`



`l2`

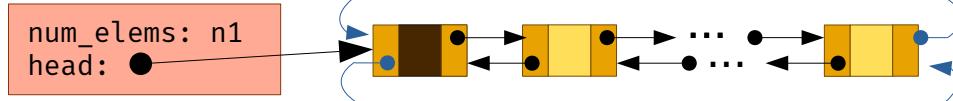


Al hacer la asignación `l2 = l1`.

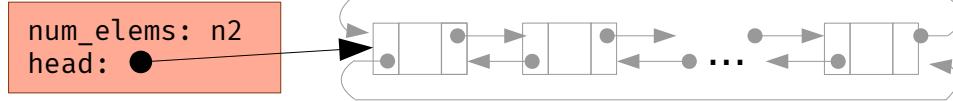
- 1) Borrar la cadena de nodos de `l2`.
- 2) Crear nodo fantasma en `l2` y hacer que `l2.head` apunte a él.
- 3) Hacer copias de los nodos de `l1` y encadenarlos en `l2` (similar al constructor de copia).
- 4) Copiar atributo `num_elems` de `l1` a `l2`.

# Listas doblemente enlazadas circulares

`l1`



`l2`

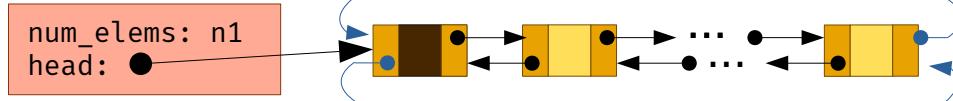


Al hacer la asignación `l2 = l1`.

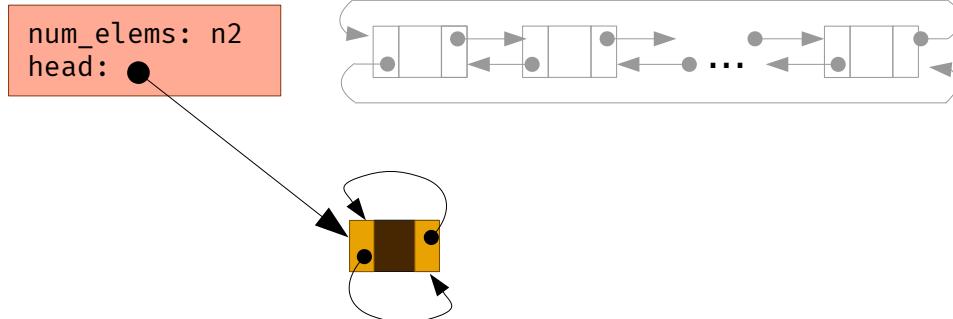
- 1) Borrar la cadena de nodos de `l2`.
- 2) Crear nodo fantasma en `l2` y hacer que `l2.head` apunte a él.
- 3) Hacer copias de los nodos de `l1` y encadenarlos en `l2` (similar al constructor de copia).
- 4) Copiar atributo `num_elems` de `l1` a `l2`.

# Listas doblemente enlazadas circulares

`l1`



`l2`



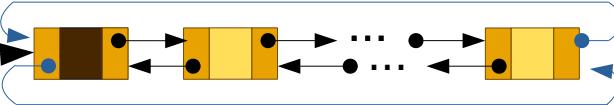
Al hacer la asignación `l2 = l1`.

- 1) Borrar la cadena de nodos de `l2`.
- 2) Crear nodo fantasma en `l2` y hacer que `l2.head` apunte a él.
- 3) Hacer copias de los nodos de `l1` y encadenarlos en `l2` (similar al constructor de copia).
- 4) Copiar atributo `num_elems` de `l1` a `l2`.

# Listas doblemente enlazadas circulares

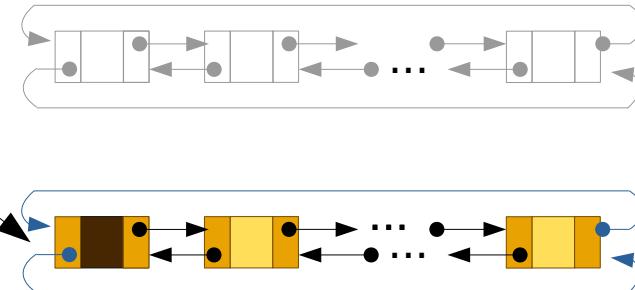
`l1`

`num_elems: n1`  
`head:` ●



`l2`

`num_elems: n2`  
`head:` ●



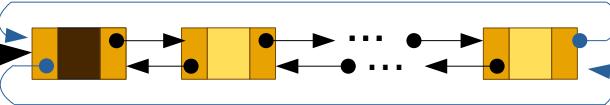
Al hacer la asignación `l2 = l1`.

- 1) Borrar la cadena de nodos de `l2`.
- 2) Crear nodo fantasma en `l2` y hacer que `l2.head` apunte a él.
- 3) Hacer copias de los nodos de `l1` y encadenarlos en `l2` (similar al constructor de copia).
- 4) Copiar atributo `num_elems` de `l1` a `l2`.

# Listas doblemente enlazadas circulares

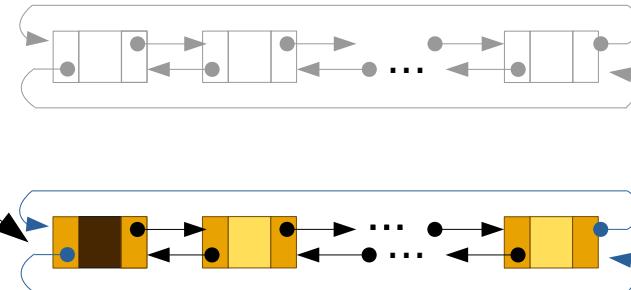
`l1`

`num_elems: n1`  
`head:` ●



`l2`

`num_elems: n1`  
`head:` ●



Al hacer la asignación `l2 = l1`.

- 1) Borrar la cadena de nodos de `l2`.
- 2) Crear nodo fantasma en `l2` y hacer que `l2.head` apunte a él.
- 3) Hacer copias de los nodos de `l1` y encadenarlos en `l2` (similar al constructor de copia).
- 4) Copiar atributo `num_elems` de `l1` a `l2`.

# Listas doblemente enlazadas circulares

```
ListLinkedDouble &  
operator=(const ListLinkedDouble &other) {  
  
    if (this != &other) {  
        delete_nodes();  
        head = new Node;  
        head->next = head->prev = head;  
        copy_nodes_from(other);  
        num_elems = other.num_elems;  
    }  
    return *this;  
}
```

Al hacer la asignación `l2 = l1`.

- 1) Borrar la cadena de nodos de `l2`.
- 2) Crear nodo fantasma en `l2` y hacer que `l2.head` apunte a él.
- 3) Hacer copias de los nodos de `l1` y encadenarlos en `l2` (similar al constructor de copia).
- 4) Copiar atributo `num_elems` de `l1` a `l2`.

ESTRUCTURAS DE DATOS

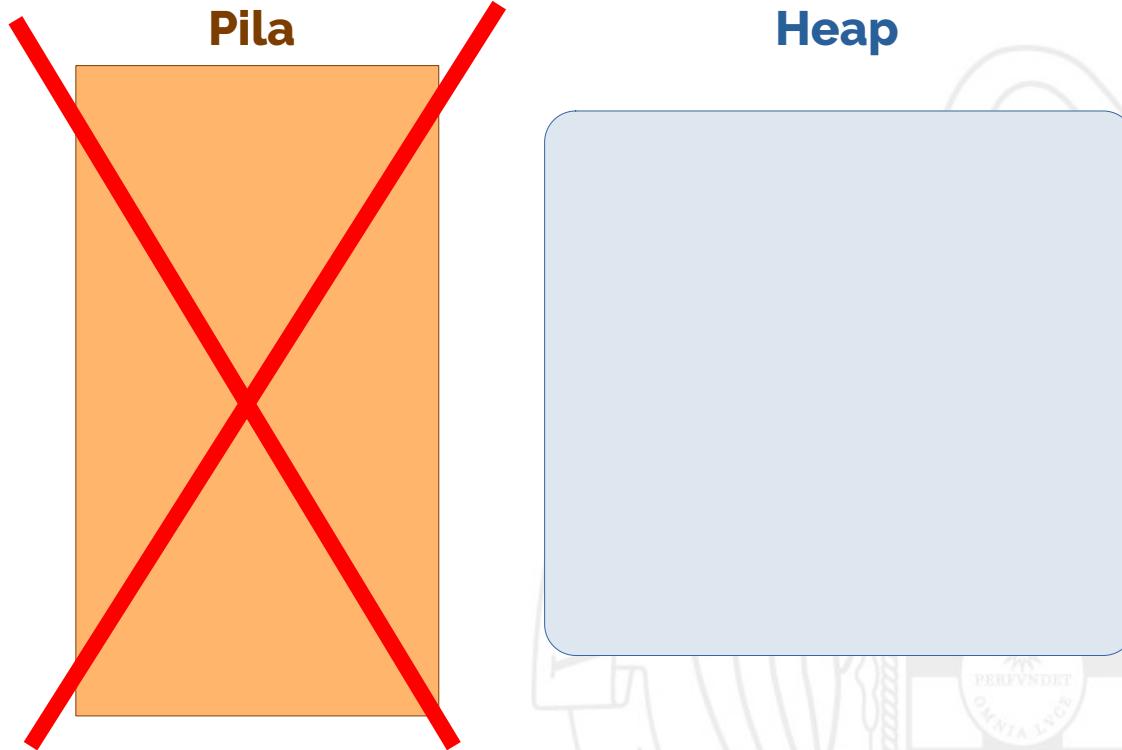
TIPOS ABSTRACTOS DE DATOS LINEALES

# EL TAD Pila

Manuel Montenegro Montes

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

# ¿Qué es una pila?

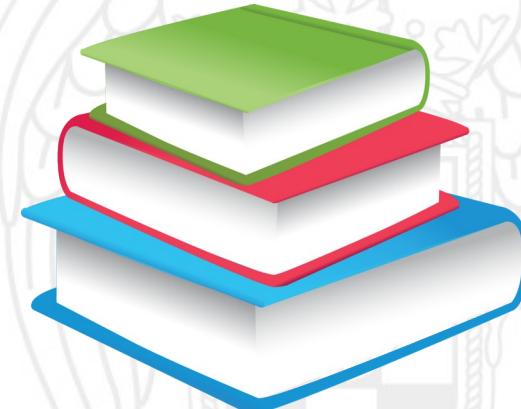


# ¿Qué es una pila?

- Es una colección de elementos que permite:
  - Insertar elementos.
  - Obtener o borrar el último elemento insertado que no haya sido borrado previamente.



Foto: John Leffmann (CC BY 3.0)



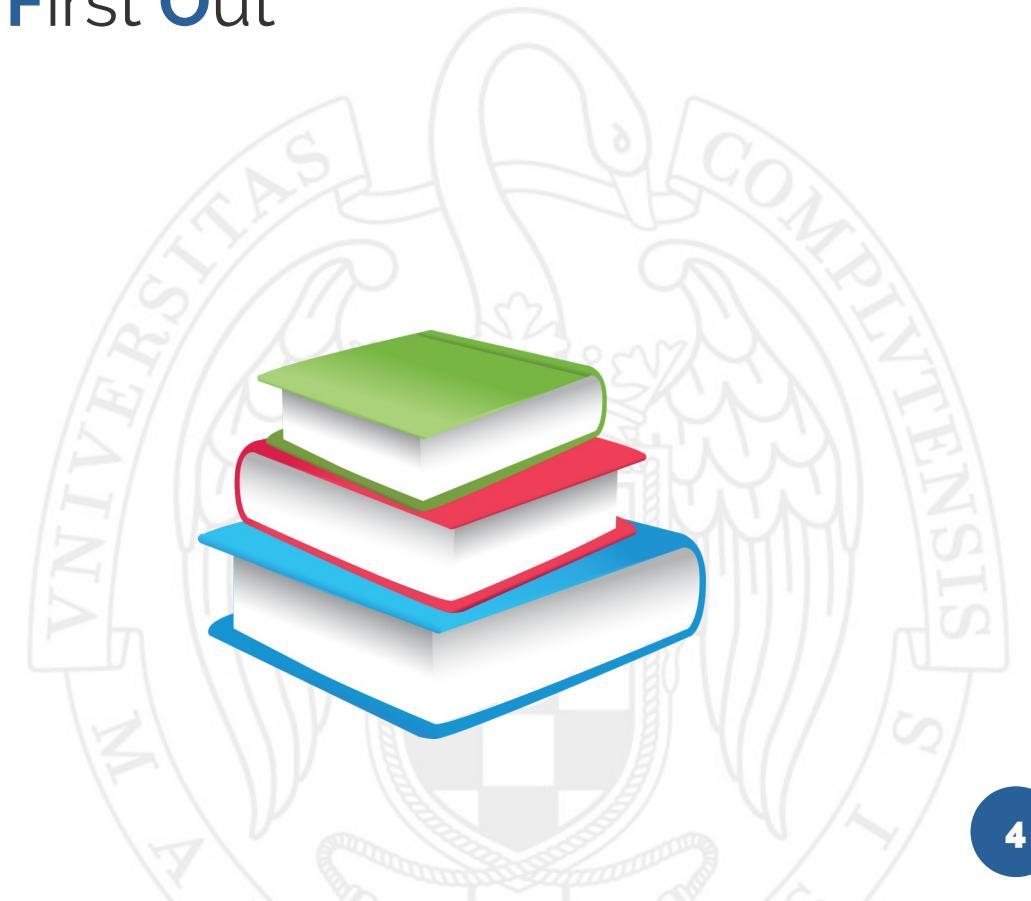
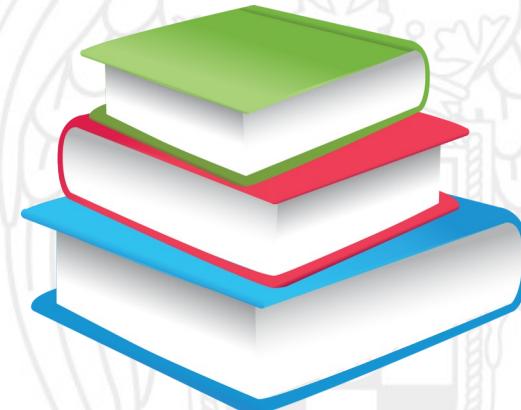
# ¿Qué es una pila?

- Las pilas reciben el nombre de estructuras de acceso **LIFO**

**Last In, First Out**

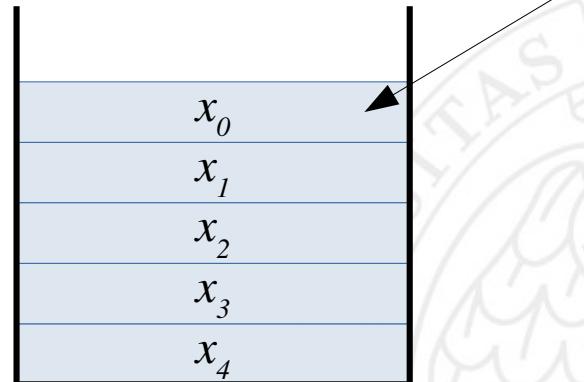


Foto: John Leffmann (CC BY 3.0)



# Modelo de pilas

- Conceptualmente representamos las pilas de esta forma:



**Cima** de la pila:  
último elemento  
insertado.

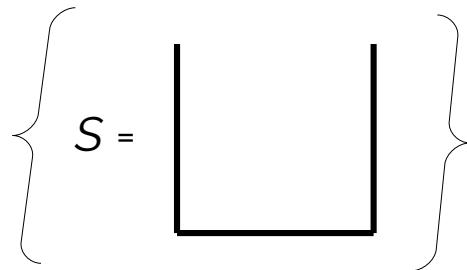
# Operaciones sobre pilas

- **Constructoras:**
  - Crear una pila vacía (***create\_empty***).
- **Mutadoras:**
  - Añadir elemento en la cima de la pila (***push***).
  - Eliminar elemento en la cima de la pila (***pop***).
- **Observadoras:**
  - Obtener el elemento en la cima de la pila (***top***).
  - Saber si una pila está vacía (***empty***).

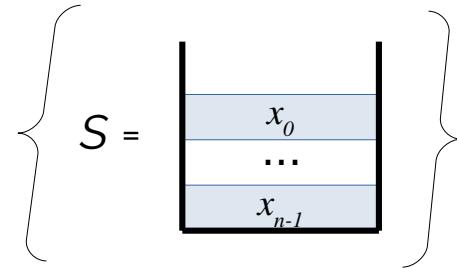
# Operación *create\_empty*

{ true }

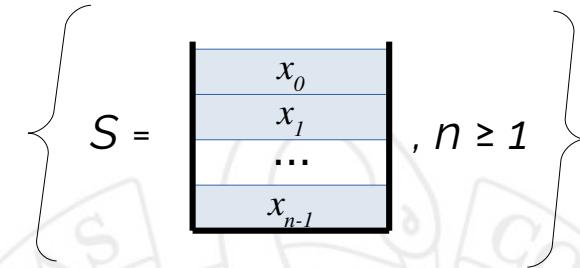
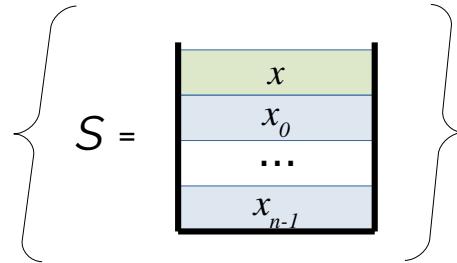
***create\_empty()*** → (S: Stack)



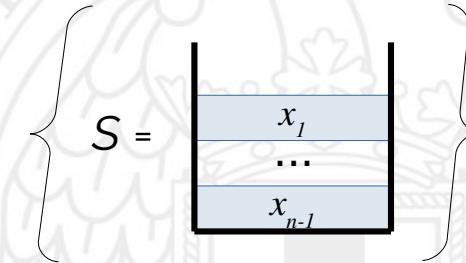
# Operaciones *push* y *pop*



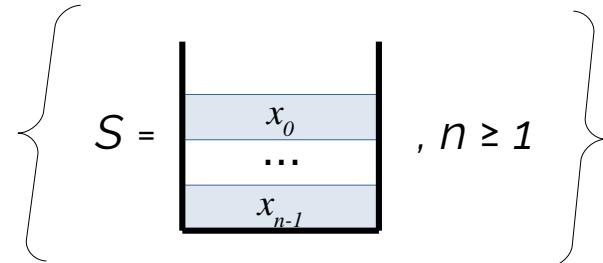
***push*( $S$ : Stack,  $x$ : elem)**



***pop*( $S$ : Stack)**

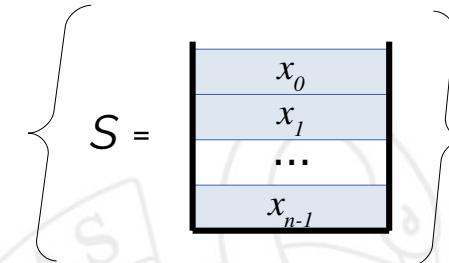


# Operaciones *top* y *empty*



***top***( $S$ : Stack)  $\rightarrow$  ( $x$ : elem)

$$\{ x = x_0 \}$$



***empty***( $S$ : Stack)  $\rightarrow$  ( $b$ : bool)

$$\{ b \Leftrightarrow n = 0 \}$$

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Implementando el TAD Pila

Manuel Montenegro Montes

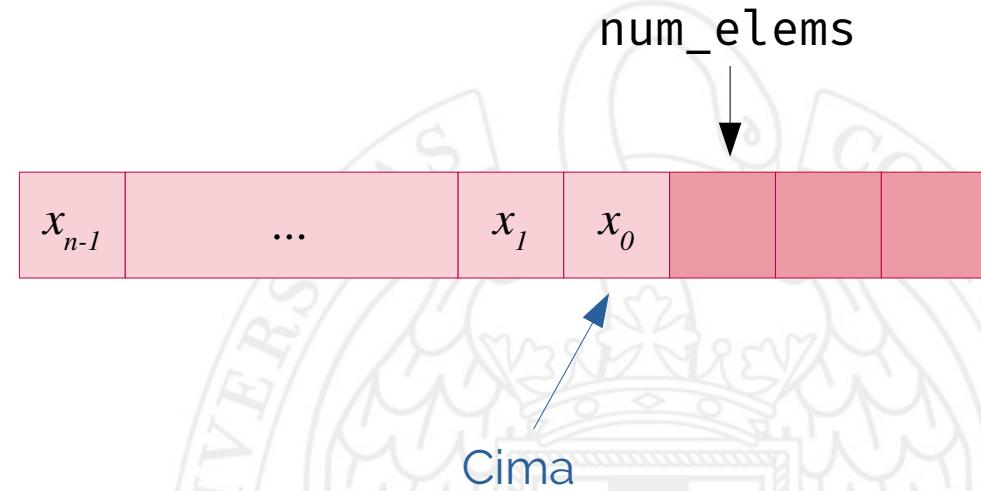
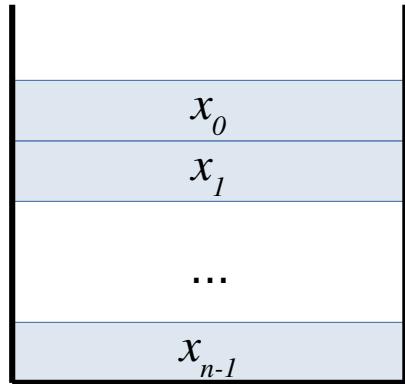
Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Operaciones sobre pilas

- **Constructoras:**
  - Crear una pila vacía (***create\_empty***).
- **Mutadoras:**
  - Añadir elemento en la cima de la pila (***push***).
  - Eliminar elemento en la cima de la pila (***pop***).
- **Observadoras:**
  - Obtener el elemento en la cima de la pila (***top***).
  - Saber si una pila está vacía (***empty***).

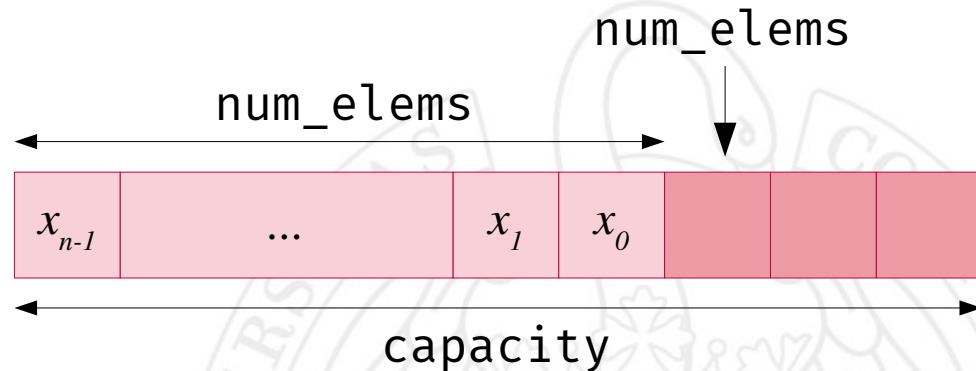
# Implementación mediante vectores

# Implementación mediante vectores



# Implementación mediante vectores

```
template<typename T>
class StackArray {
public:
    ...
private:
    int num_elems;
    int capacity;
    T *elems;
};
```



# Interfaz pública de StackArray

```
template<typename T>
class StackArray {
public:
    StackArray(int initial_capacity = DEFAULT_CAPACITY);
    StackArray(const StackArray &other);
    ~StackArray();

    StackArray & operator=(const StackArray<T> &other);

    void push(const T &elem);
    void pop();
    const T & top() const;
    T & top();
    bool empty() const;

private:
    ...
};
```



# Interfaz pública de StackArray

```
template<typename T>
class StackArray {
public:
    StackArray(int initial_capacity = DEFAULT_CAPACITY);
    StackArray(const StackArray &other);
    ~StackArray();

    StackArray & operator=(const StackArray<T> &other);

    void push(const T &elem);
    void pop();
    const T & top() const;
    T & top();
    bool empty() const;

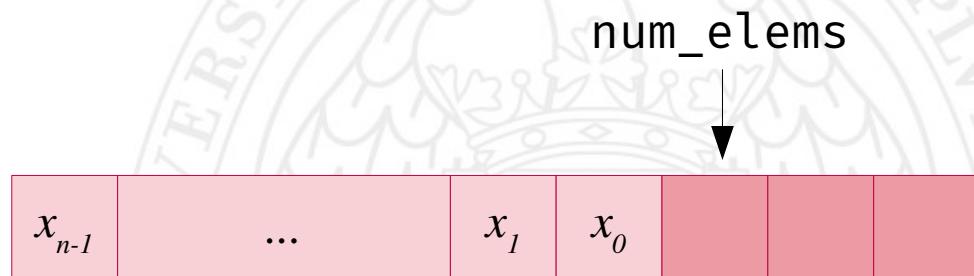
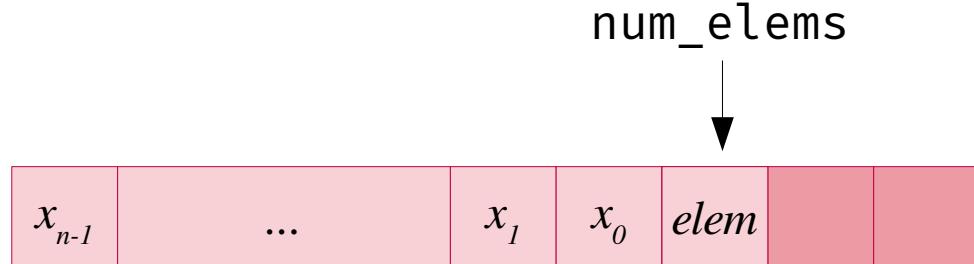
private:
    ...
};
```



# Métodos push() y pop()

```
void push(const T &elem) {  
    if (num_elems == capacity) {  
        resize_array(capacity * 2);  
    }  
    elems[num_elems] = elem;  
    num_elems++;  
}
```

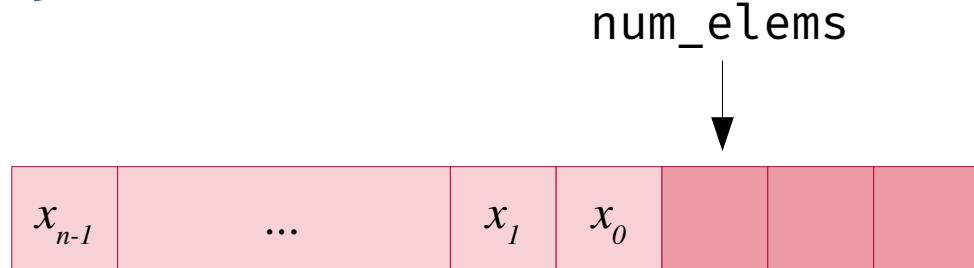
```
void pop() {  
    assert(num_elems > 0);  
    num_elems--;  
}
```



# Métodos `top()` y `empty()`

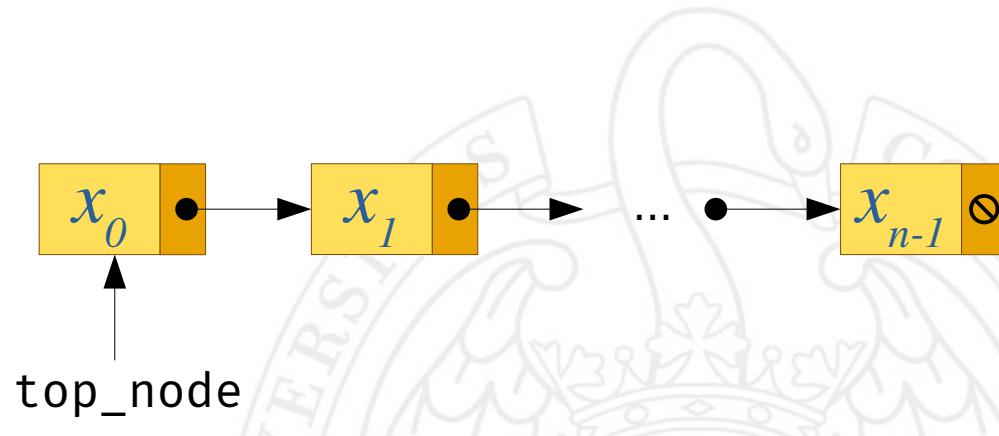
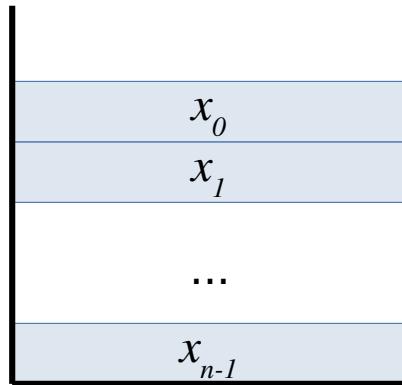
```
const T & top() const {  
    assert(num_elems > 0);  
    return elems[num_elems - 1];  
}
```

```
bool empty() const {  
    return num_elems == 0;  
}
```



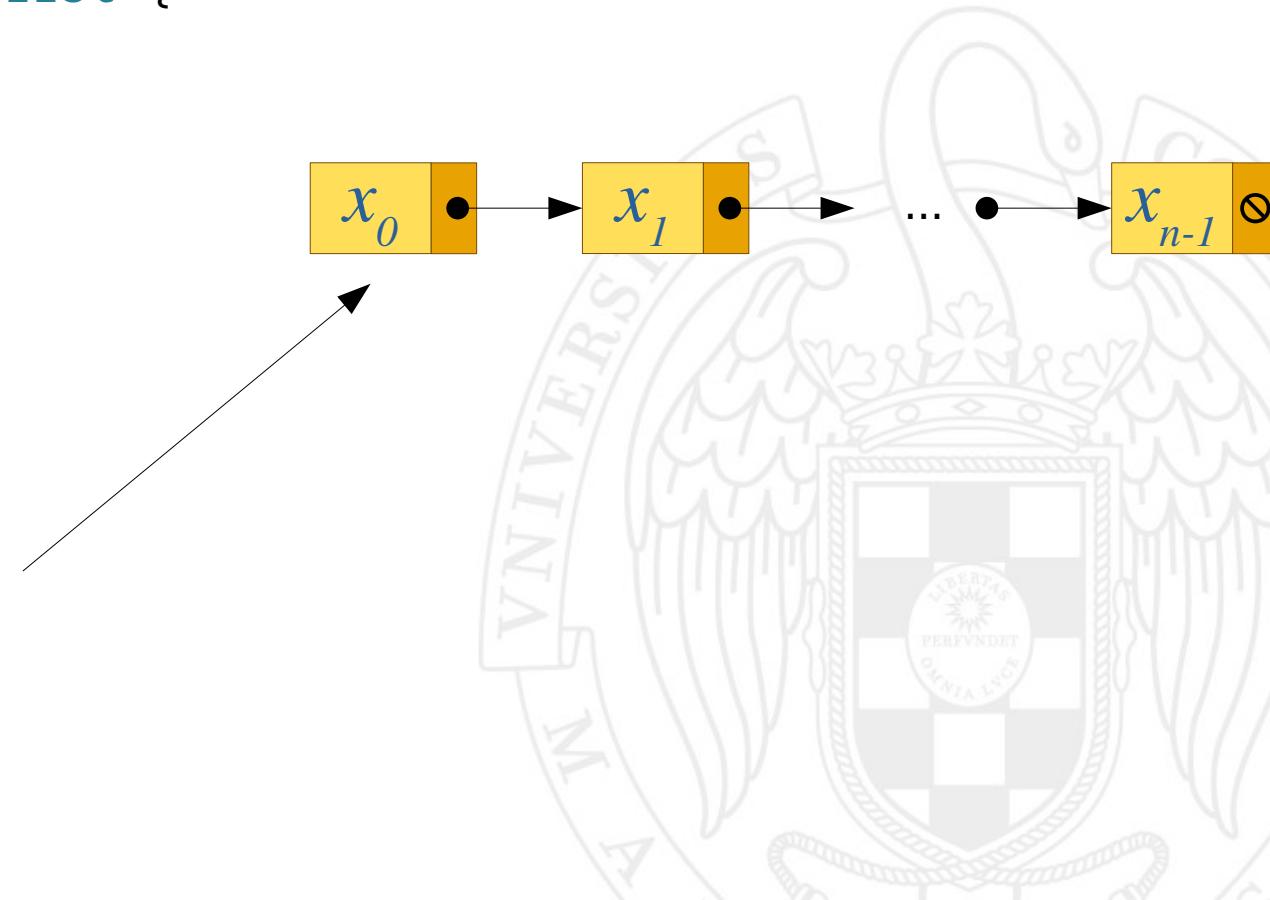
# Implementación mediante listas enlazadas simples

# Implementación mediante listas enlazadas



# Implementación mediante listas enlazadas

```
template<typename T>
class StackLinkedList {
public:
    ...
private:
    struct Node {
        T value;
        Node *next;
    };
    Node *top_node;
};
```



# Interfaz pública de StackLinkedList

```
template<typename T>
class StackLinkedList {
public:
    StackLinkedList();
    StackLinkedList(const StackLinkedList &other);
    ~StackLinkedList();

    StackLinkedList & operator=(const StackLinkedList<T> &other);

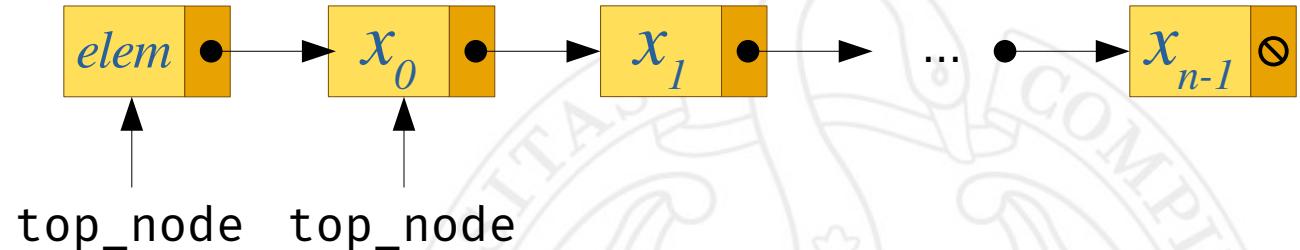
    void push(const T &elem);
    void pop();
    const T & top() const;
    T & top();
    bool empty() const;

private:
    ...
};
```

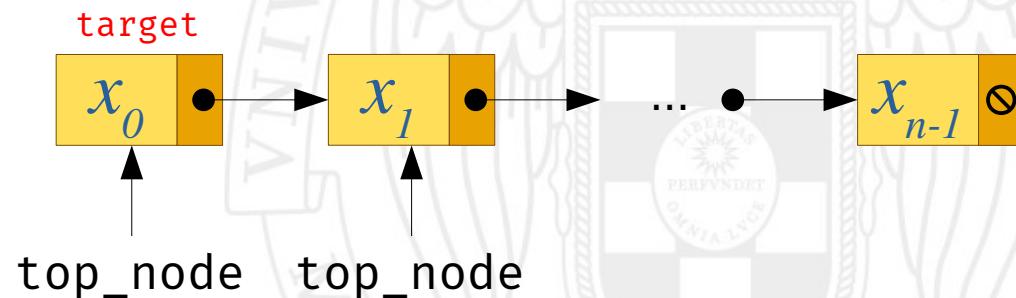


# Operaciones push() y pop()

```
void push(const T &elem) {  
    top_node = new Node{ elem, top_node };  
}
```



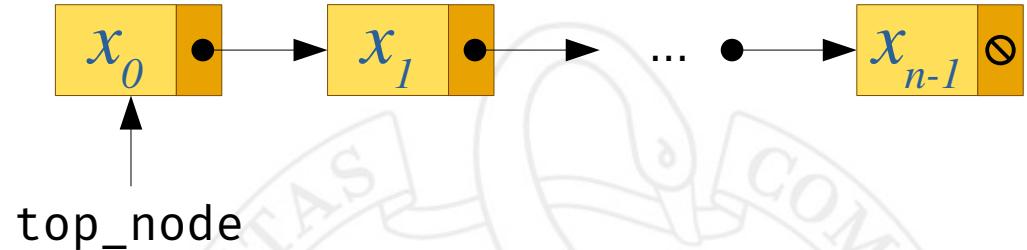
```
void pop() {  
    assert (top_node != nullptr);  
    Node *target = top_node;  
    top_node = top_node->next;  
    delete target;  
}
```



# Operaciones top() y empty()

```
const T & top() const {
    assert (top_node != nullptr);
    return top_node->value;
}

bool empty() const {
    return (top_node == nullptr);
}
```



# Coste de las operaciones

Operación	Vectores	Listas enlazadas
push	$O(n) / O(1)$	$O(1)$
pop	$O(1)$	$O(1)$
top	$O(1)$	$O(1)$
empty	$O(1)$	$O(1)$

$n$  = número de elementos en la pila

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Aplicaciones de pilas

Manuel Montenegro Montes

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

# Expresiones en forma postfija

# Expresiones en forma postfija

- Expresión en forma infija:

$$(3 * (5 + 2)) - 6$$

- La misma expresión en forma postfija:

$$(3 (5 2 +) *) 6 -$$

$$3 5 2 + * 6 -$$

- **Objetivo:** evaluar expresión en forma postfija.

- Por ejemplo,  $3 5 2 + * 6 -$  se evalúa al valor 15.

# Expresiones en forma postfija

- Las expresiones en forma postfija no contienen ambigüedades, no necesitan paréntesis o reglas de precedencia entre operadores, al contrario que las expresiones en forma infija.

2 + 4 \* 5

# Otros ejemplos

- 2 3 + 6 + 1 +
- 3 1 - 6 5 \* +



# Evaluando las expresiones mediante una pila

- Comenzamos con una pila vacía.
- Recorremos de izquierda a derecha los caracteres de la expresión.
- Si el carácter actual es un número:
  - Insertar el número en la pila.
- Si el carácter actual es un operador:
  - Desapilar los dos operandos y realizar la operación con ellos.
  - Apilar el resultado.
- Al finalizar el recorrido, el elemento que quede en la pila es el resultado de evaluar la expresión.

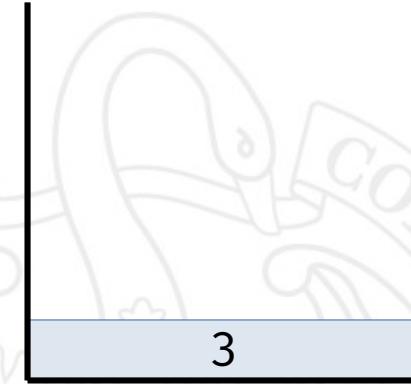
# Ejemplo de ejecución

3 5 2 + \* 6 -  
↑



# Ejemplo de ejecución

3 5 2 + \* 6 -  
↑



# Ejemplo de ejecución

3 5 2 + \* 6 -  
↑

5
3

# Ejemplo de ejecución

3 5 2 + \* 6 -



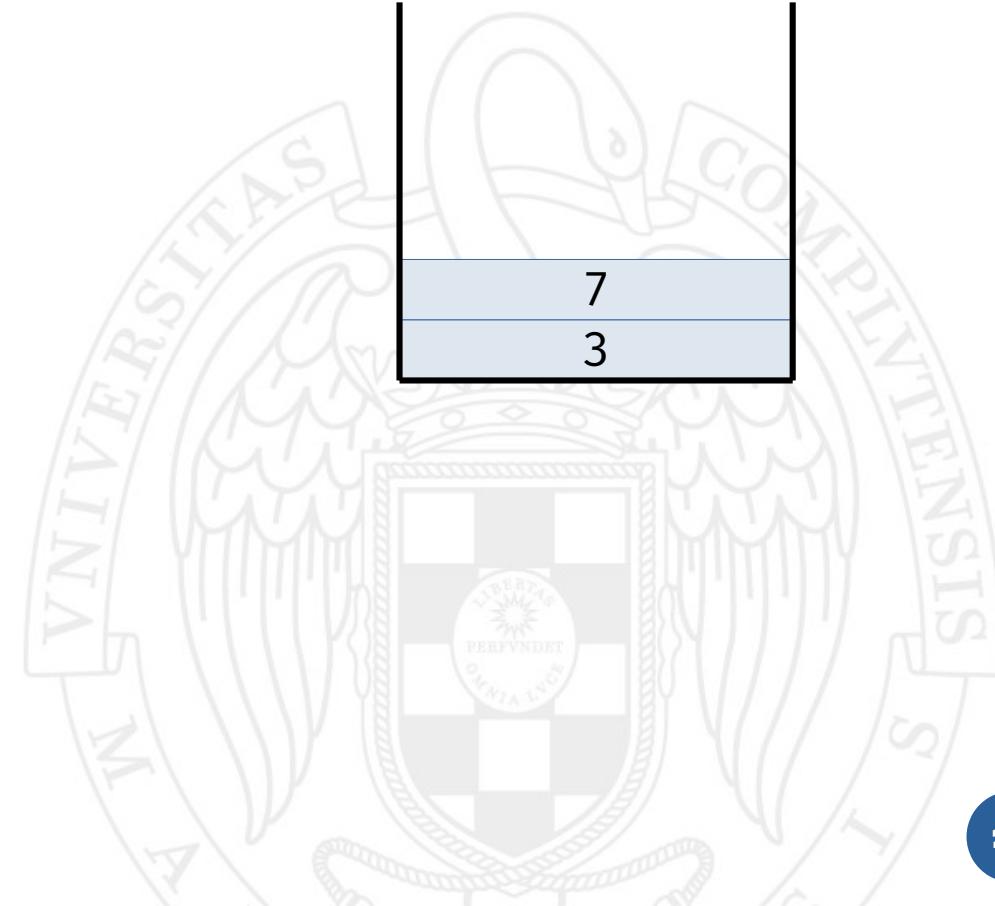
2
5
3

# Ejemplo de ejecución

3 5 2 + \* 6 -



7
3



# Ejemplo de ejecución

3 5 2 + \* 6 -  
      ↑

21

# Ejemplo de ejecución

3 5 2 + \* 6 -

6
21

# Ejemplo de ejecución

3 5 2 + \* 6 -

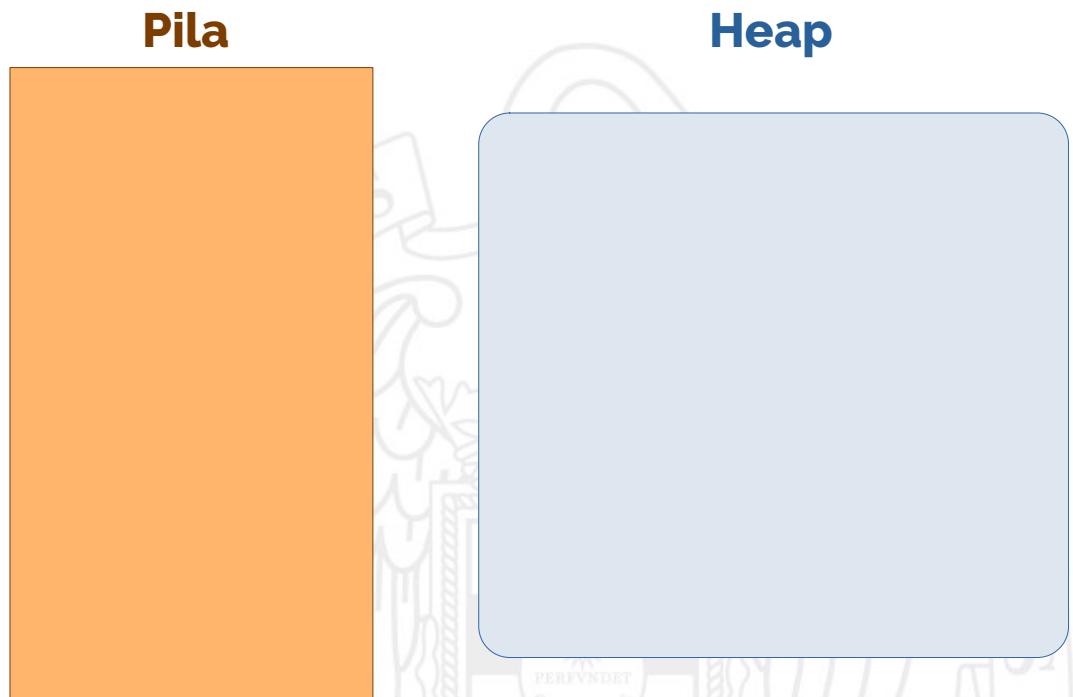


15

# Pila de llamadas a funciones

# Ejemplo de ejecución

- La región de memoria en la que se almacenan las variables locales y parámetros de un programa funciona como una pila.



# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

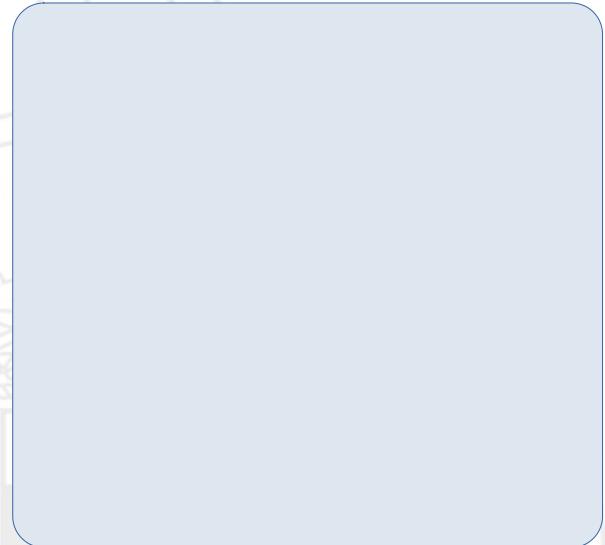
```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

Pila



Heap



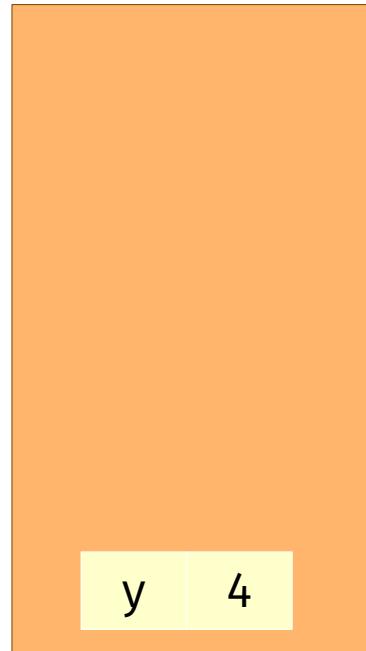
# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

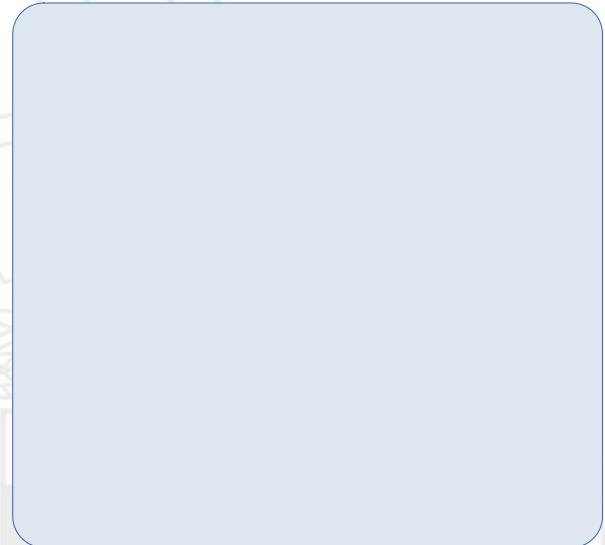
```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

Pila



Heap



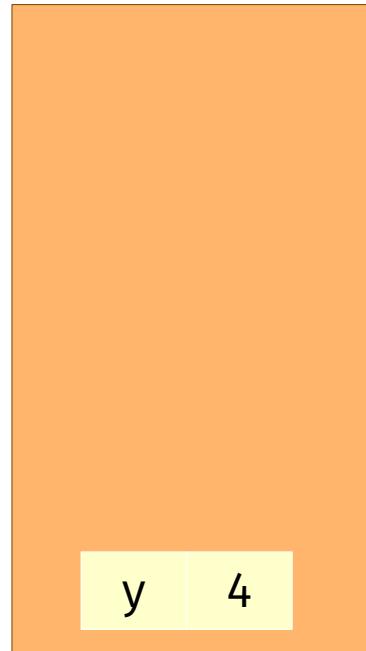
# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

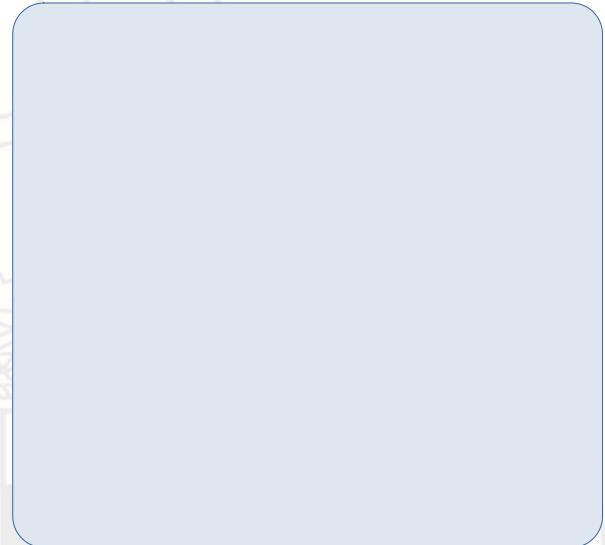
```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

Pila



Heap

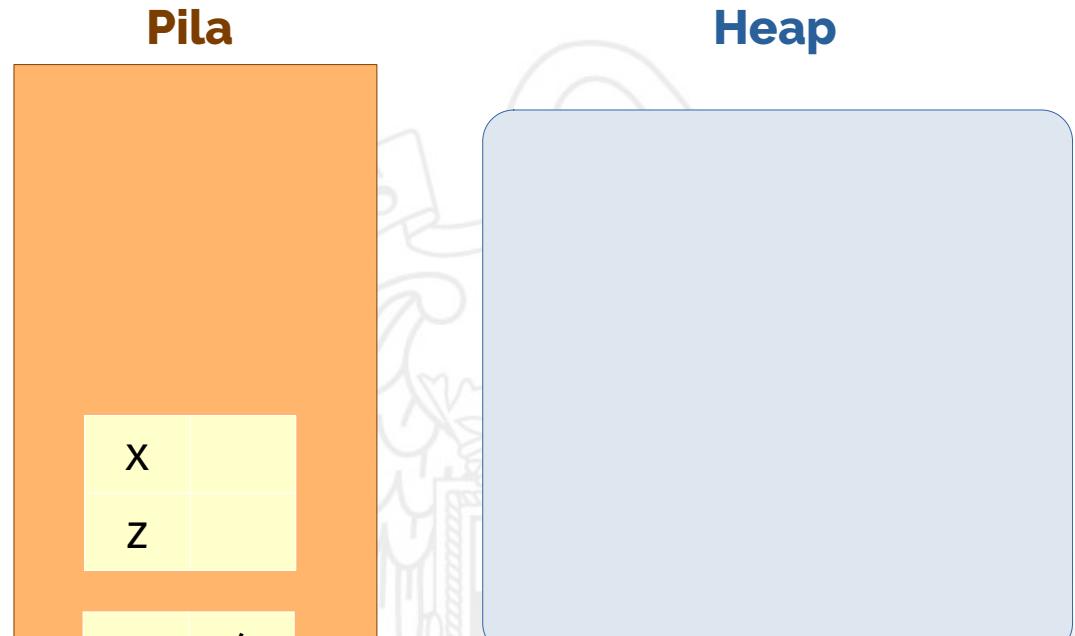


# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```



# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

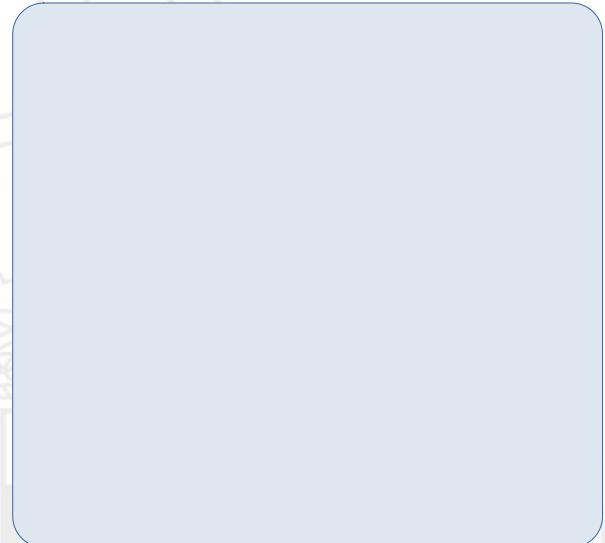
```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

Pila

x	10
z	
y	4

Heap



# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

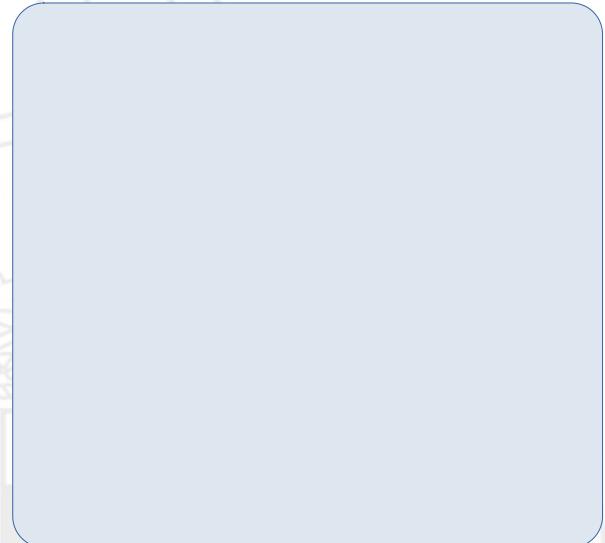
```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

Pila

x	10
z	20
y	4

Heap



# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

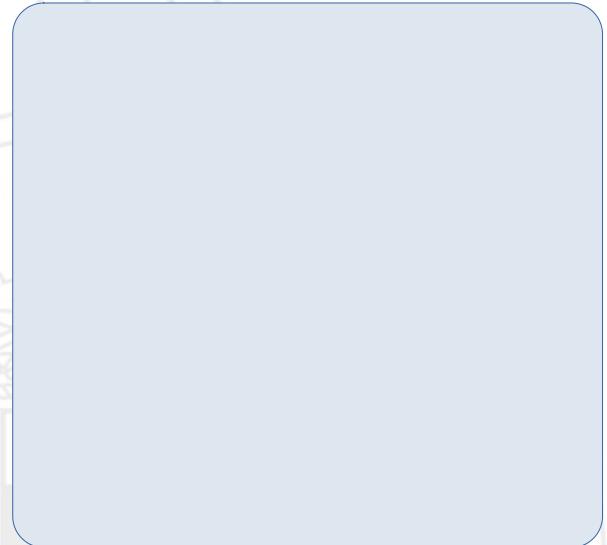
```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

Pila

x	10
z	20
y	4

Heap



# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

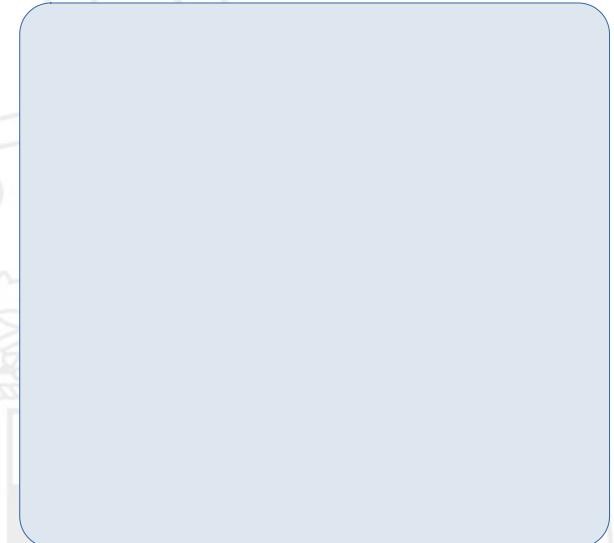
Pila

x	30
y	

x	10
z	20

y	4
---	---

Heap



# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

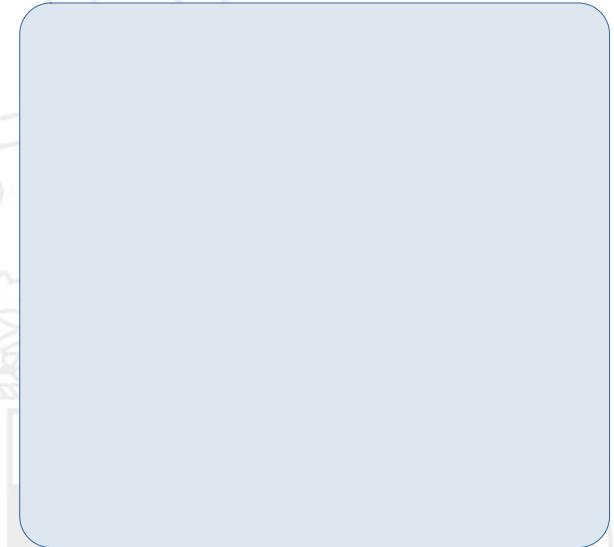
```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

Pila

x	30
y	40
x	10
z	20
y	4

Heap



# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

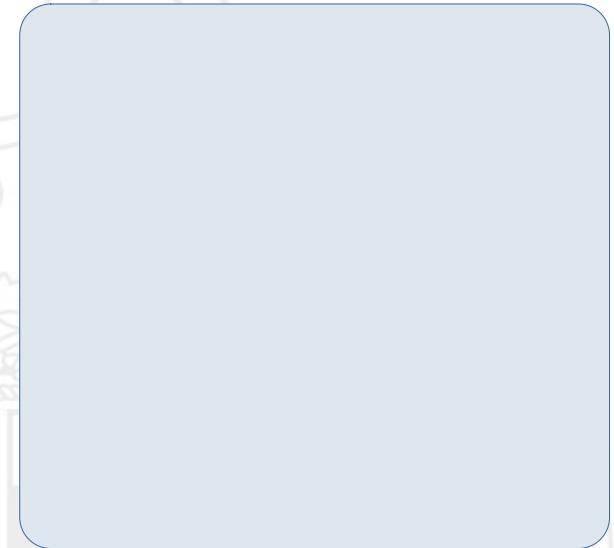
Pila

x	30
y	40

x	10
z	20

y	4
---	---

Heap



# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

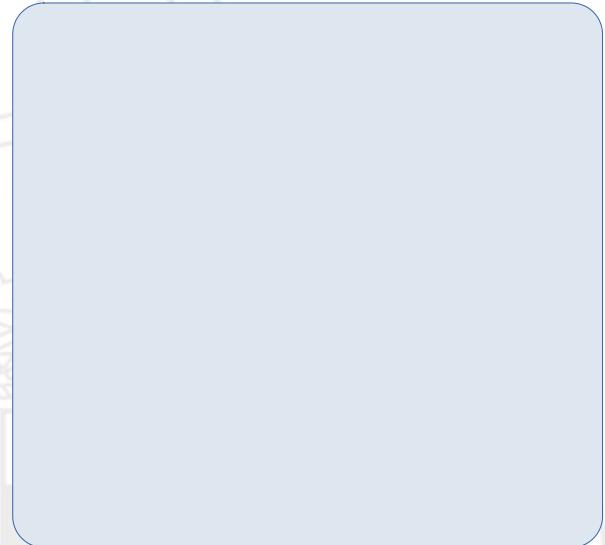
```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

Pila

x	50
z	20
y	4

Heap



# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

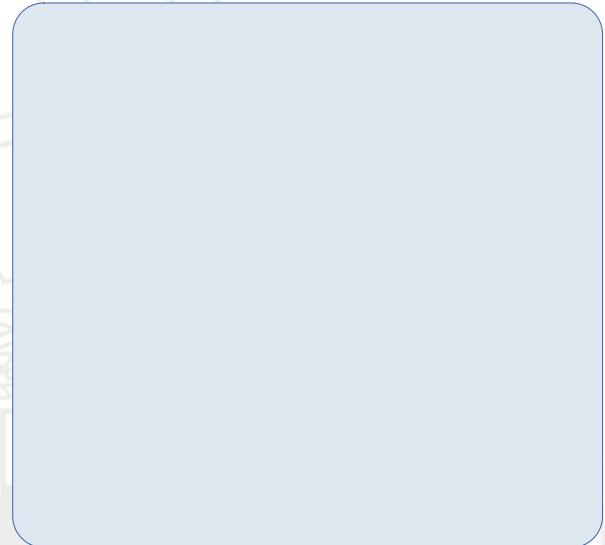
```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

Pila

x	50
z	20
y	4

Heap



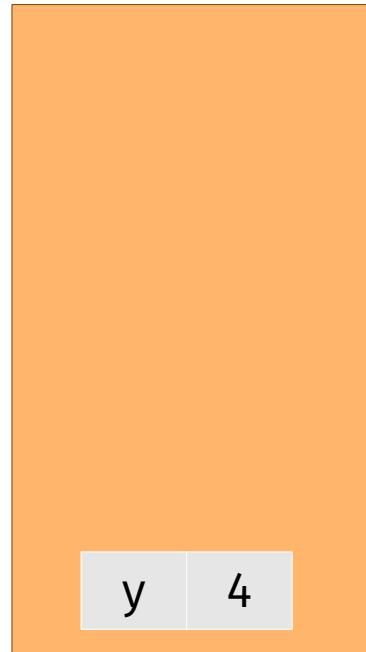
# Ejemplo de ejecución

```
void f(int x) {  
    int y = 40;  
}
```

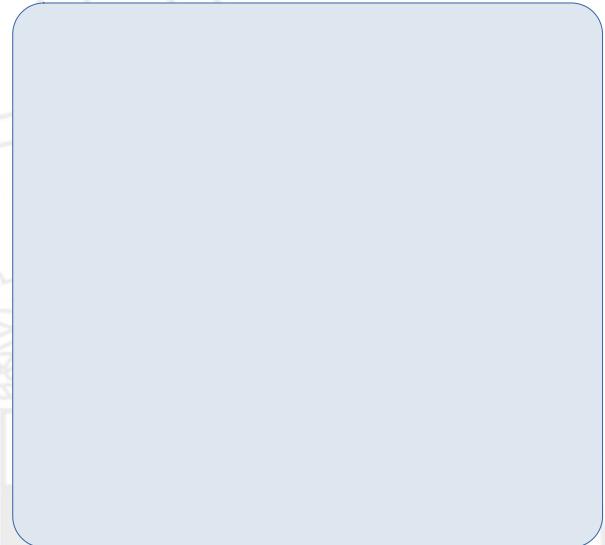
```
void g() {  
    int x = 10;  
    int z = 20;  
    f(30);  
    x = 50;  
}
```

```
int main() {  
    int y = 4;  
    g();  
}
```

Pila



Heap



ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# EL TAD Cola

Manuel Montenegro Montes

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

# ¿Qué es una cola?

- Es una colección de elementos que permite:
  - Insertar elementos.
  - Borrar elementos en el orden en el que han sido insertados.
  - Obtener el primer elemento insertado no borrado.



# ¿Qué es una cola?

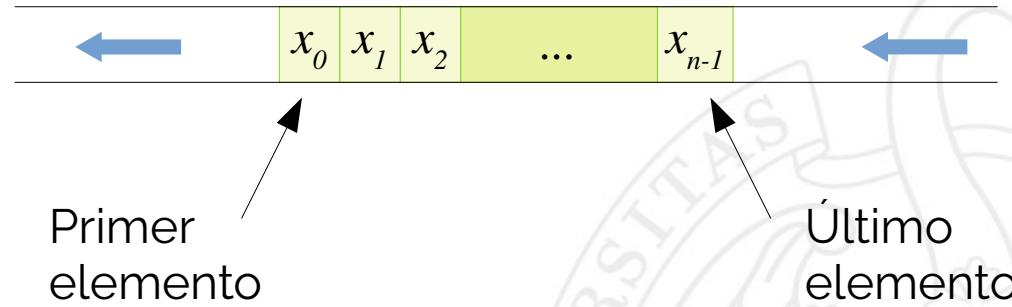
- Las colas siguen una disciplina de acceso **FIFO**

**First In, First Out**



# Modelo de colas

- Conceptualmente representamos las colas de esta forma:



# Operaciones sobre colas

- **Constructoras:**
  - Crear una cola vacía (***create\_empty***).
- **Mutadoras:**
  - Añadir un elemento al final de la cola (***enqueue***)
  - Eliminar el primer elemento de la cola (***dequeue***).
- **Observadoras:**
  - Obtener el primer elemento de la cola (***front***)
  - Saber si una cola está vacía (***empty***).

# Operaciones sobre colas

- **Constructoras:**
  - Crear una cola vacía (*create\_empty*).
- **Mutadoras:**
  - Añadir un elemento al final de la cola (*enqueue push*)
  - Eliminar el primer elemento de la cola (*dequeue pop*).
- **Observadoras:**
  - Obtener el primer elemento de la cola (*front*)
  - Saber si una cola está vacía (*empty*).

# Operación *create\_empty*

{ true }

***create\_empty()*** → (Q: Queue)

{ Q = \_\_\_\_\_ }



# Operaciones *push* (*enqueue*) y *pop* (*dequeue*)

$$\{ Q = \underline{\underline{x_0 \ x_1 \ \dots \ x_{n-1}}} \}$$

***push*(Q: Queue, x: elem)**

$$\{ Q = \underline{\underline{x_0 \ x_1 \ \dots \ x_{n-1} \ x}} \}$$

$$\{ Q = \underline{\underline{x_0 \ x_1 \ \dots \ x_{n-1}}} \quad n \geq 1 \}$$

***pop*(Q: Queue)**

$$\{ Q = \underline{\underline{\phantom{x_0 \ x_1 \ \dots \ x_{n-1}}}} \}$$

# Operaciones *front* y *size*

$$\left\{ Q = \overbrace{\quad | x_0 | x_1 | \dots | x_{n-1} | \quad}^{n \geq 1} \right\}$$

**front**(Q: Queue)  $\rightarrow$  (x: elem)

$$\{ x = x_0 \}$$

$$\left\{ Q = \overbrace{\quad | x_0 | x_1 | \dots | x_{n-1} | \quad}^{n > 0} \right\}$$

**empty**(Q: Queue)  $\rightarrow$  (x: elem)

$$\{ b \Leftrightarrow n = 0 \}$$

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

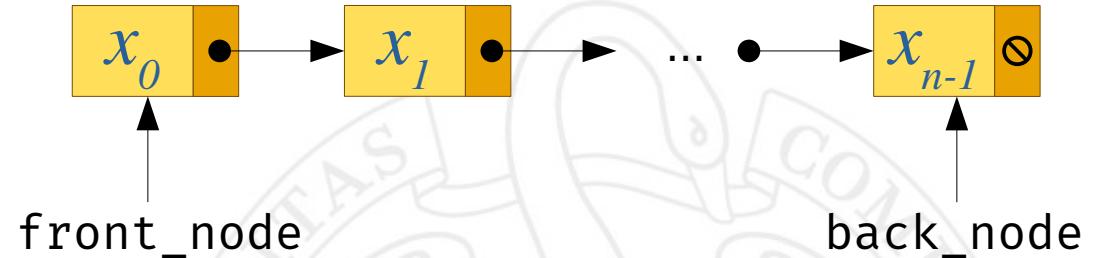
# Implementando el TAD Cola

Manuel Montenegro Montes

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

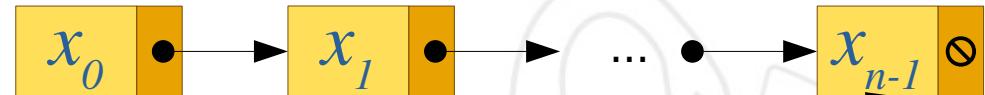
# Implementación mediante listas enlazadas

# Implementación mediante listas enlazadas



# Clase QueueLinkedList

```
template<typename T>
class QueueLinkedList {  
  
...  
  
private:  
    struct Node {  
        T value;  
        Node *next;  
    };  
  
    Node *front_node;  
    Node *back_node;  
};
```



- Si la cola está vacía:

$\text{front\_node} = \text{back\_node} = \text{nullptr}$

# Interfaz pública de QueueLinkedList

```
template<typename T>
class QueueLinkedList {

    QueueLinkedList();
    QueueLinkedList(const QueueLinkedList &other);
    ~QueueLinkedList();

    QueueLinkedList & operator=(const QueueLinkedList &other);

    void push(const T &elem);
    void pop();
    T & front();
    const T & front() const;
    bool empty() const;

    ...
};

}
```



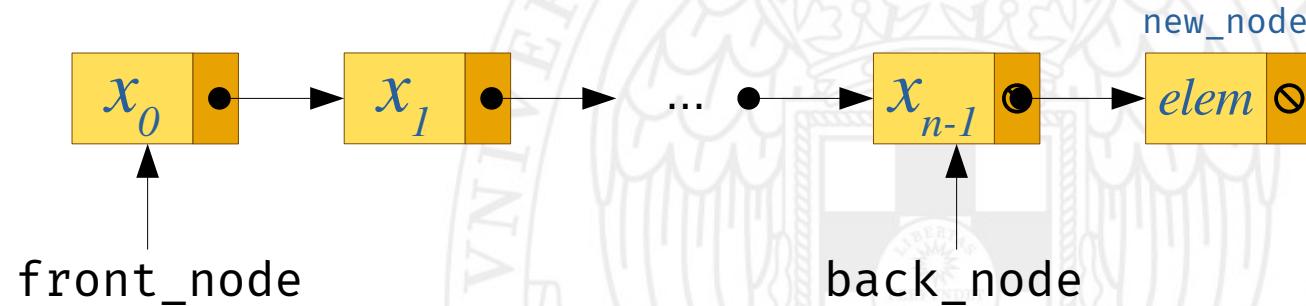
# Interfaz pública de QueueLinkedList

```
template<typename T>
class QueueLinkedList {  
  
    QueueLinkedList();
    QueueLinkedList(const QueueLinkedList &other);
    ~QueueLinkedList();  
  
    QueueLinkedList & operator=(const QueueLinkedList &other);  
  
    void push(const T &elem);
    void pop();
    T & front();
    const T & front() const;
    bool empty() const;  
  
    ...  
};
```



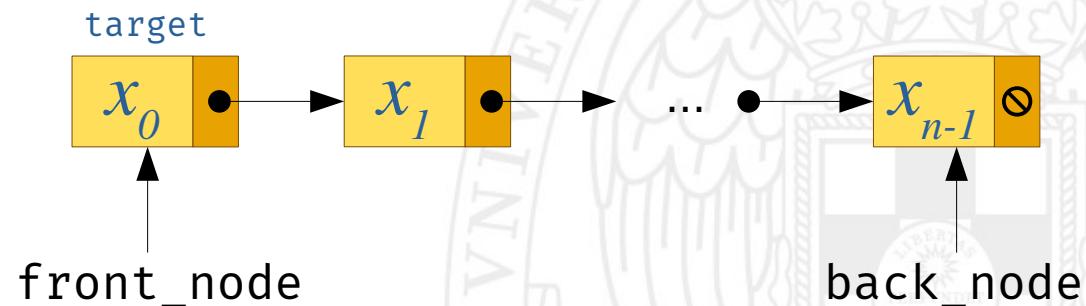
# Método push()

```
void push(const T &elem) {  
    Node *new_node = new Node { elem, nullptr };  
    if (back_node == nullptr) {  
        back_node = new_node;  
        front_node = new_node;  
    } else {  
        back_node->next = new_node;  
        back_node = new_node;  
    }  
}
```



# Método pop()

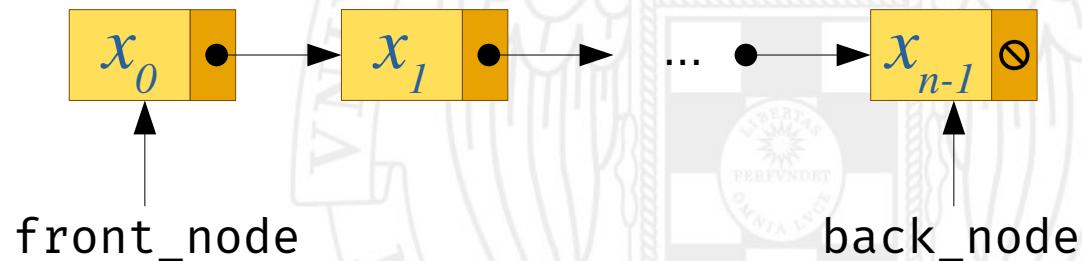
```
void pop() {
    assert (front_node != nullptr);
    Node *target = front_node;
    front_node = front_node->next;
    if (back_node == target) {
        back_node = nullptr;
    }
    delete target;
}
```



# Métodos front() y empty()

```
const T & front() const {
    assert (front_node != nullptr);
    return front_node->value;
}

bool empty() const {
    return (front_node == nullptr);
}
```



# Implementación mediante vectores circulares

# Implementación mediante vectores circulares



# Idea: vectores circulares

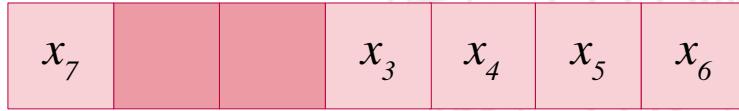
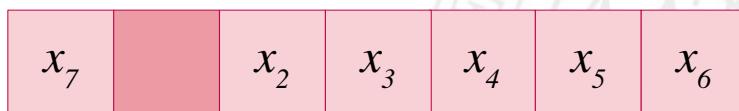
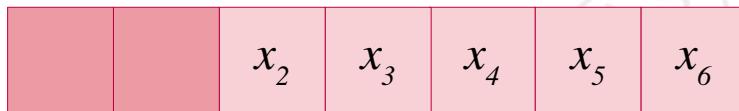
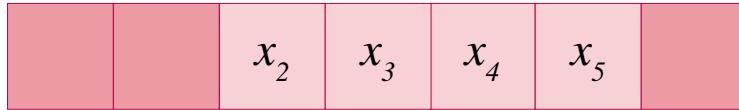
$x_0$	$x_1$	$x_2$	$x_3$	$x_4$		
-------	-------	-------	-------	-------	--	--

$x_0$	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	
-------	-------	-------	-------	-------	-------	--

	$x_1$	$x_2$	$x_3$	$x_4$	$x_5$	
--	-------	-------	-------	-------	-------	--

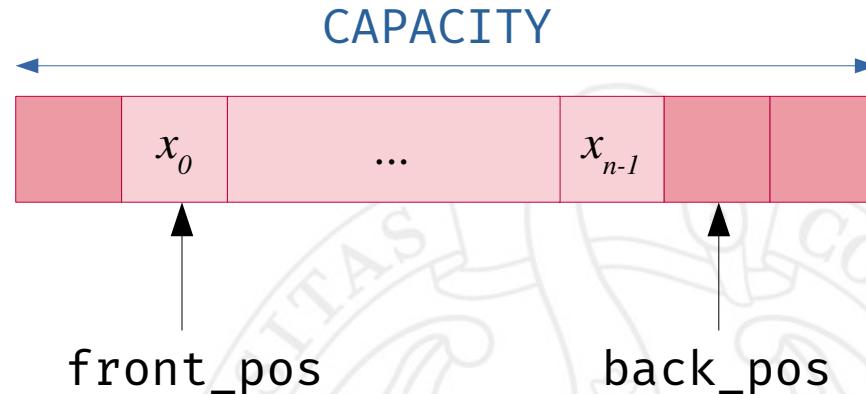
		$x_2$	$x_3$	$x_4$	$x_5$	
--	--	-------	-------	-------	-------	--

# Idea: vectores circulares



# Clase QueueArray

```
class QueueArray {  
public:  
    ...  
  
private:  
    T *elems;  
    int front_pos, back_pos;  
};
```



- Si la cola está vacía:  
 $\text{front\_pos} == \text{back\_pos}$

# Constructor

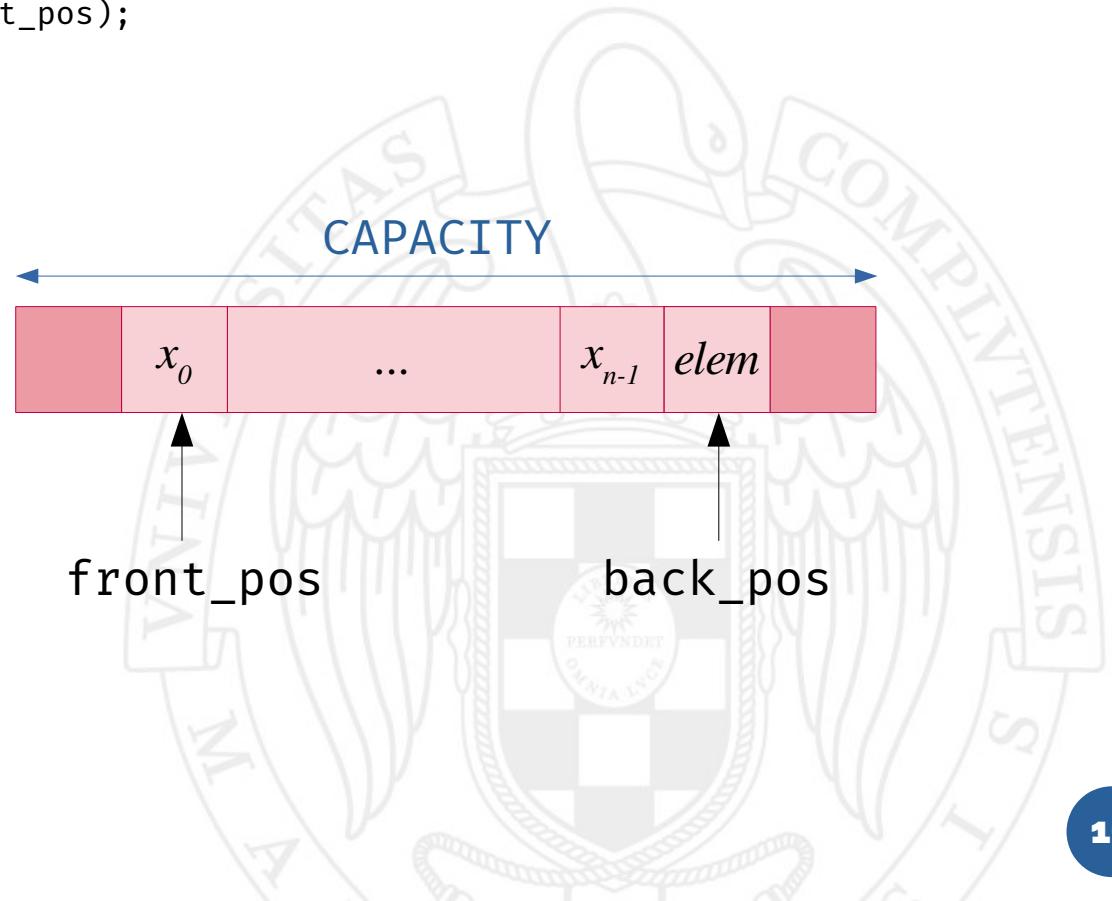
```
QueueArray() {  
    elems = new T[CAPACITY];  
    front_pos = 0;  
    back_pos = 0;  
}
```



front\_pos  
back\_pos

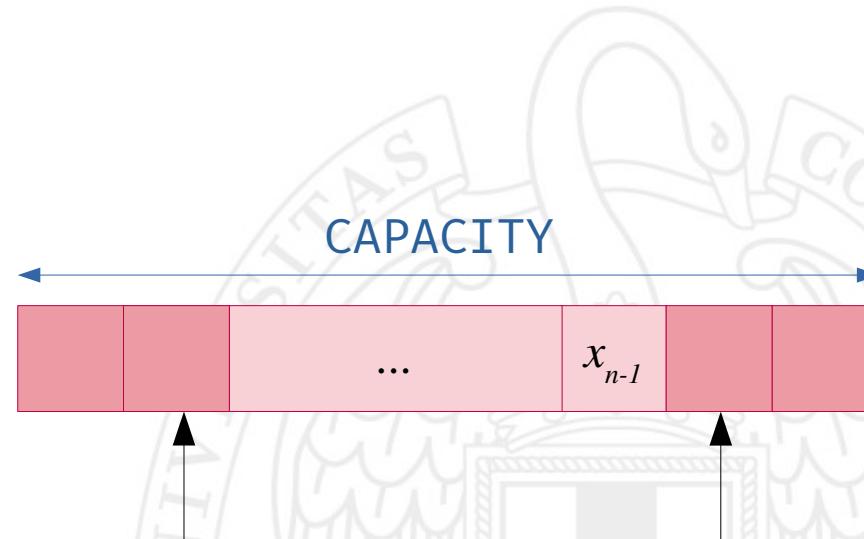
# Método push()

```
void push(const T &elem) {  
    // Cabe el elemento en la cola?  
    assert ((back_pos + 1) % CAPACITY != front_pos);  
    elems[back_pos] = elem;  
    back_pos = (back_pos + 1) % CAPACITY;  
}
```



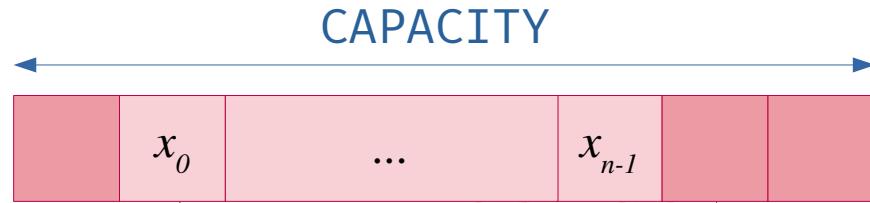
# Método pop()

```
void pop() {  
    assert (front_pos != back_pos);  
    front_pos = (front_pos + 1) % CAPACITY;  
}
```

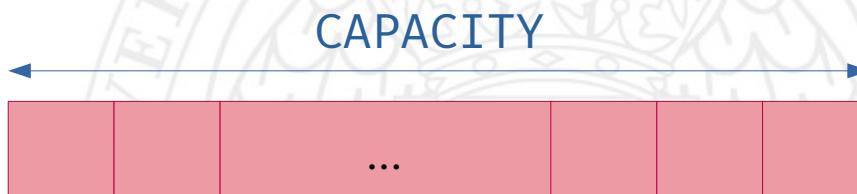


# Métodos front() y empty()

```
const T& front() const {
    assert(front_pos != back_pos);
    return elems[front_pos];
}
```



```
bool empty() const {
    return front_pos == back_pos;
}
```



front\_pos  
back\_pos

# Coste de las operaciones

Operación	Listas enlazadas	Vectores circulares
push	$O(1)$	$O(1)$
pop	$O(1)$	$O(1)$
front	$O(1)$	$O(1)$
empty	$O(1)$	$O(1)$

$n$  = número de elementos en la cola

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# EL TAD de colas dobles (*deque*)

Manuel Montenegro Montes

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

# ¿Qué es una cola doble?

- Es una estructura de datos que combina la funcionalidad de la pila con la funcionalidad de la cola. En particular:
  - Insertar/eliminar/acceder a elementos al final de la cola doble.
  - Insertar/eliminar/acceder a elementos al principio de la cola doble.



# Operaciones de una cola doble

- **Constructoras:**
  - Crear una cola doble vacía (*create\_empty*).
- **Mutadoras:**
  - Añadir un elemento al principio (*push\_front*).
  - Añadir un elemento al final (*push\_back*).
  - Eliminar un elemento del principio (*pop\_front*).
  - Eliminar un elemento del final (*pop\_back*).
- **Observadoras:**
  - Comprobar si una cola doble es vacía (*empty*).
  - Acceder al primer elemento de la cola doble (*front*).
  - Acceder al último elemento de la cola doble (*back*).



# TAD Colas dobles vs TAD Lista

## Colas dobles

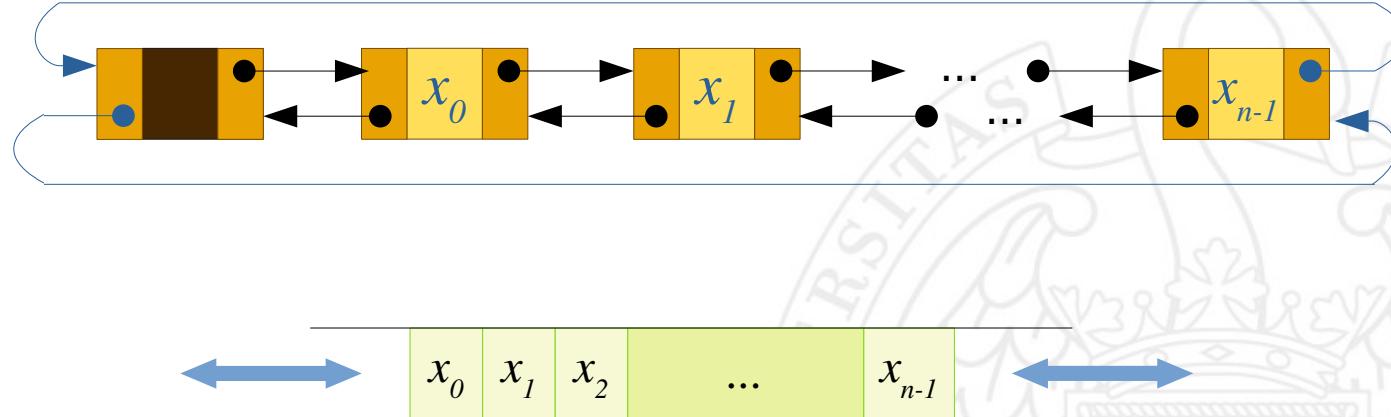
- Constructoras:
  - *create\_empty*
- Mutadoras:
  - *push\_front*
  - *push\_back*
  - *pop\_front*
  - *pop\_back*
- Observadoras:
  - *empty*
  - *front*
  - *back*

## Listas

- Constructoras:
  - *create\_empty*
- Mutadoras:
  - *push\_front*
  - *push\_back*
  - *pop\_front*
  - *pop\_back*
- Observadoras:
  - *empty*
  - *front*
  - *back*
  - *size*
  - *at*

# Implementación de colas dobles

- Puede utilizarse una lista circular con nodo fantasma:



# Coste de las operaciones

Operación	Coste en tiempo
create_empty	$O(1)$
push_back	$O(1)$
push_front	$O(1)$
pop_back	$O(1)$
pop_front	$O(1)$
front	$O(1)$
back	$O(1)$
empty	$O(1)$

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Introducción a los iteradores

Manuel Montenegro Montes

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

# Motivación

# Problema

- Tenemos una lista de enteros, y queremos calcular la suma de todos los elementos de la lista.

```
int suma_elems(const ListLinkedDouble<int> &l) {  
    int suma = 0;  
    for (int i = 0; i < l.size(); i++) {  
        suma += l[i];  
    }  
    return suma;  
}
```

¿Qué coste tiene esta función?

# Possible solución 1

- Utilizar otra implementación de listas, de modo que la operación `at()` tenga coste constante.

```
int suma_elems(const ListArray<int> &l) {  
    int suma = 0;  
    for (int i = 0; i < l.size(); i++) {  
        suma += l[i];  
    }  
    return suma;  
}
```

¿Y si necesito una lista enlazada?

# Possible solución 2

- Hacer copia de la lista de entrada, e ir eliminando los elementos de la copia uno a uno.

```
int suma_elems(const ListLinkedDouble<int> &l) {  
    ListLinkedDouble<int> copia = l;  
    int suma = 0;  
    while (!copia.empty()) {  
        suma += copia.front();  
        copia.pop_front();  
    }  
    return suma;  
}
```

¿Cuál es el coste en espacio de esta solución?

# Possible solución 3

- Integrar la operación dentro de la clase `ListLinkedDouble`.

```
template <typename T>
class ListLinkedDouble {
    ...
    int suma_elems() const {
        int suma = 0;
        Node *current = head->next;
        while (current != head) {
            suma += current->value;
            current = current->next;
        }
        return suma;
    }
};
```

¿Y si la lista no es de enteros?

¿Tengo que prever de antemano todas las cosas que puedo hacer en el recorrido de una lista?

# La solución que presentamos

- Proporcionar una abstracción al programador/a para que pueda navegar por los elementos de una lista, de modo independiente de la implementación.
- La navegación se realiza de manera secuencial.
- Esta abstracción recibe el nombre de **iterador**.



# Iteradores

# Iteradores

- Un iterador es un cursor que se mueve por los elementos de la lista de manera secuencial.
- Es posible realizar operaciones de acceso y modificación en la posición actual de un iterador.

```
[ 1, 4, 15, 7, 10, 23 ]
```

# Iteradores

- En el momento de su creación, un iterador está ligado a una lista.
- Representamos los iteradores de la siguiente forma:

$$[ \ x_0, |x_1|, x_2, \dots, x_{n-1} ]$$

iterador

Elemento apuntado por  
el iterador.

# Operaciones sobre un iterador

- Obtener el elemento apuntado por el iterador (**elem**)
- Avanzar el iterador a la siguiente posición de la lista (**advance**)
- Igualdad entre dos iteradores (==)
  - Dos iteradores son iguales si recorren la misma lista y apuntan a la misma posición dentro de esta.

$$\{ [x_0, \dots, \underset{\text{it}}{|} x_i, \dots, x_{n-1}], 0 \leq i < n \}$$

**elem**(*it*: Iterator) → (*x*: Elem)

$$\{ x = x_i \}$$
$$\{ [x_0, \dots, \underset{\text{it}}{|} x_i, \dots, x_{n-1}], 0 \leq i < n \}$$

**advance**(*it*: Iterator)

$$\{ [x_0, \dots, x_i, \underset{\text{it}}{|} x_{i+1}, \dots, x_{n-1}] \}$$

# Creación de iteradores

- Añadimos dos operaciones al TAD Lista:
  - Obtener un iterador al principio de la lista (**begin**)
  - Obtener un iterador al final de la lista (**end**)

$\{ l = [x_0, \dots, x_i, \dots, x_{n-1}] \}$

**begin**(*l: List*) → (*it: Iterator*)

$\{ [ \underset{\text{it}}{|} x_0, \dots, x_i, \dots, x_{n-1} ] \}$

$\{ l = [x_0, \dots, x_i, \dots, x_{n-1}] \}$

**end**(*l: List*) → (*it: Iterator*)

$\{ [ x_0, \dots, x_i, \dots, \underset{\text{it}}{|} x_{n-1} ] \}$

# Creación de iteradores

- Añadimos dos operaciones al TAD Lista:
  - Obtener un iterador al principio de la lista (**begin**)
  - Obtener un iterador al final de la lista (**end**)



Las operaciones **elem()** y **advance()** no están definidas para el iterador devuelto por **end()**

# Creación de iteradores

- Añadimos dos operaciones al TAD Lista:
  - Obtener un iterador al principio de la lista (**begin**)
  - Obtener un iterador al final de la lista (**end**)

$$\{ [x_0, \dots, |x_i, \dots, x_{n-1}], \textcolor{red}{0 \leq i < n} \}$$

**elem**(*it: Iterator*) → (*x: Elem*)

$$\{ x = x_i \}$$
$$\{ [x_0, \dots, |x_i, \dots, x_{n-1}], \textcolor{red}{0 \leq i < n} \}$$

**advance**(*it: Iterator*)

$$\{ [x_0, \dots, x_i, |x_{i+1}, \dots, x_{n-1}] \}$$

# Operaciones adicionales sobre listas

- Insertar un elemento en la posición apuntada por un iterador (**insert**)
- Eliminar el elemento apuntado por el iterador (**erase**)

$$\{ l = [ x_0, \dots, |x_i, \dots, x_{n-1}| ] \}$$

**insert**(*l*: List, *it*: Iterator, *e*: Elem)  $\rightarrow$  *it'*: Iterator

$$\{ l = [ x_0, \dots, |e, x_i, \dots, x_{n-1}| ] \}$$
$$\{ l = [ x_0, \dots, |x_i, x_{i+1}, \dots, x_{n-1}| ], i < n \}$$

**erase**(*l*: List, *it*: Iterator)  $\rightarrow$  *it'*: Iterator

$$\{ l = [ x_0, \dots, |x_{i+1}, \dots, x_{n-1}| ] \}$$

# Ejemplo: suma de enteros

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



# Ejemplo: suma de enteros

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



ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Iteradores y listas enlazadas

Manuel Montenegro Montes

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

# Operaciones a implementar

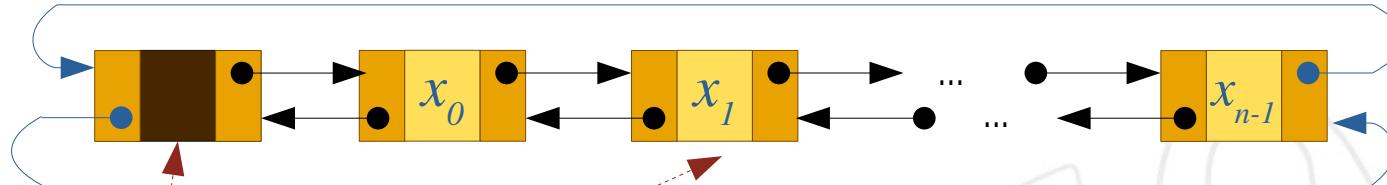
## Para iteradores

- Obtener el elemento apuntado por el iterador (**elem**)
- Avanzar el iterador a la siguiente posición de la lista (**advance**)
- Igualdad entre dos iteradores (==)
  - Dos iteradores son iguales si recorren la misma lista y apuntan a la misma posición dentro de esta.

## Para listas

- Obtener un iterador al principio de la lista (**begin**)
- Obtener un iterador al final de la lista (**end**)

# Iteradores en listas doblemente enlazadas



**Iterador**

head:	•
current:	•

- Un iterador contiene dos atributos:
  - El nodo del elemento apuntado por el iterador (**current**).
  - El nodo fantasma de la lista a la que el iterador pertenece (**head**).

# La clase ListLinkedDouble<T> :: iterator

```
template <typename T>
class ListLinkedDouble {
public:
    ...
    class iterator {
public:
    iterator(Node *head, Node *current): head(head), current(current) { }
    ...
private:
    Node *head;
    Node *current;
};
...
};
```

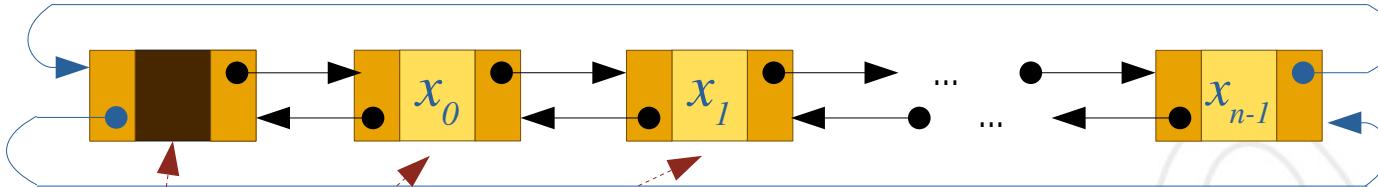
# La clase ListLinkedDouble<T> :: iterator

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

};

...;
```

# Implementación de advance()

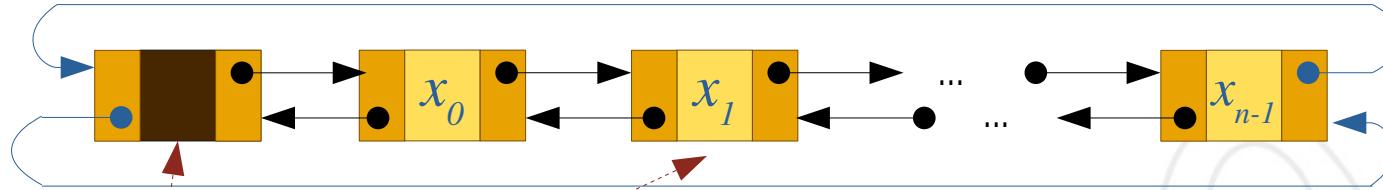


Iterador

head:	
current:	

```
class iterator {  
public:  
    void advance() {  
        assert (current != head);  
        current = current->next;  
    }  
    ...  
};
```

# Implementación de elem()



**Iterador**

head:	
current:	

```
class iterator {  
public:  
    T & elem() {  
        assert (current != head);  
        return current->value;  
    }  
    ...  
};
```

# Implementación de los operadores == y !=

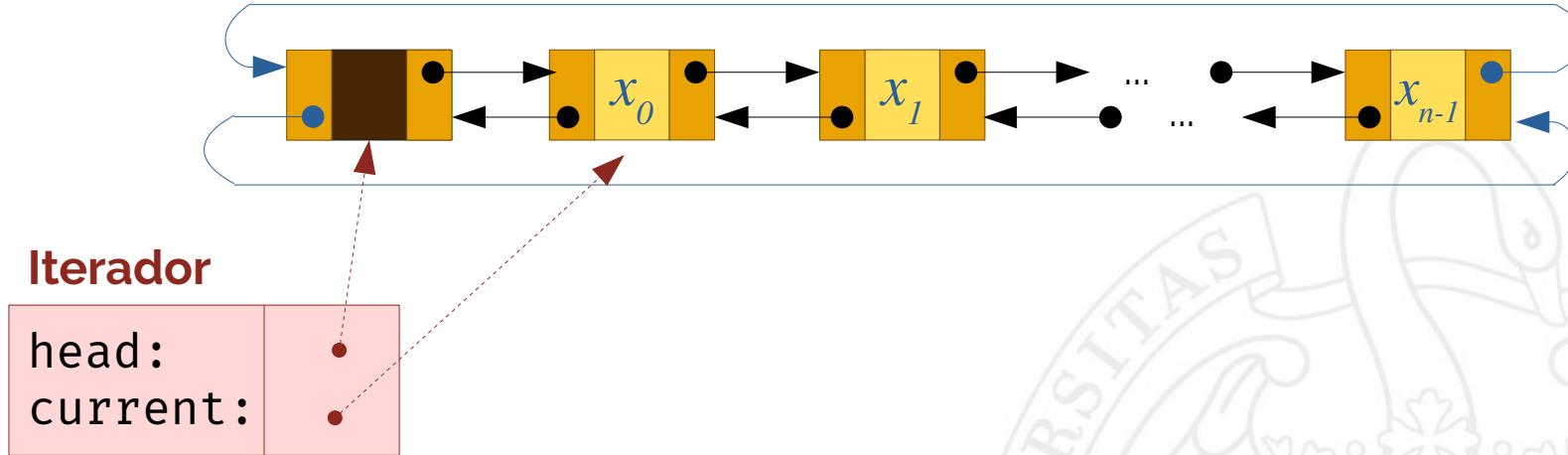
```
class iterator {  
public:  
    bool operator==(const iterator &other) const {  
        return (head == other.head) &&  
               (current == other.current);  
    }  
  
    bool operator!=(const iterator &other) const {  
        return !(*this == other);  
    }  
};
```

# Creación de iteradores

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

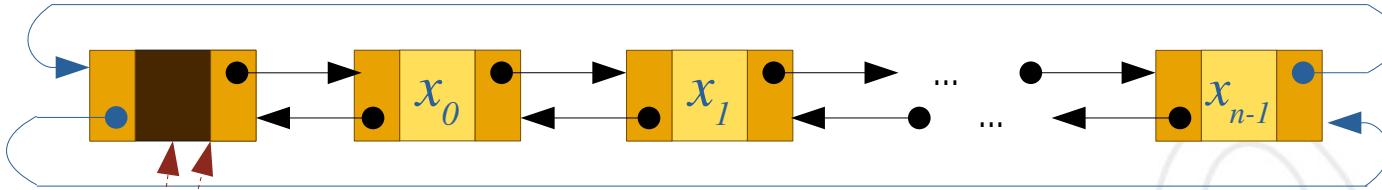
Nuevos métodos

# Implementación de begin()



```
template <typename T>
class ListLinkedDouble {
    ...
    iterator begin() {
        return iterator(head, head->next);
    }
};
```

# Implementación de end()



Iterador

head:	•
current:	•

```
template <typename T>
class ListLinkedDouble {
    ...
    iterator end() {
        return iterator(head, head);
    }
};
```

# Ejemplo

- Dada una lista de nombres de personas, imprimir aquellas que empiecen por un carácter pasado como parámetro:

```
void imprime_empieza_por(ListLinkedDouble<std::string> &l, char c) {  
    for (ListLinkedDouble<std::string>::iterator it = l.begin();  
        it != l.end();  
        it.advance()) {  
  
        if (it.elem()[0] == c) {  
            std::cout << it.elem() << std::endl;  
        }  
    }  
}
```

# Ejemplo

- Modificar todos los elementos de una lista de enteros, multiplicándolos por uno pasado como parámetro.

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

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

# Inserción y borrado con iteradores

Manuel Montenegro Montes

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

# Operaciones a implementar

## Para listas

- Insertar un elemento en la posición apuntada por un iterador (**insert**)
- Eliminar el elemento apuntado por el iterador (**erase**)

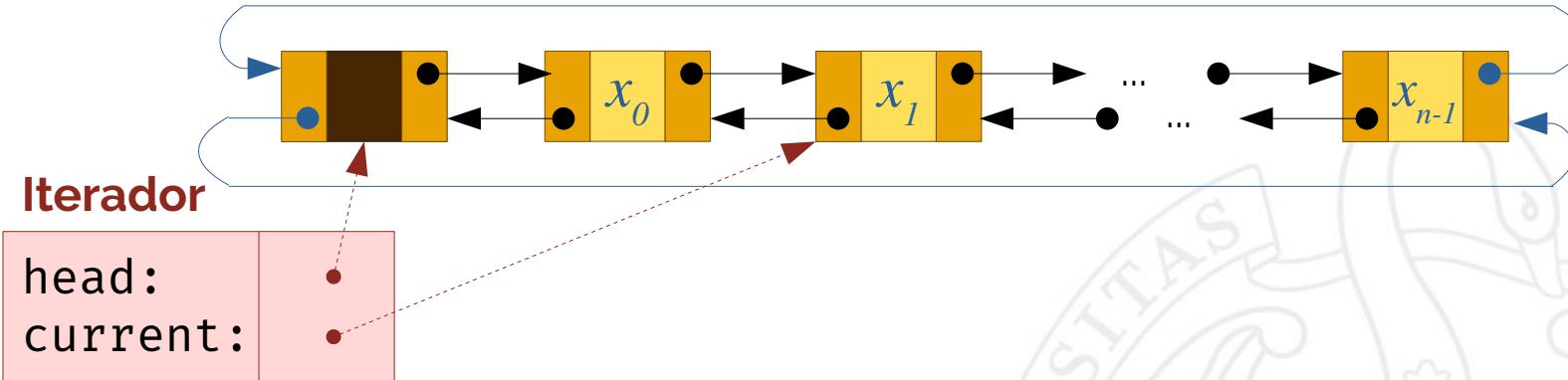


# Interfaz de ListLinkedDouble<T>

```
template <typename T>
class ListLinkedDouble {
public:
    ...
    iterator insert(iterator it, const T &elem);
    iterator erase(iterator it);
    ...
};
```

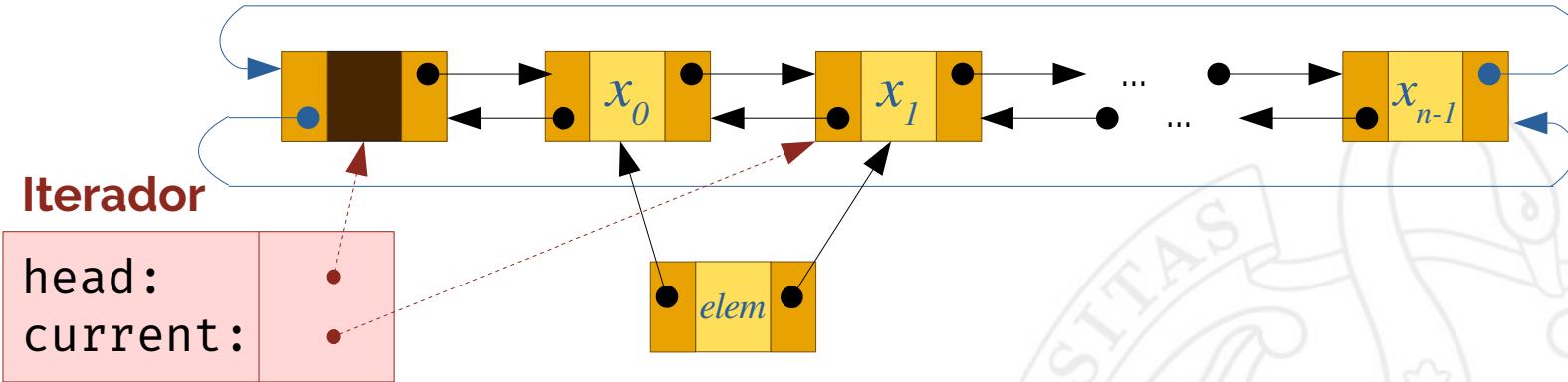


# Insertar un elemento



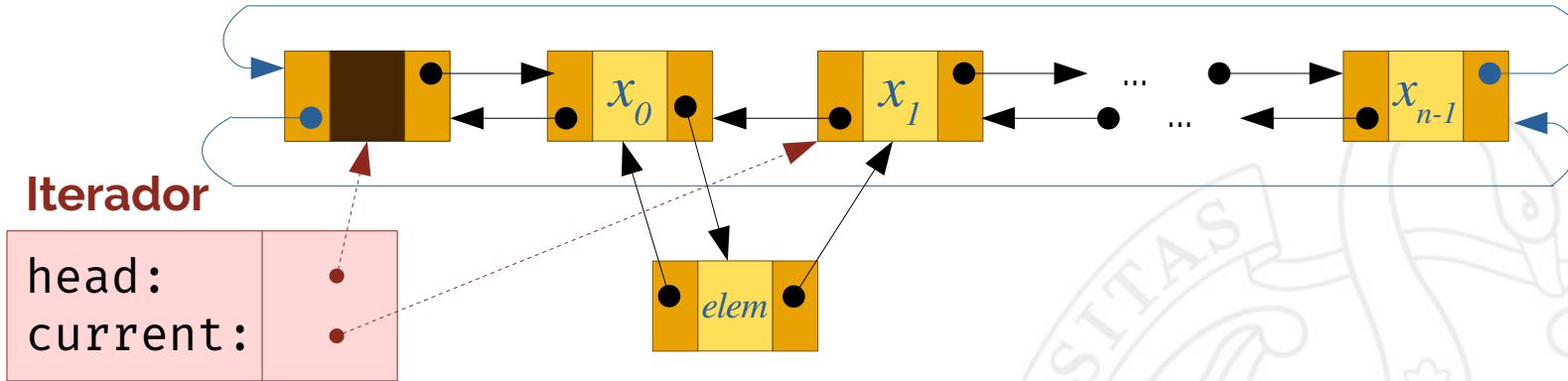
```
iterator insert(iterator it, const T &elem) {
    assert(it.head == head);
    Node *new_node = new Node { elem, it.current, it.current->prev };
    it.current->prev->next = new_node;
    it.current->prev = new_node;
    num_elems++;
    return iterator(head, new_node);
}
```

# Insertar un elemento



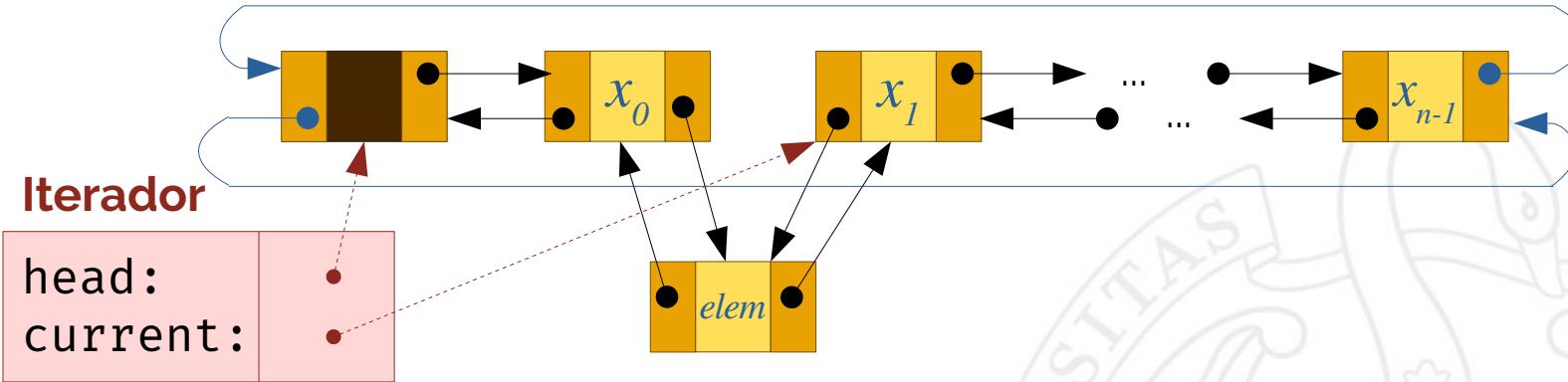
```
iterator insert(iterator it, const T &elem) {
    assert(it.head == head);
    Node *new_node = new Node { elem, it.current, it.current->prev };
    it.current->prev->next = new_node;
    it.current->prev = new_node;
    num_elems++;
    return iterator(head, new_node);
}
```

# Insertar un elemento



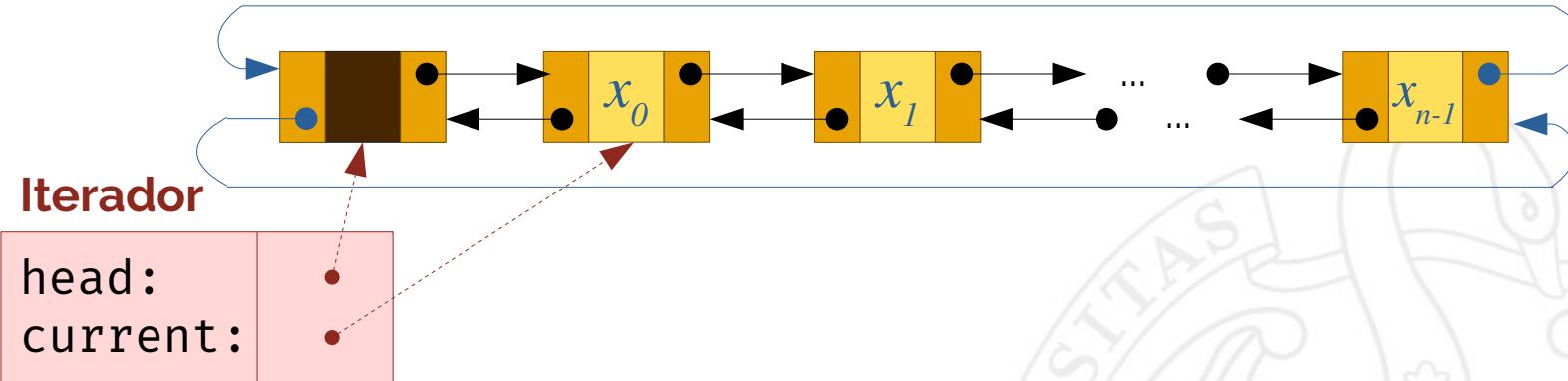
```
iterator insert(iterator it, const T &elem) {
    assert(it.head == head);
    Node *new_node = new Node { elem, it.current, it.current->prev };
    it.current->prev->next = new_node;
    it.current->prev = new_node;
    num_elems++;
    return iterator(head, new_node);
}
```

# Insertar un elemento



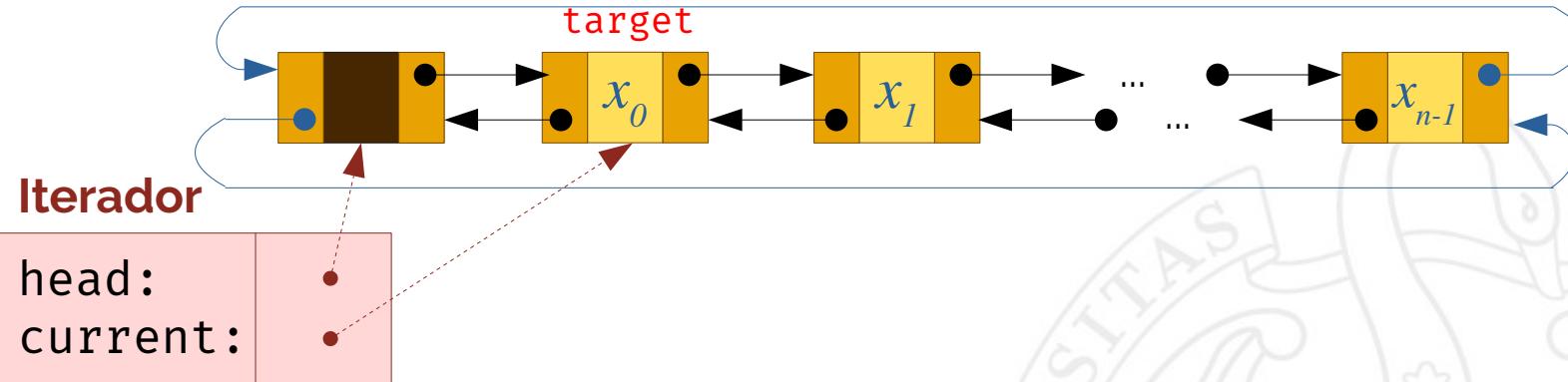
```
iterator insert(iterator it, const T &elem) {
    assert(it.head == head);
    Node *new_node = new Node { elem, it.current, it.current→prev };
    it.current→prev→next = new_node;
    it.current→prev = new_node;
    num_elems++;
    return iterator(head, new_node);
}
```

# Borrar un elemento



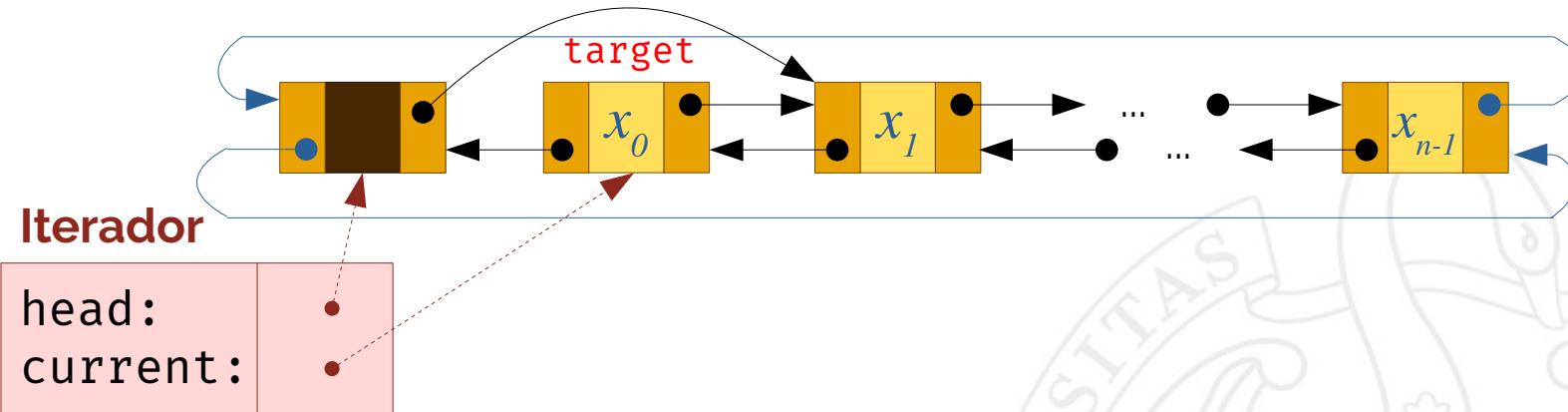
```
iterator erase(iterator it) {
    assert(it.head == head && it.current != head);
    Node *target = it.current;
    target->prev->next = it.current->next;
    target->next->prev = it.current->prev;
    iterator result(head, target->next);
    delete target;
    num_elems--;
    return result;
}
```

# Borrar un elemento



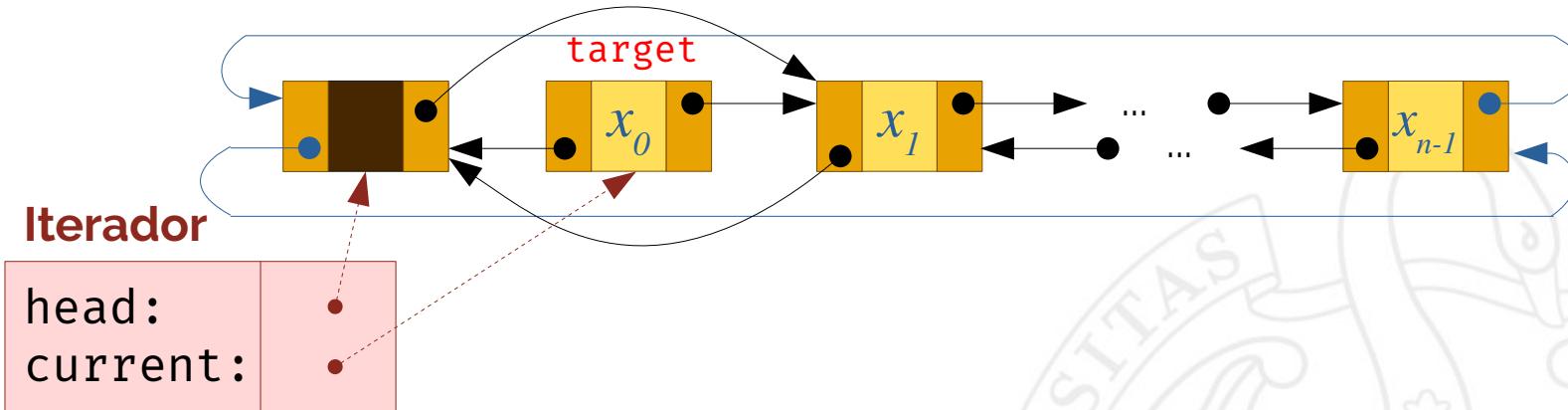
```
iterator erase(iterator it) {
    assert(it.head == head && it.current != head);
    Node *target = it.current;
    target->prev->next = it.current->next;
    target->next->prev = it.current->prev;
    iterator result(head, target->next);
    delete target;
    num_elems--;
    return result;
}
```

# Borrar un elemento



```
iterator erase(iterator it) {
    assert(it.head == head && it.current != head);
    Node *target = it.current;
    target->prev->next = it.current->next;
    target->next->prev = it.current->prev;
    iterator result(head, target->next);
    delete target;
    num_elems--;
    return result;
}
```

# Borrar un elemento

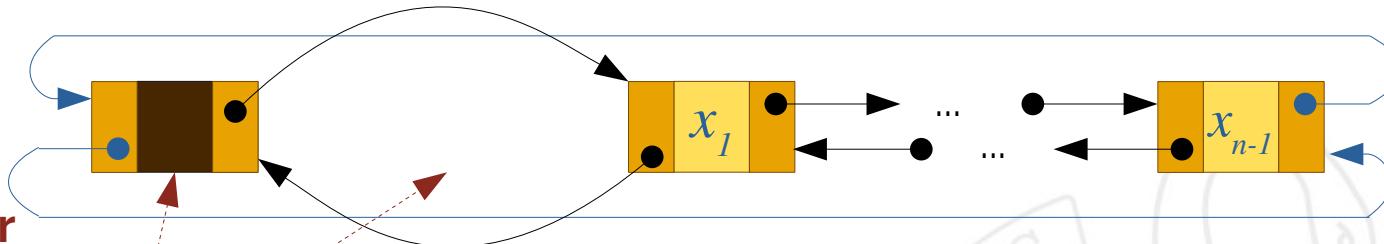


```
iterator erase(iterator it) {
    assert(it.head == head && it.current != head);
    Node *target = it.current;
    target->prev->next = it.current->next;
    target->next->prev = it.current->prev;
    iterator result(head, target->next);
    delete target;
    num_elems--;
    return result;
}
```

# Borrar un elemento

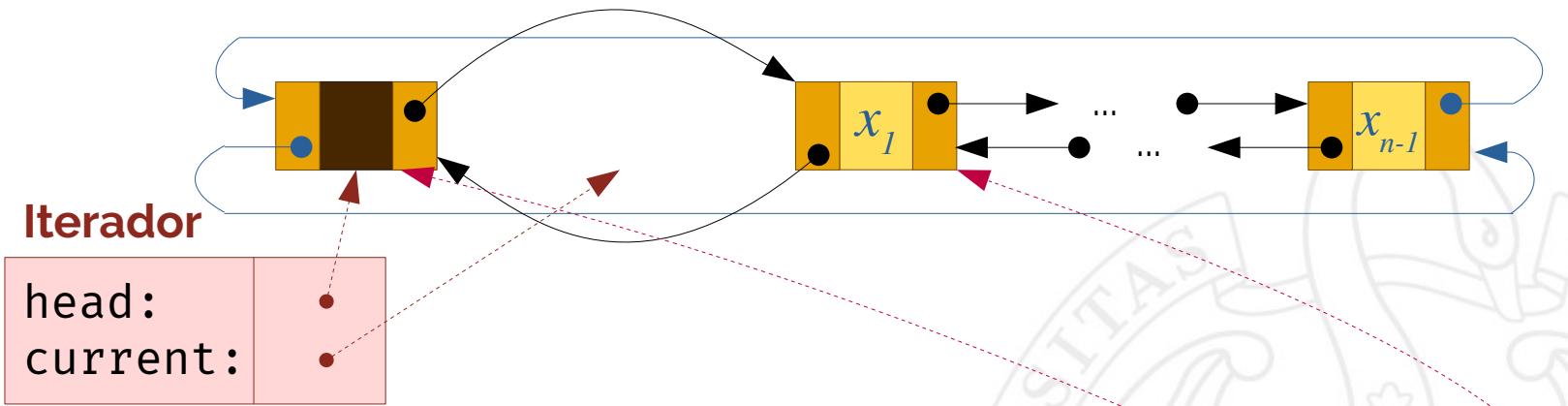
Iterador

head:  
current:



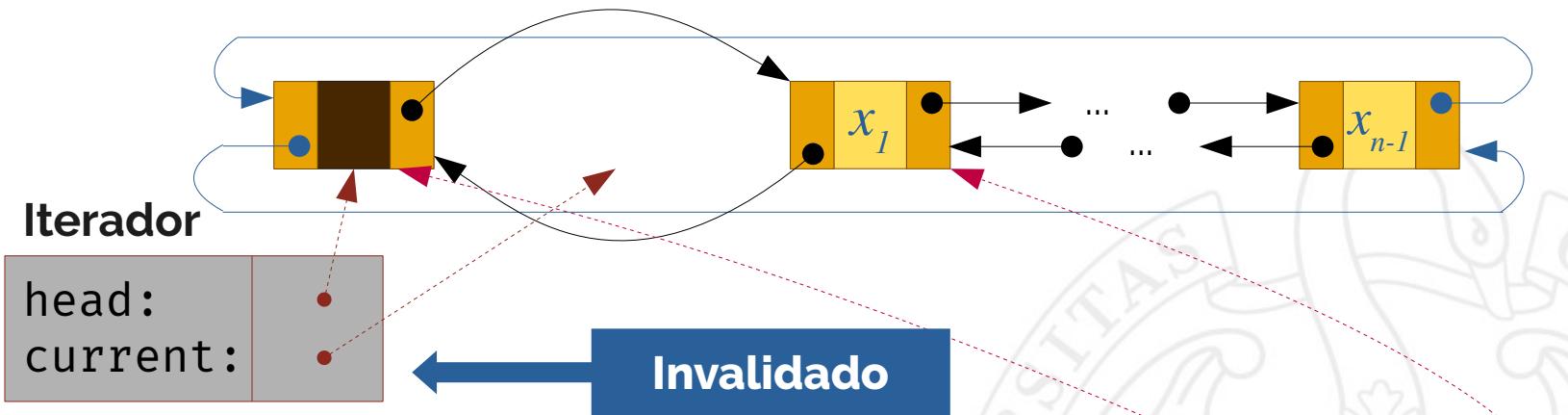
```
iterator erase(iterator it) {
    assert(it.head == head && it.current != head);
    Node *target = it.current;
    target->prev->next = it.current->next;
    target->next->prev = it.current->prev;
    iterator result(head, target->next);
    delete target;
    num_elems--;
    return result;
}
```

# Borrar un elemento



```
iterator erase(iterator it) {
    assert(it.head == head && it.current != head);
    Node *target = it.current;
    target->prev->next = it.current->next;
    target->next->prev = it.current->prev;
    iterator result(head, target->next);
    delete target;
    num_elems--;
    return result;
}
```

# Borrar un elemento



```
iterator erase(iterator it) {
    assert(it.head == head && it.current != head);
    Node *target = it.current;
    target->prev->next = it.current->next;
    target->next->prev = it.current->prev;
    iterator result(head, target->next);
    delete target;
    num_elems--;
    return result;
}
```



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

# ¿Podemos pasar l por referencia constante?

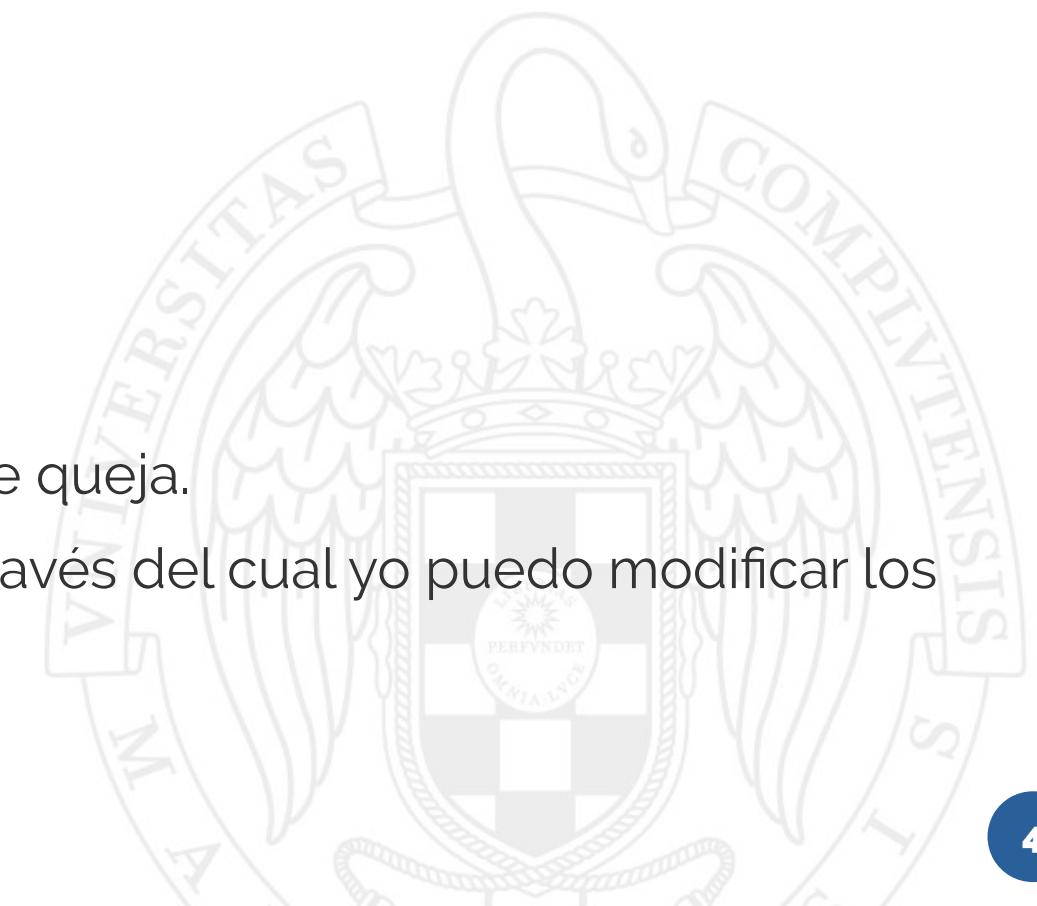
- No, porque begin() y end() no son métodos constantes.

```
int suma_elems(const ListLinkedDouble<int> &l) {  
    int suma = 0;  
    for (ListLinkedDouble<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 ListLinkedDouble {
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 iterador.
- 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(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.

```
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 ListLinkedDouble {
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;

};
```



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;
}
```



# 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;
}
```

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

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

# 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  
}
```

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;  
}
```



# 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;  
    }  
}
```



ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS LINEALES

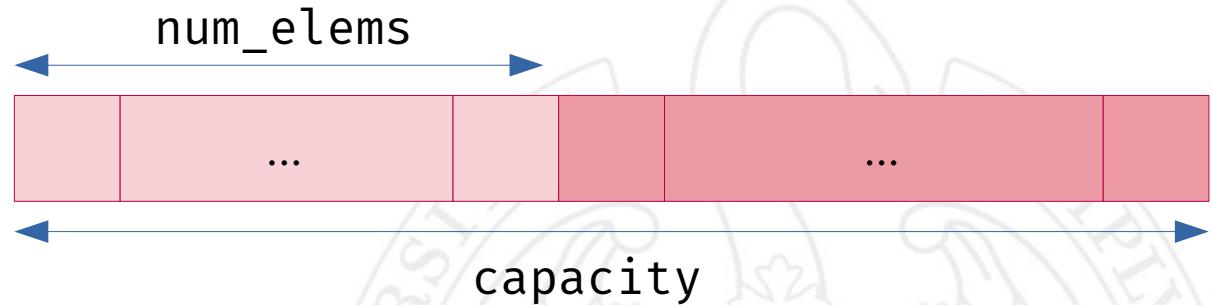
# Iteradores en ListArray

Manuel Montenegro Montes

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

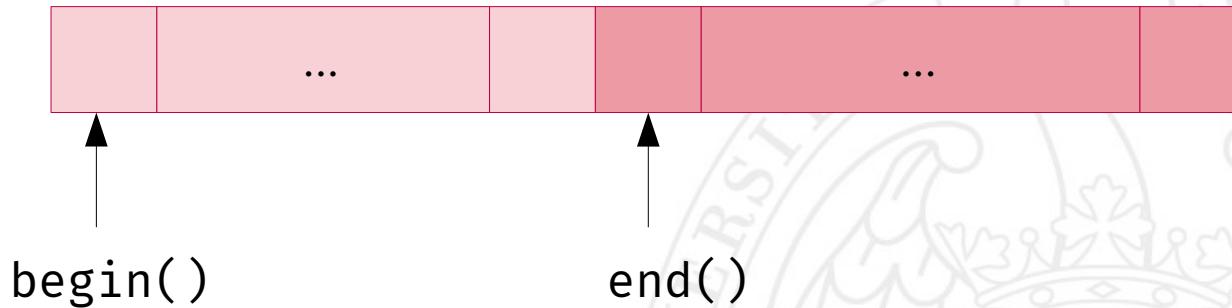
# Recordatorio: ListArray

```
template<typename T>
class ListArray {
public:
...
private:
    int num_elems;
    int capacity;
    T *elems;
};
```



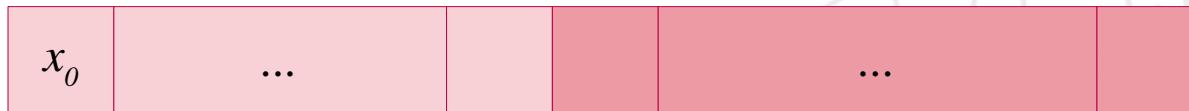
# Iteradores en ListArray

- Los iteradores van a ser **punteros** a los elementos del array.



# Iteradores en ListArray

- Los iteradores van a ser **punteros** a los elementos del array.

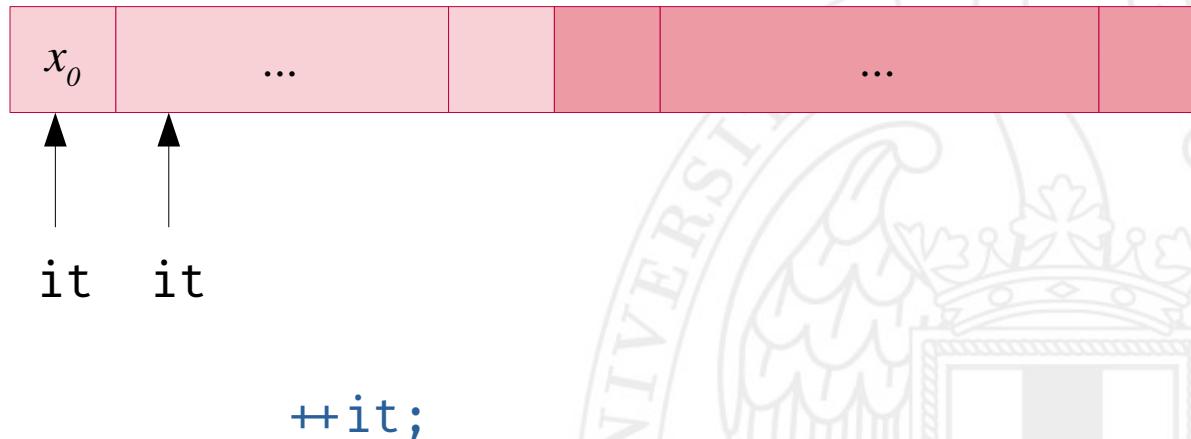


it

$$*it = x_0$$

# Iteradores en ListArray

- Los iteradores van a ser **punteros** a los elementos del array.



# Definición de iteradores

```
template<typename T>
class ListArray {
public:
    ...
    using iterator = T *;
    using const_iterator = const T *;

private:
    int num_elems;
    int capacity;
    T *elems;
};
```



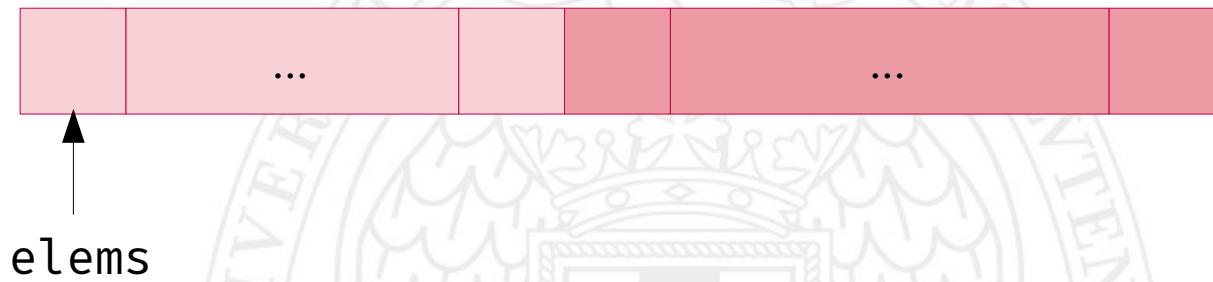
# Método begin()

```
template<typename T>
class ListArray {
public:
    ...
    using iterator = T *;
    using const_iterator = const T *;

    iterator begin() {
        return elems;
    }

    const_iterator begin() const {
        return elems;
    }

private:
    int num_elems;
    int capacity;
    T *elems;
};
```



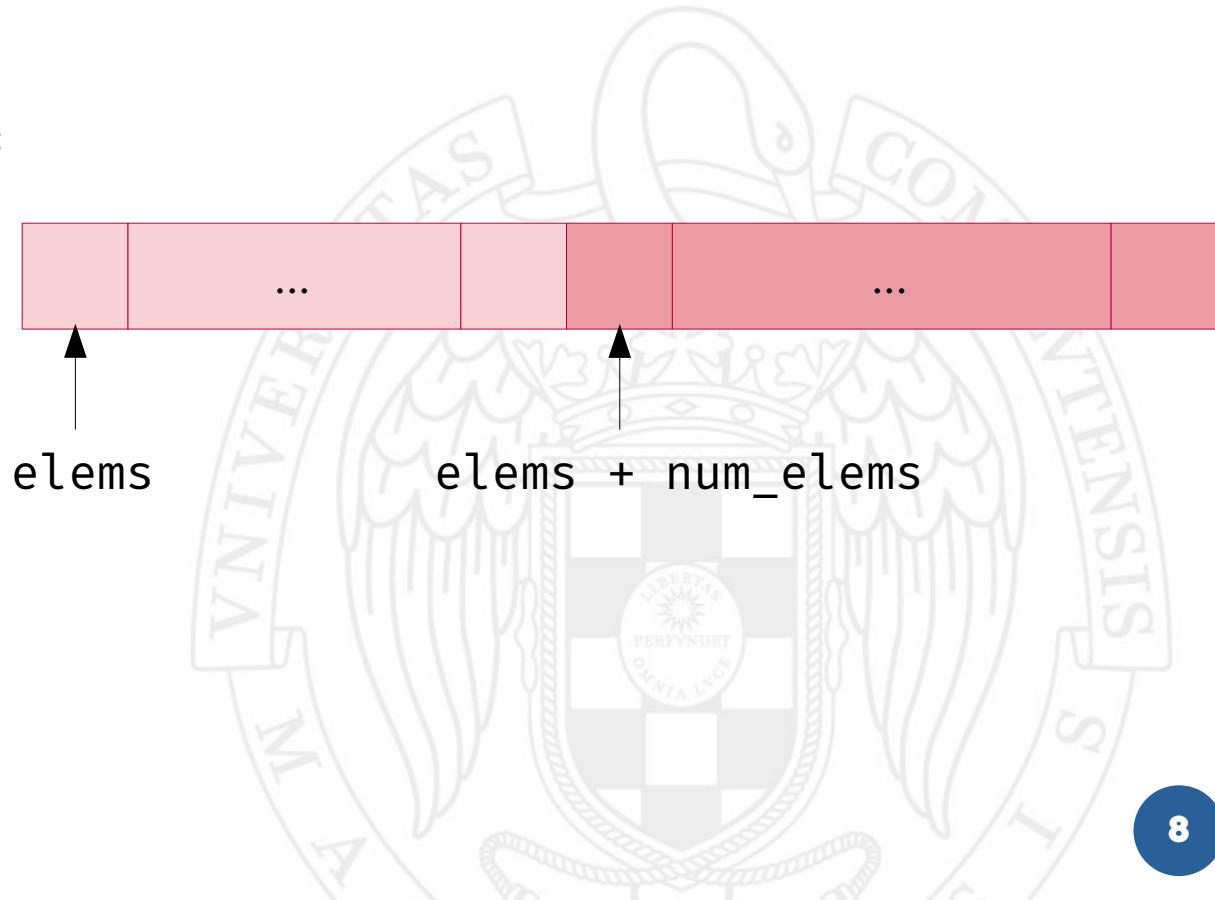
# Método end()

```
template<typename T>
class ListArray {
public:
    ...
    using iterator = T *;
    using const_iterator = const T *;

    iterator end() {
        return elems + num_elems;
    }

    const_iterator end() const {
        return elems + num_elems;
    }

private:
    int num_elems;
    int capacity;
    T *elems;
};
```



# Ejemplos

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

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

# Ejemplos

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

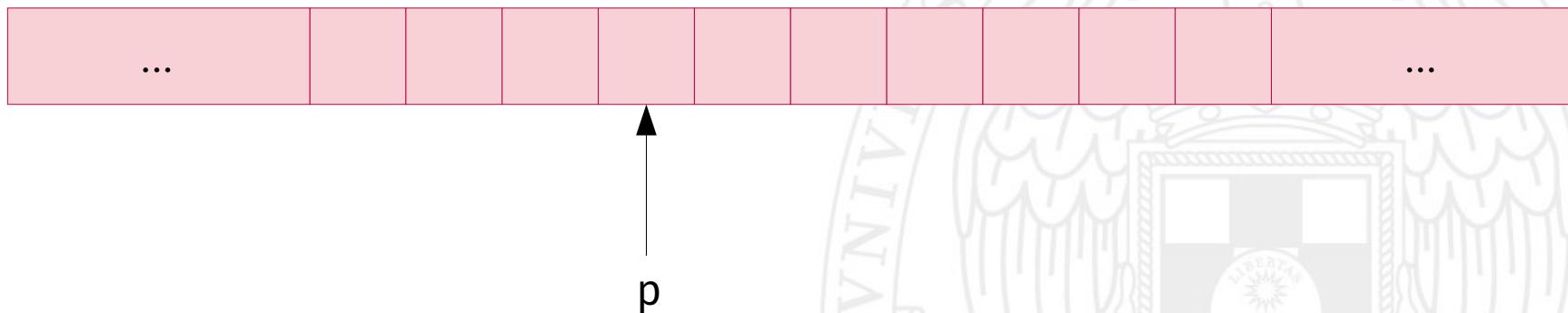
int suma_elems(const ListArray<int> &l) {
    int suma = 0;
    for (int x : l) {
        suma += x;
    }
    return suma;
}
```



# Moraleja

- En C++, los iteradores son generalizaciones de punteros.

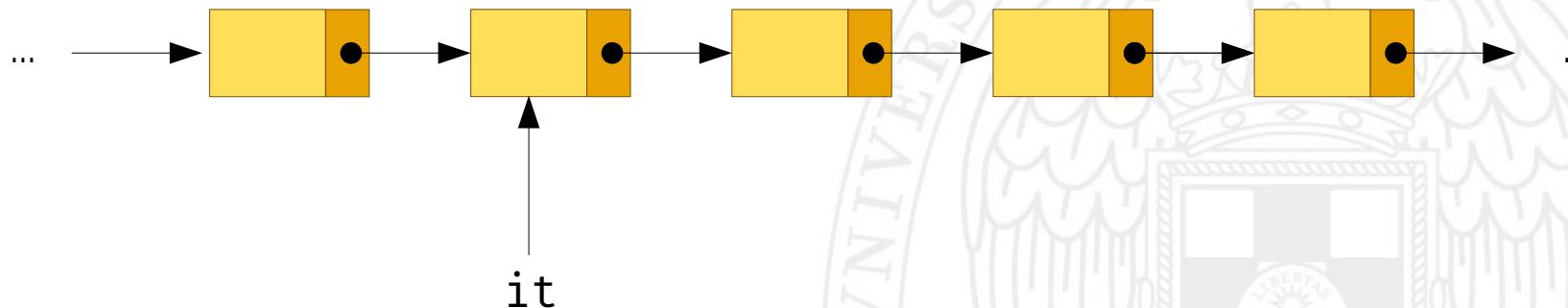
```
char *p;
```



# Moraleja

- En C++, los iteradores son generalizaciones de punteros.

```
ListLinked<int>::iterator it;
```



# Moraleja

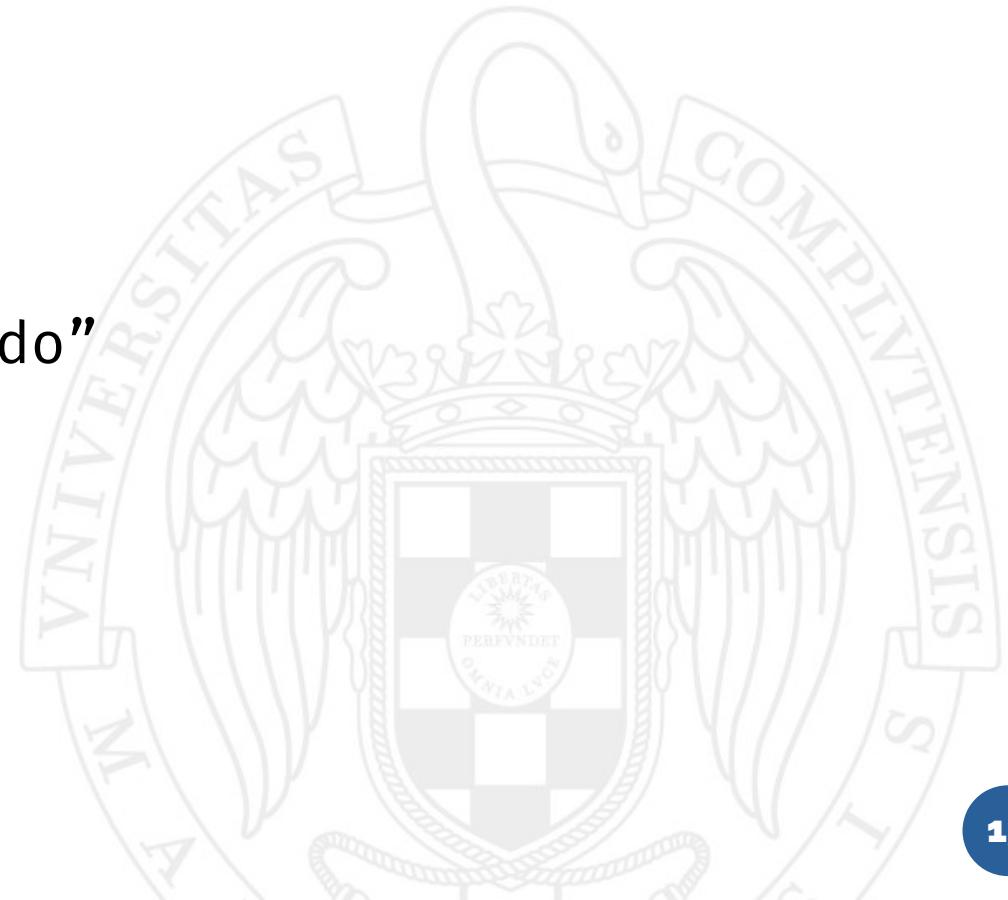
- En C++, los iteradores son generalizaciones de punteros.

```
string::iterator it;
```

“Hola, mundo”



it



# Moraleja

- En C++, los iteradores son generalizaciones de punteros.

```
std::string cadena = "Hola, mundo";
for (auto it = cadena.begin(); it != cadena.end(); ++it) {
    std::cout << *it << std::endl;
}
```

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

# Introducción a los árboles

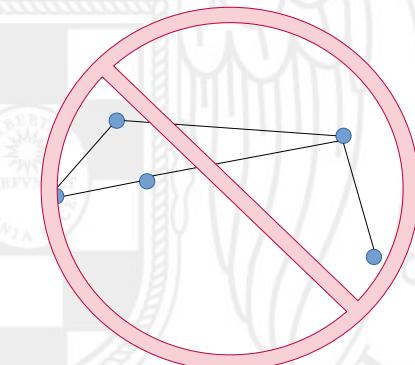
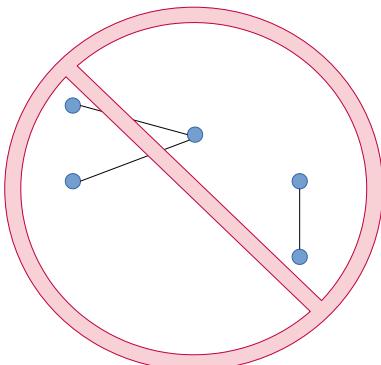
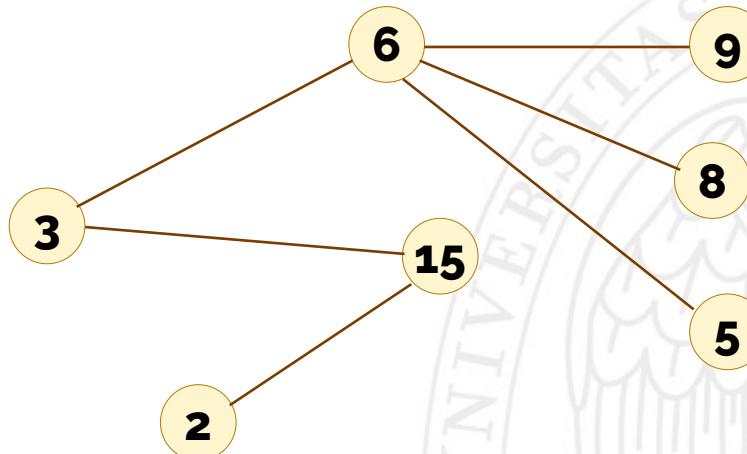
Manuel Montenegro Montes

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

# ¿Qué es un árbol?

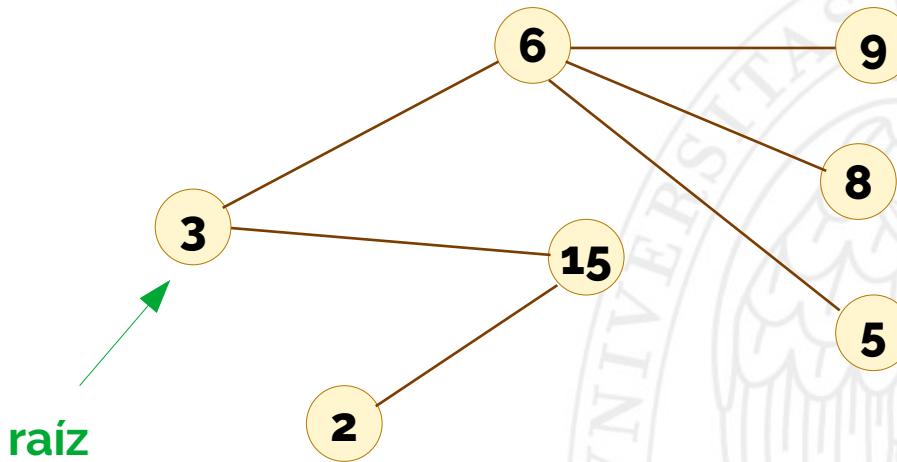
# El TAD Árbol

- Un árbol es un grafo **conexo** y **sin ciclos**.
- Llamamos a sus vértices **nodos**.



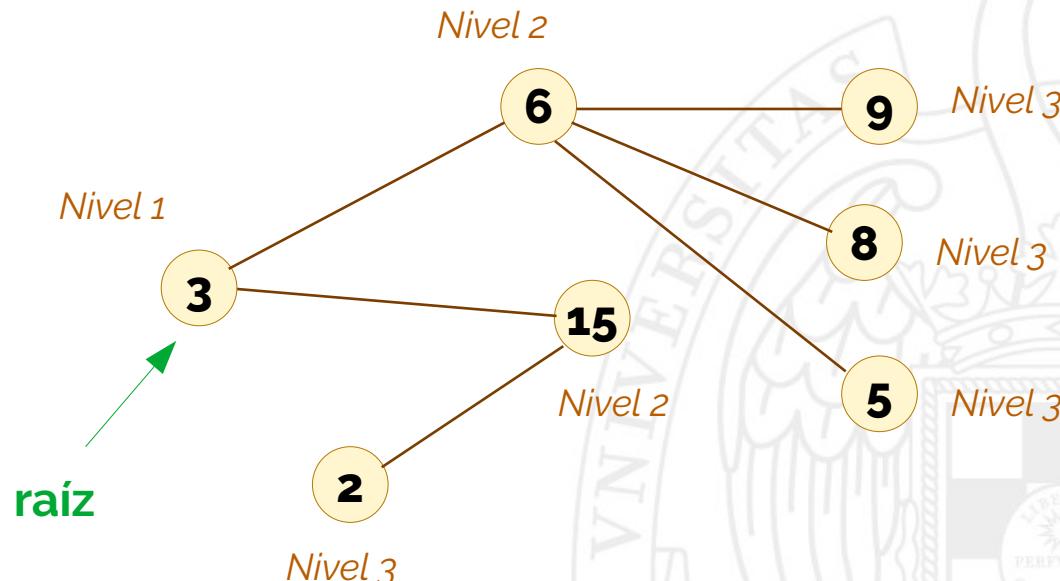
# Árboles con raíz

- Distinguimos un nodo en particular, que es la **raíz** del árbol.

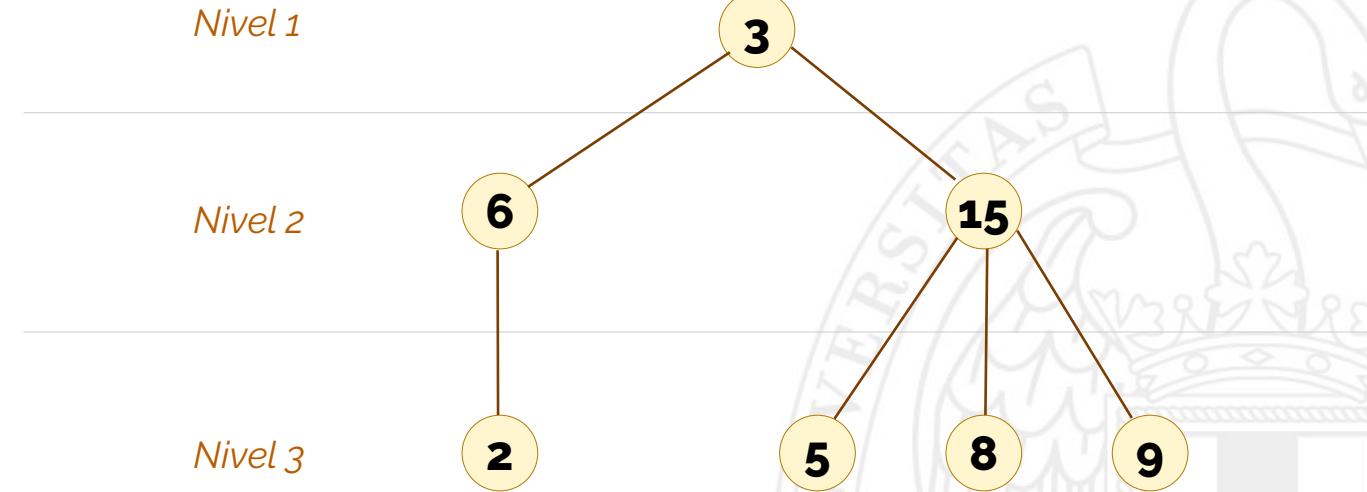


# Nivel de un nodo

- El **nivel** de un nodo se define como el número de aristas que lo separan de la raíz incrementado en 1.
- La raíz está en el nivel 1.

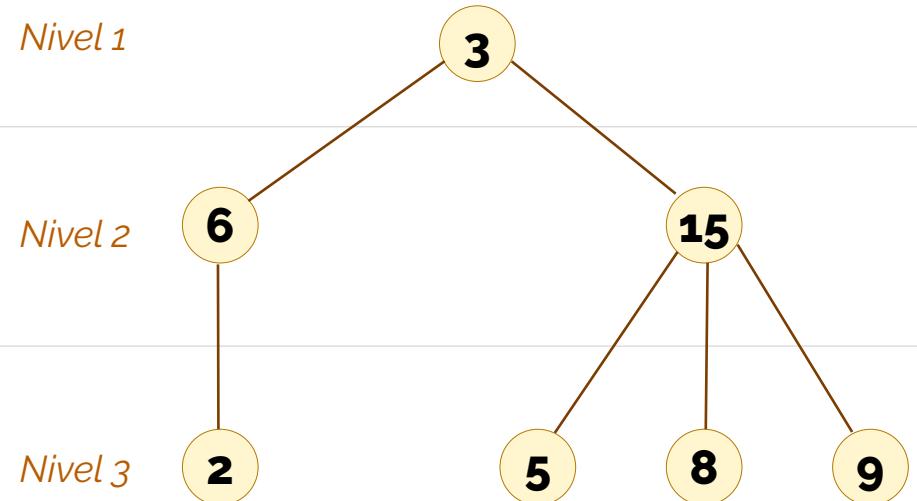


# Nivel de un nodo



# Definiciones

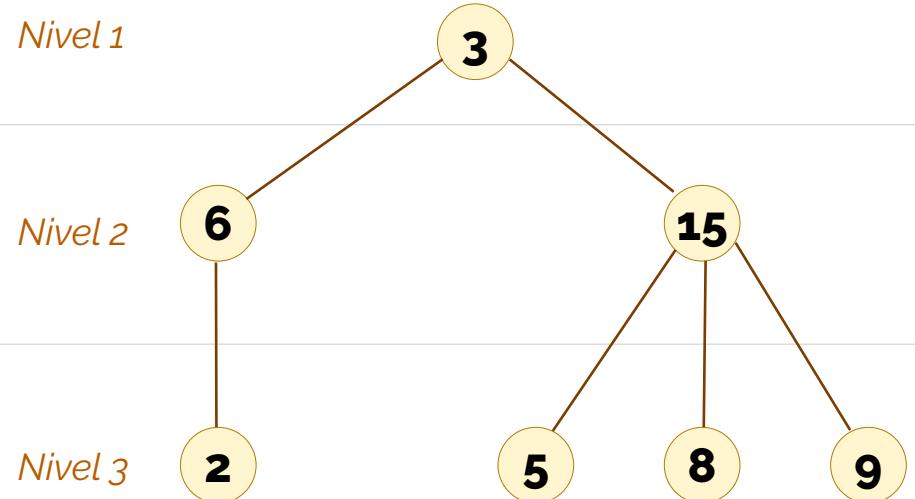
# Padres e hijos



- Si **X** es un nodo que está a nivel  $n$ , su **padre** es el que está conectado con él en el nivel  $n-1$ .



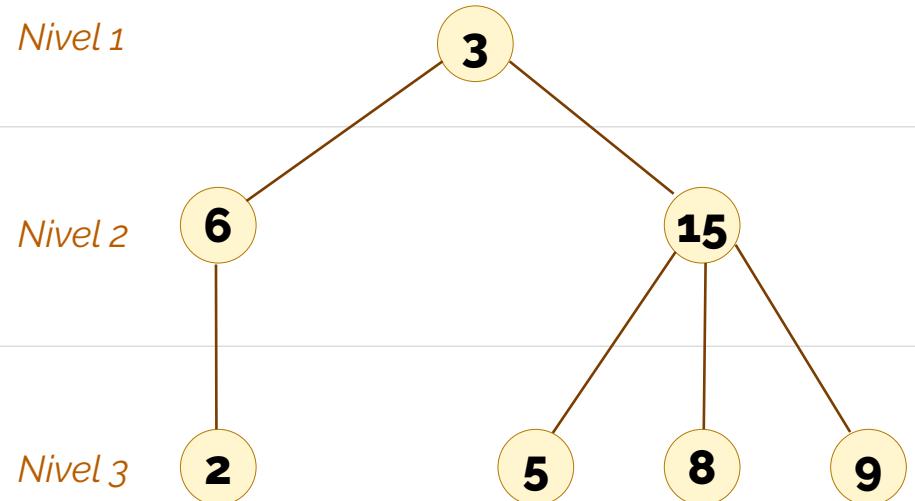
# Padres e hijos



- Si **X** es un nodo que está a nivel  $n$ , sus **hijos** son aquellos conectados con él en el nivel  $n+1$ .



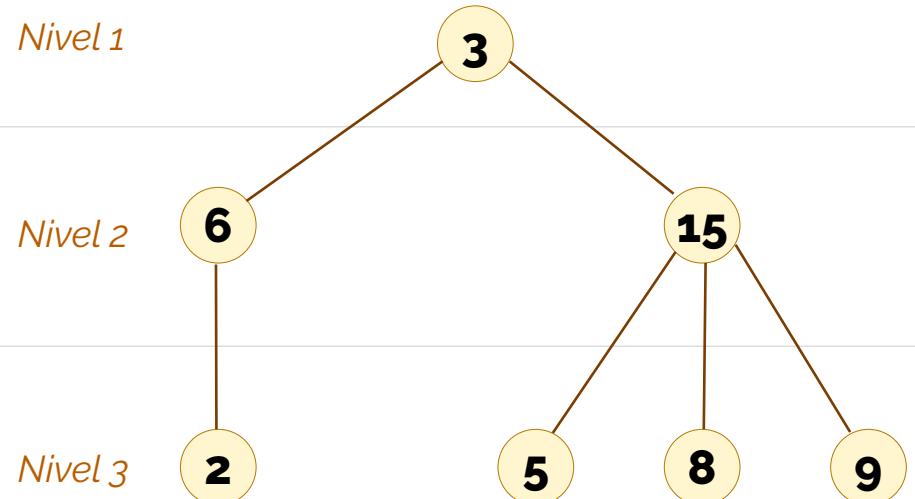
# Hermanos



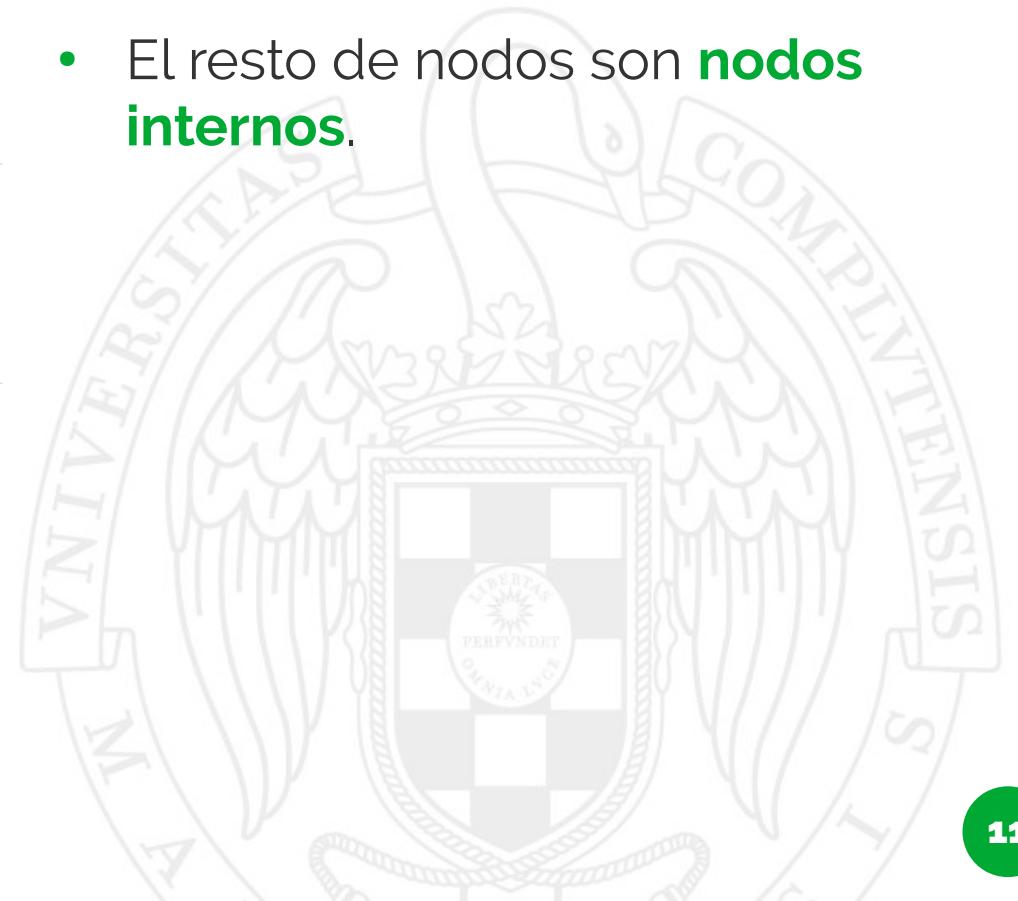
- Dos nodos son **hermanos** si tienen el mismo padre.



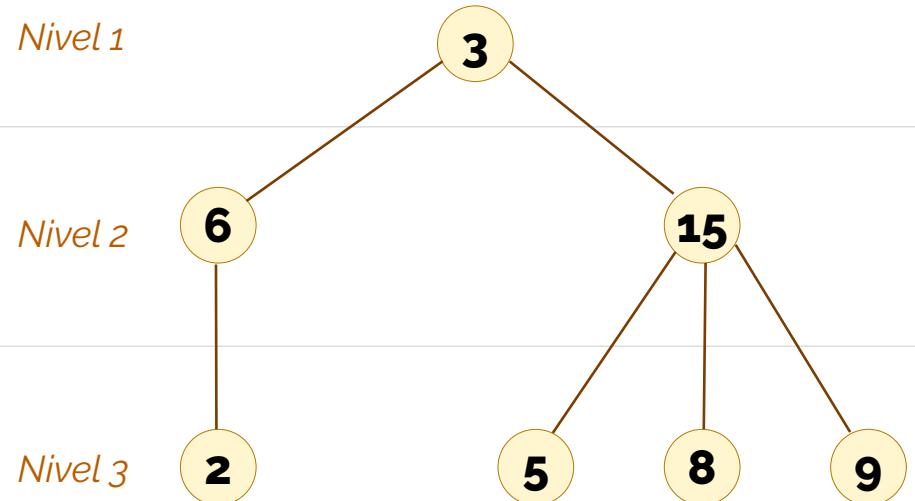
# Hojas vs nodos internos



- Una **hoja** es un nodo que no tiene hijos.
- El resto de nodos son **nodos internos**.

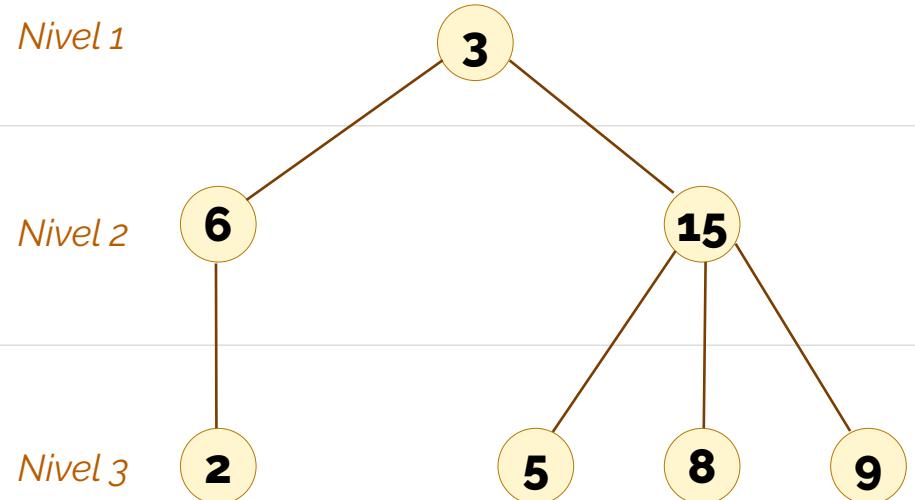


# Caminos y longitud



- Un **camino** es una sucesión de nodos en la que cada nodo es padre del siguiente.
- La **longitud de un camino** es el número de nodos que hay en él.

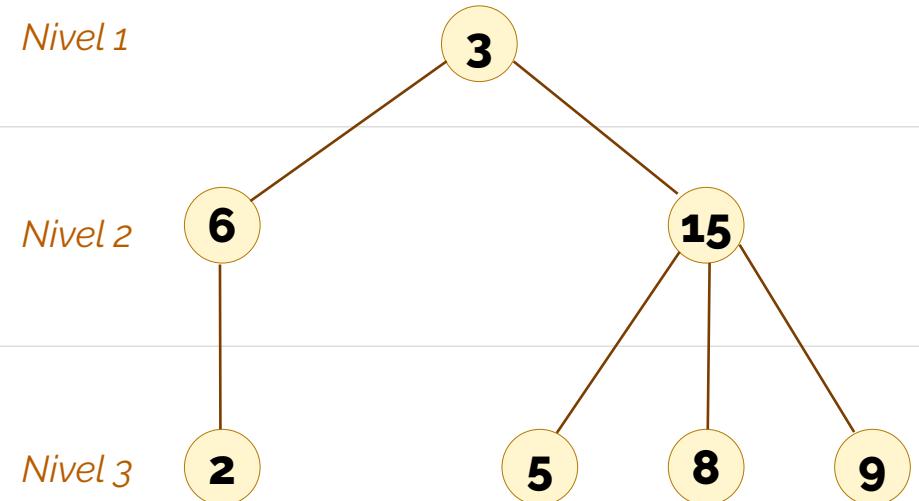
# Ramas



- Si un camino empieza en la raíz y termina en una hoja, decimos que es una **rama**.

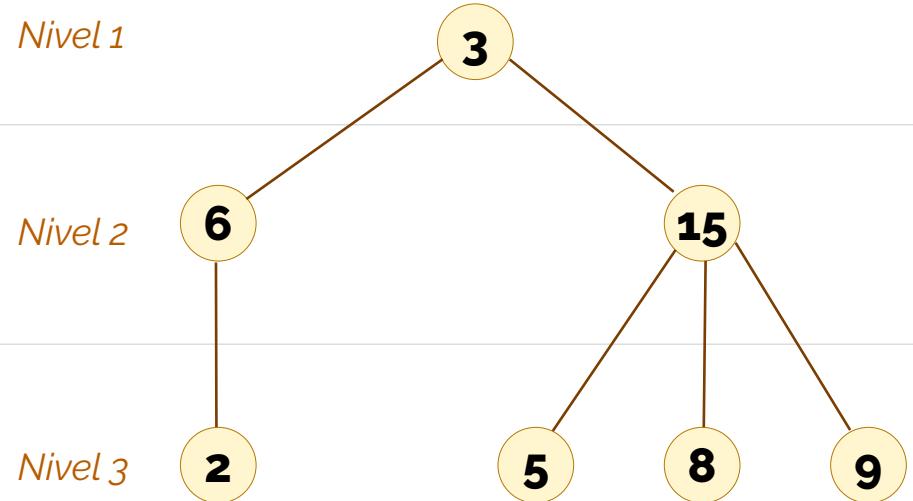


# Antepasados y descendientes



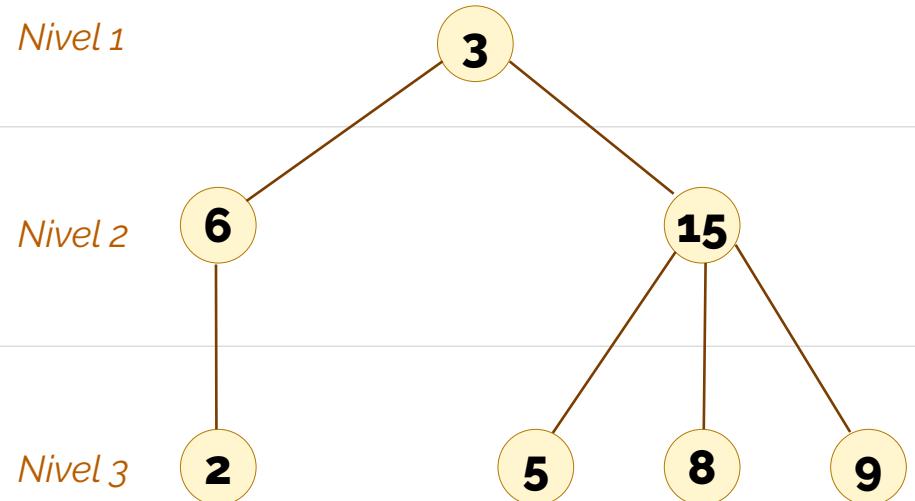
- Decimos que X es **antepasado** de Y si existe un camino de X a Y.
- Decimos que Y es **descendiente** de X si existe un camino de X a Y.

# Altura



- La **altura** de un árbol es el máximo de los niveles de los nodos.
- Equivalentemente, es la longitud de la rama más larga.

# Grado o aridad

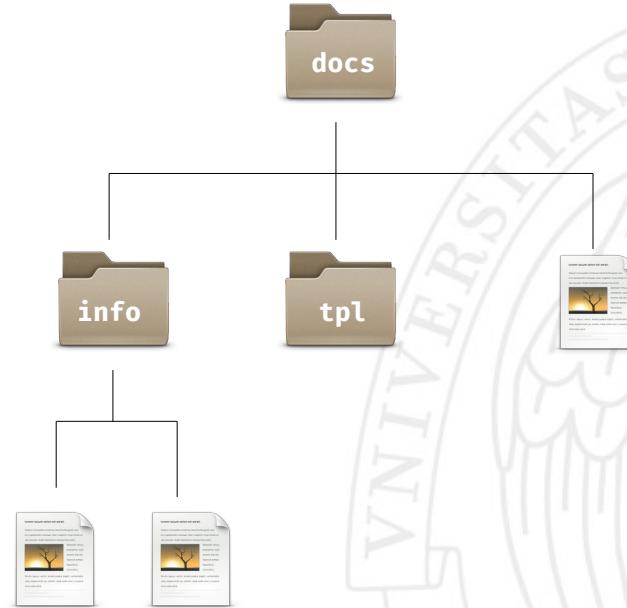


- El **grado** (o aridad) de un **nodo** es el número de hijos que tiene.
- La **aridad** de un **árbol** es el máximo de los grados de los nodos.

# Aplicaciones en un árbol

# Aplicaciones de los árboles

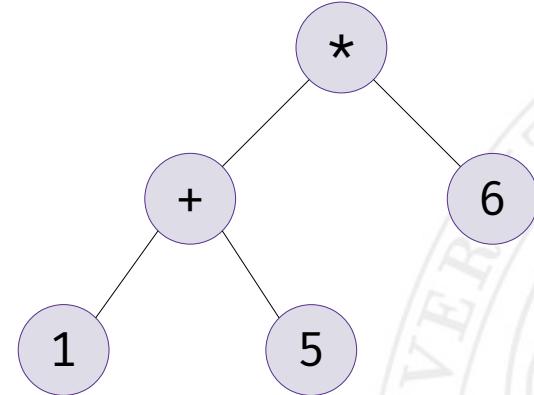
- Los árboles se utilizan para representar datos que están jerarquizados de alguna manera, o se contienen unos a otros.



# Aplicaciones de los árboles

- Los árboles se utilizan para representar datos que están jerarquizados de alguna manera, o se contienen unos a otros.

“(1 + 5) \* 6”



# Aplicaciones de los árboles

- Los árboles se utilizan para representar datos que están jerarquizados de alguna manera, o se contienen unos a otros.

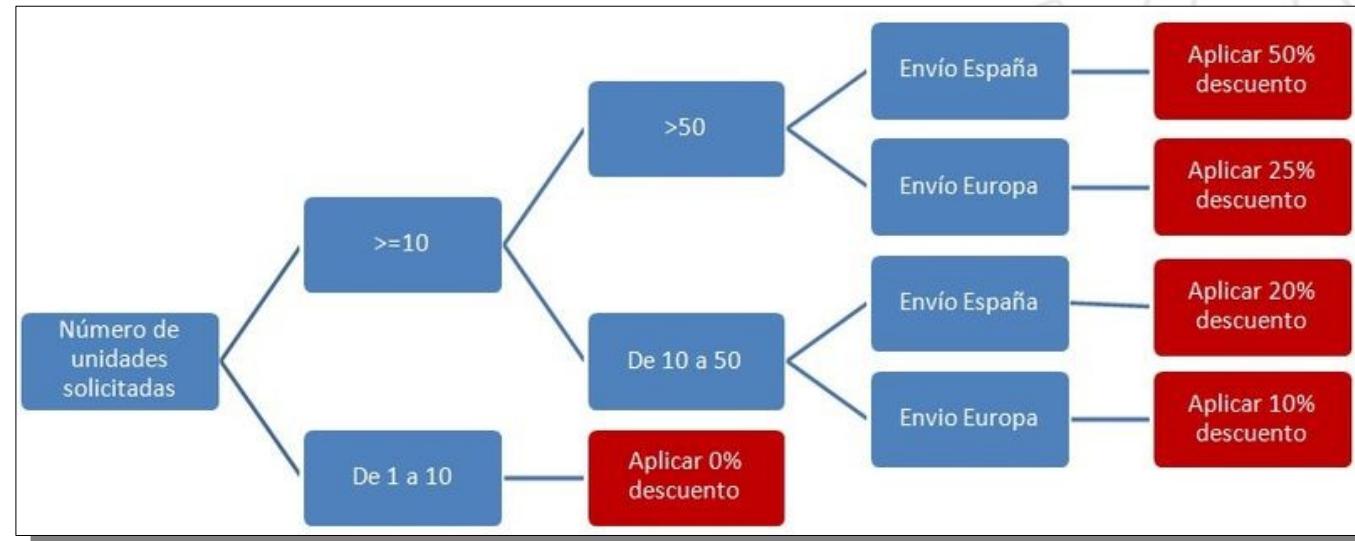


Imagen: Sargantano (CC BY-SA 3.0)

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

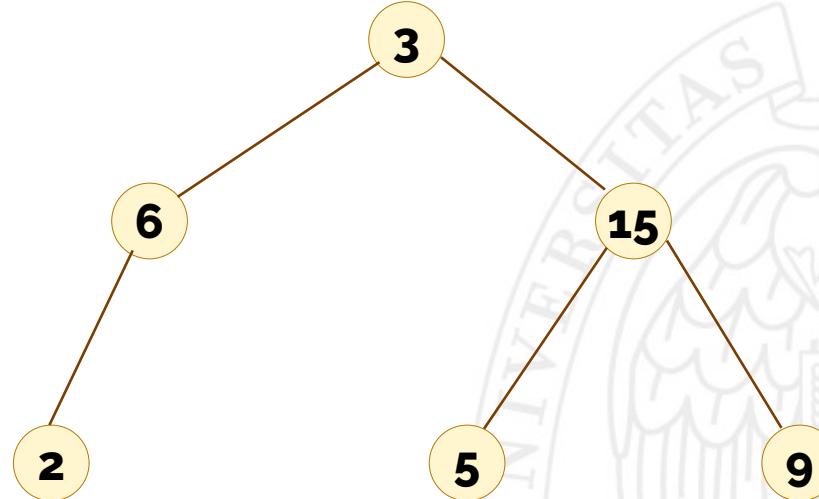
# EL TAD Árbol Binario

Manuel Montenegro Montes

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

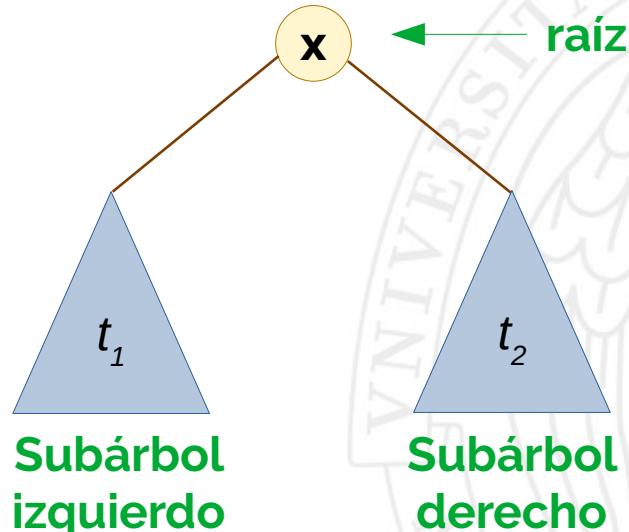
# Árboles binarios

- Un árbol binario es un árbol de aridad 2.
- Cada nodo tiene **2** hijos, algunos de los cuales pueden ser vacíos.



# Definición inductiva de un árbol binario

- Caso base: Un grafo sin nodos es un **árbol vacío**.
- Caso recursivo: Si  $t_1$  y  $t_2$  son árboles binarios, y  $x$  es un elemento, entonces lo siguiente es un árbol binario:



Si  $t_1$  o  $t_2$  son vacíos,  
no existen aristas  
desde  $x$  hacia ellos.

# Definición inductiva de un árbol binario

- Un árbol binario  $T$  es un conjunto finito tal que:
  - $T = \emptyset$ , o bien
  - $T = \{x\} \uplus T_1 \uplus T_2$ , donde  $T_1$  y  $T_2$  son árboles.  
 $x$  es la raíz,  
 $T_1$  es el subárbol izquierdo, y  
 $T_2$  es el subárbol derecho.



# Operaciones en el TAD Árbol Binario

- Constructoras:
  - Crear un árbol vacío: ***create\_empty***.
  - Crear una hoja: ***create\_leaf***.
  - Crear un árbol a partir de una raíz y dos hijos: ***create\_tree***.
- Observadoras:
  - Determinar si el árbol es vacío: ***empty***.
  - Obtener la raíz si el árbol no es vacío: ***root***.
  - Obtener el hijo izquierdo, si existe: ***left***.
  - Obtener el hijo derecho, si existe: ***right***.

# Operaciones constructoras

{ *true* }

***create\_empty()*** → ( $T$ : ArBin)

{  $T = \text{—}$  }

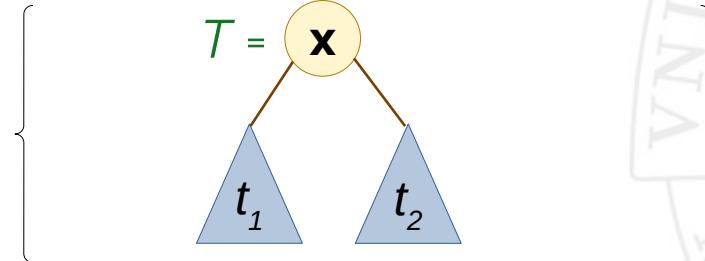
{ *true* }

***create\_leaf( $x$ : Elem)*** → ( $T$ : ArBin)

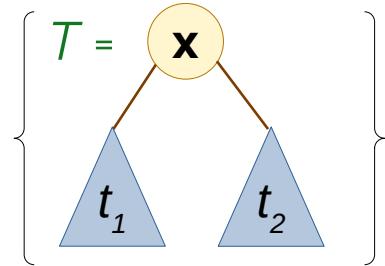
{  $T = \text{x}$  }

{  $T_1 = t_1$        $T_2 = t_2$  }

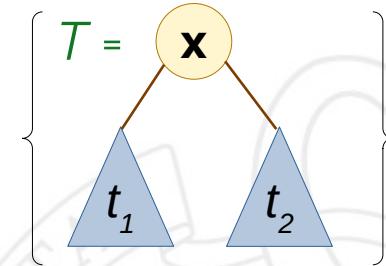
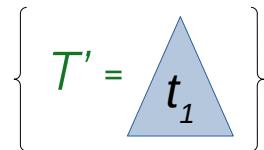
***create\_tree( $T_1$ : ArBin,  $x$ : Elem,  $T_2$ : ArBin)*** → ( $T$ : ArBin)



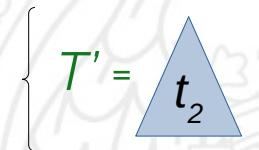
# Operaciones observadoras



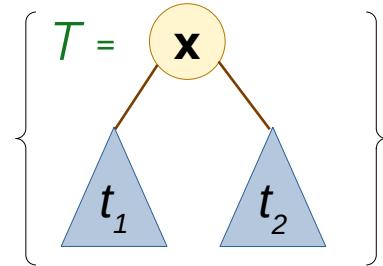
**left**( $T : \text{ArBin}$ )  $\rightarrow (T' : \text{ArBin})$



**right**( $T : \text{ArBin}$ )  $\rightarrow (T' : \text{ArBin})$



# Operaciones observadoras



**root**( $T: \text{ArBin}$ )  $\rightarrow (e: \text{elem})$

$$\left\{ e = \text{x} \right\}$$

{ true }

**empty**( $T: \text{ArBin}$ )  $\rightarrow (b: \text{bool})$

$$\left\{ b \Leftrightarrow T = \text{---} \right\}$$

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

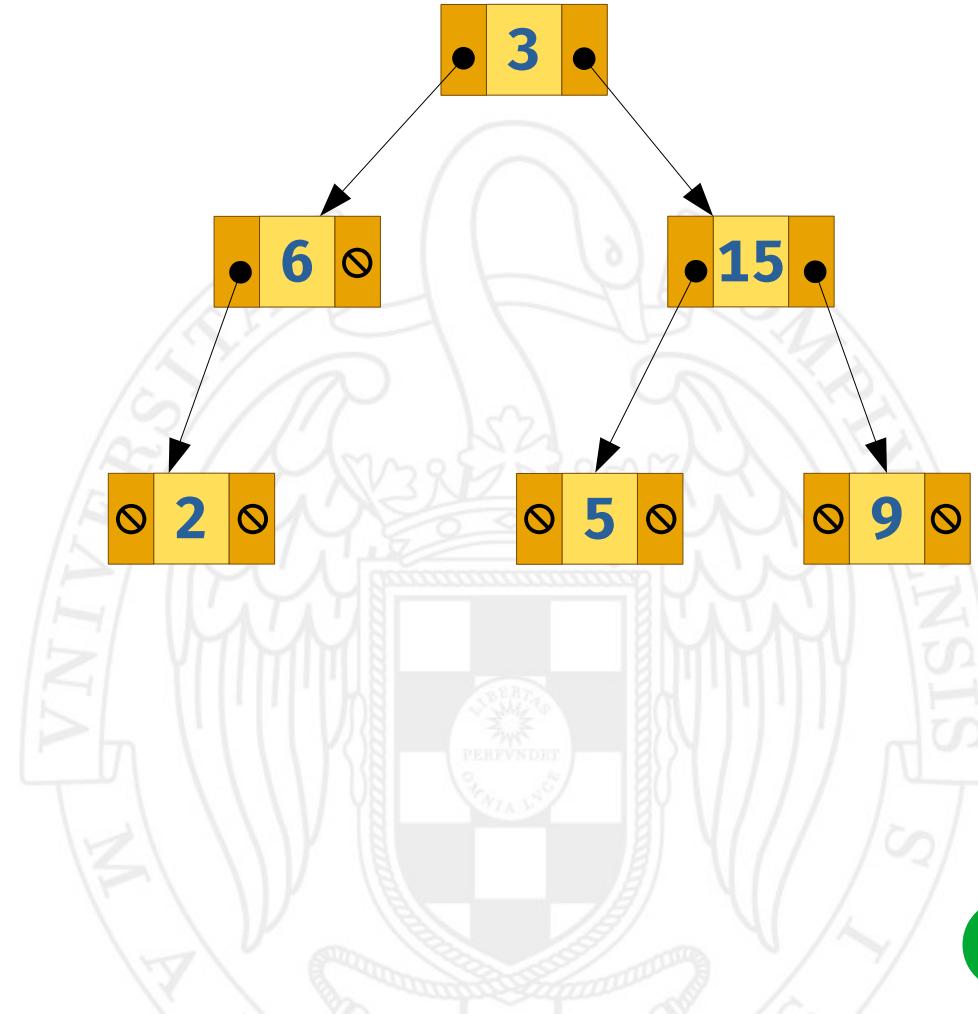
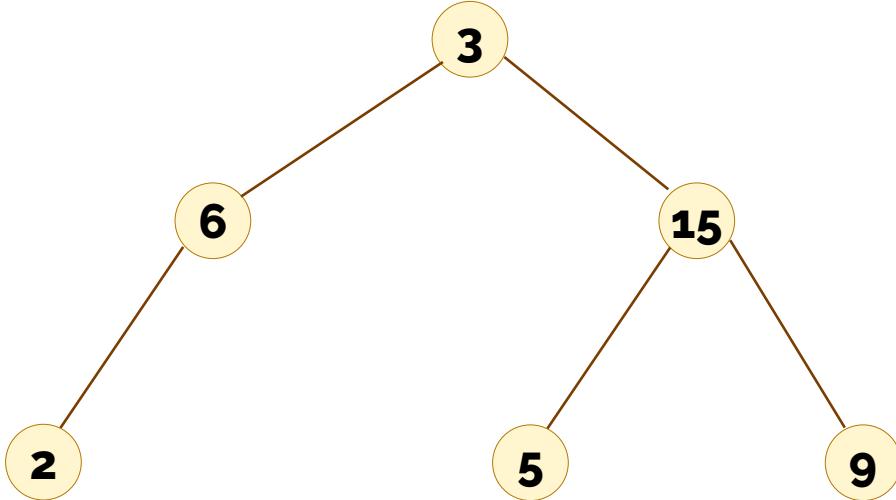
# Implementación de árboles binarios

Manuel Montenegro Montes

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

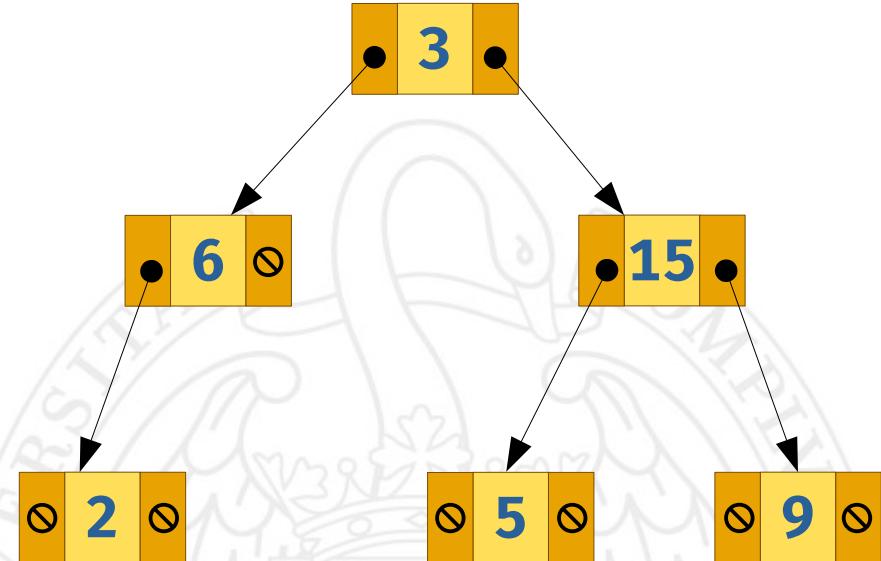
# Representación mediante nodos

# Representando árboles binarios



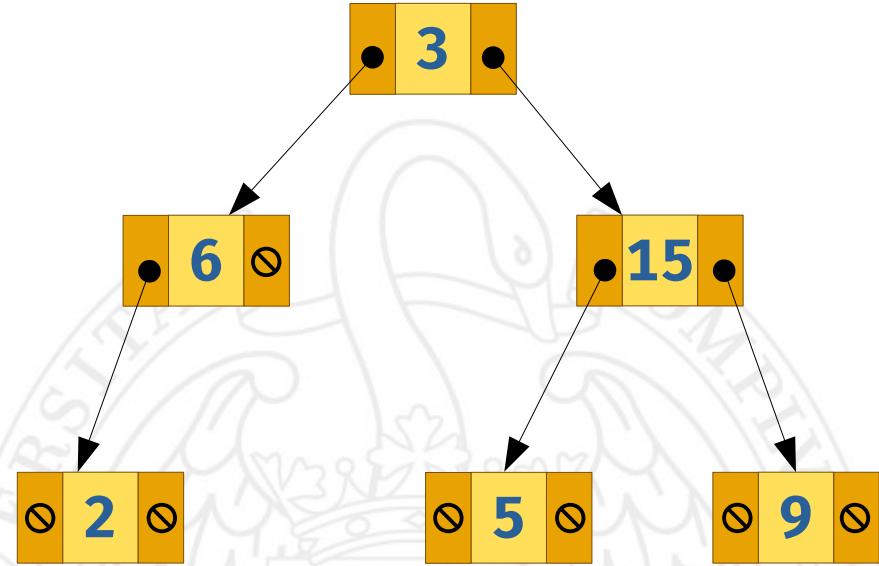
# Representando árboles binarios

```
struct TreeNode {  
    T elem;  
    TreeNode *left, *right;  
};
```



# Representando árboles binarios

```
struct TreeNode {  
    T elem;  
    TreeNode *left, *right;  
  
    TreeNode(const TreeNode *left,  
             const T &elem,  
             const TreeNode *right)  
        : elem(elem), left(left),  
              right(right) { }  
};
```

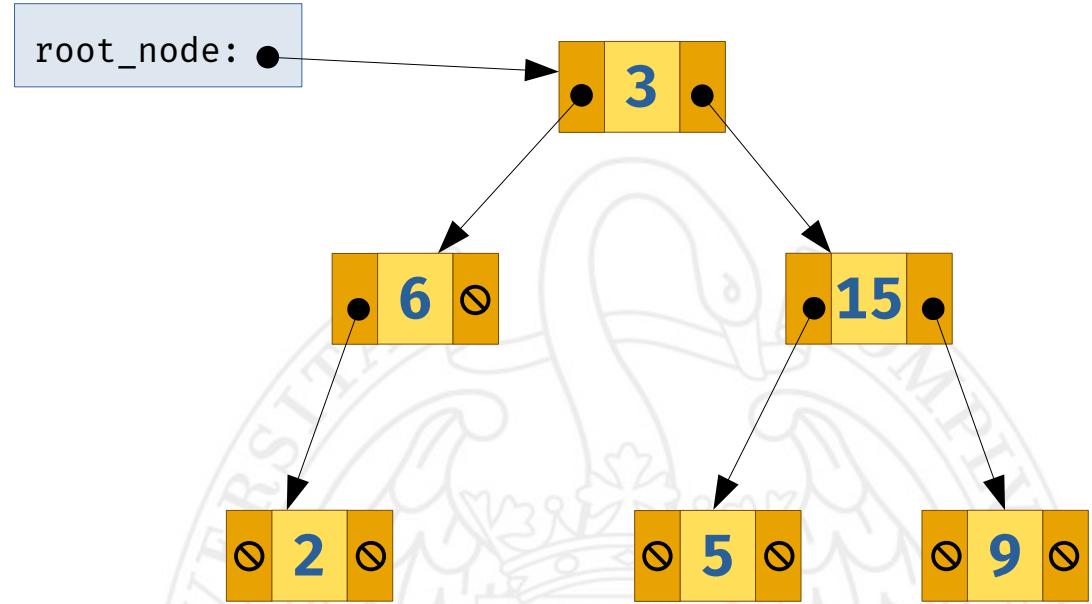


# La clase BinTree

```
template<class T>
class BinTree {
public:
    ...
private:
    struct TreeNode { ... }
    TreeNode *root_node;
};
```

En el caso en que el árbol es vacío:

```
root_node = nullptr
```



# Operaciones básicas

# Operaciones en el TAD Árbol Binario

- Constructoras:
  - Crear un árbol vacío: ***create\_empty***.
  - Crear una hoja: ***create\_leaf***.
  - Crear un árbol a partir de una raíz y dos hijos: ***create\_tree***.
- Observadoras:
  - Determinar si el árbol es vacío: ***empty***.
  - Obtener la raíz si el árbol no es vacío: ***root***.
  - Obtener el hijo izquierdo, si existe: ***left***.
  - Obtener el hijo derecho, si existe: ***right***.

# Interfaz de la clase BinTree

```
template<class T>
class BinTree {
public:
    BinTree();
    BinTree(const T &elem);
    BinTree(const BinTree &left, const T &elem, const BinTree &right);

    const T & root() const;
    BinTree left() const;
    BinTree right() const;
    bool empty() const;

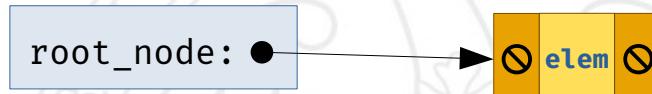
private:
    struct TreeNode { ... }
    TreeNode *root_node;
};
```

# Creación de árboles

```
template<class T>
class BinTree {
public:
    BinTree(): root_node(nullptr) { }

    BinTree(const T &elem)
        : root_node(new TreeNode(nullptr, elem, nullptr)) { }

private:
    struct TreeNode { ... }
    TreeNode *root_node;
};
```



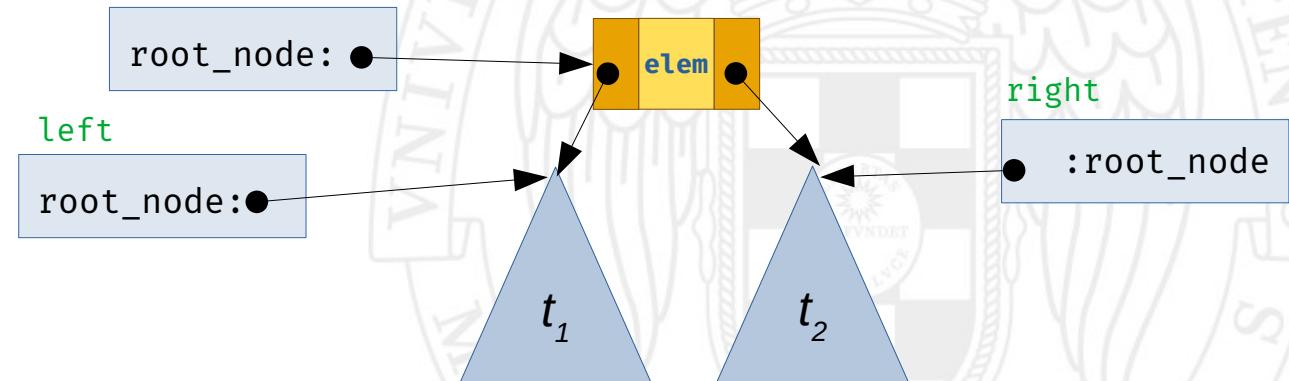
# Creación de árboles

```
template<class T>
class BinTree {
public:
    BinTree(): root_node(nullptr) { }

    BinTree(const T &elem)
        : root_node(new TreeNode(nullptr, elem, nullptr)) { }

    BinTree(const BinTree &left, const T &elem, const BinTree &right)
        : root_node(new TreeNode(left.root_node, elem, right.root_node)) { }

private:
    struct TreeNode { ... }
    TreeNode *root_node;
};
```

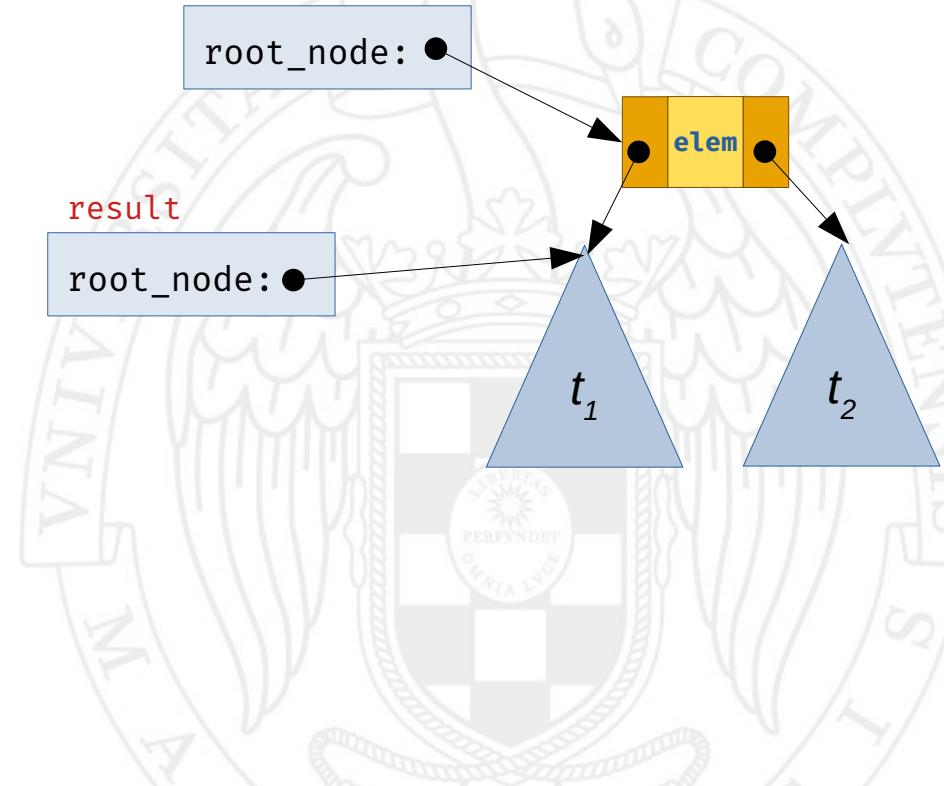


# Operaciones observadoras

```
template<class T>
class BinTree {
public:
    ...
    const T & root() const {
        assert(root_node != nullptr);
        return root_node->elem;
    }

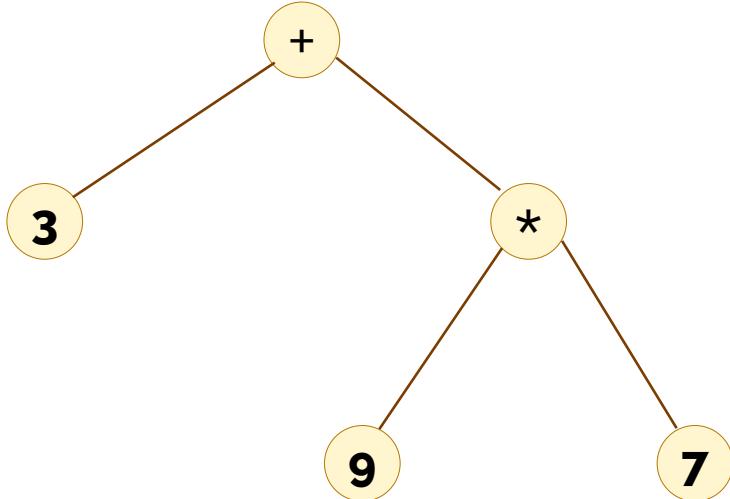
    BinTree left() const {
        assert (root_node != nullptr);
        BinTree result;
        result.root_node = root_node->left;
        return result;
    }

    bool empty() const {
        return root_node == nullptr;
    }
};
```



# E/S de árboles

# Representación textual de un árbol



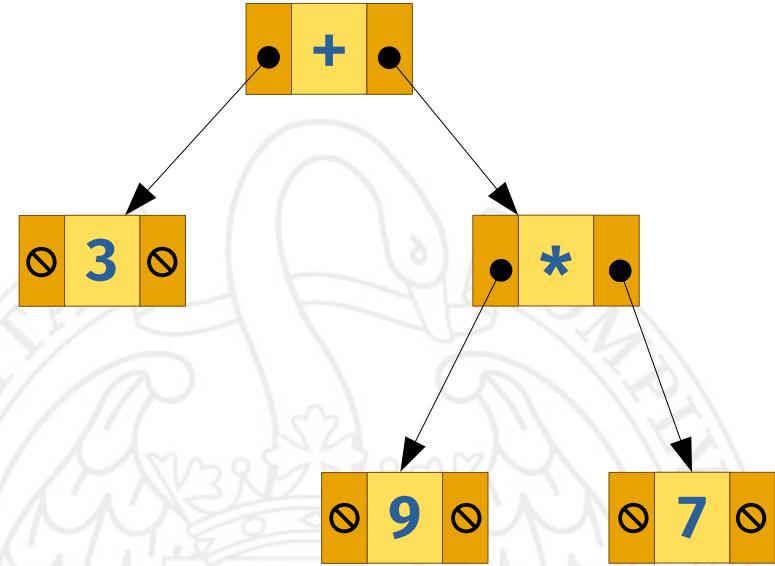
((. 3 .) + ((. 9 .) \* (. 7 .)))

- Árbol vacío: .
- Árbol no vacío: (*hijo-iz raiz hijo-dr*)

# Mostrar un árbol por pantalla

```
template<class T>
class BinTree {
    ...
private:
    struct TreeNode { ... }
    TreeNode *root_node;

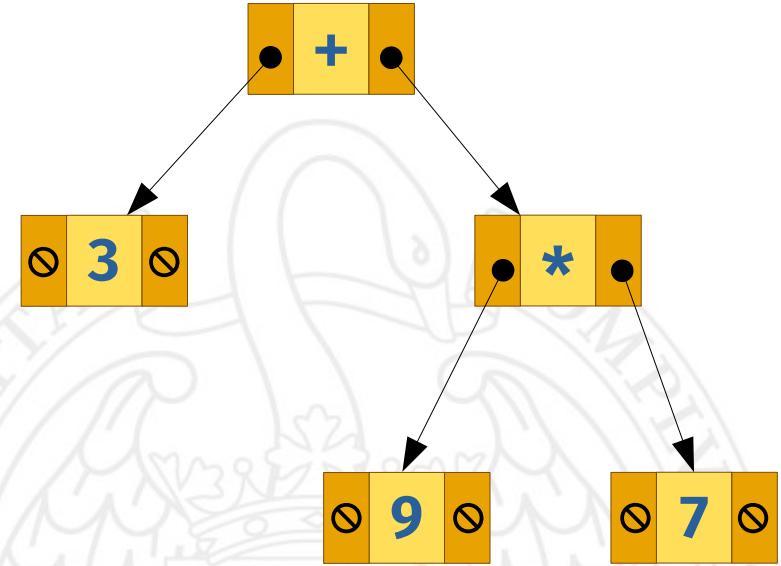
    static void display_node(const TreeNode *root,
                           std::ostream &out) {
        if (root == nullptr) {
            out << ".";
        } else {
            out << "(";
            display_node(root->left, out);
            out << " " << root->elem << " ";
            display_node(root->right, out);
            out << ")";
        }
    }
};
```



# Mostrar un árbol por pantalla

```
template<class T>
class BinTree {
public:
    ...
    void display(std::ostream &out) const {
        display_node(root_node, out);
    }
private:
    TreeNode *root_node;
};

template<typename T>
std::ostream & operator<<(std::ostream &out, const BinTree<T> &tree) {
    tree.display(out);
    return out;
}
```



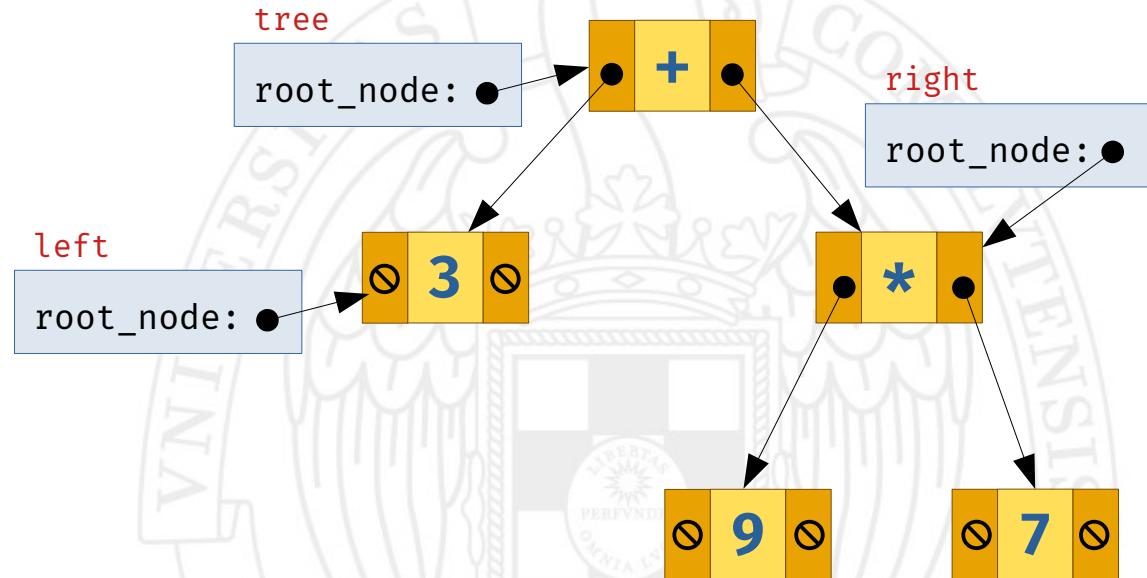
# Ejemplo

```
int main() {
    BinTree<std::string> left("3");
    BinTree<std::string> right(BinTree<std::string>("9"), "*", BinTree<std::string>("7"));
    BinTree<std::string> tree(left, "+", right);

    std::cout << tree << std::endl;

    return 0;
}
```

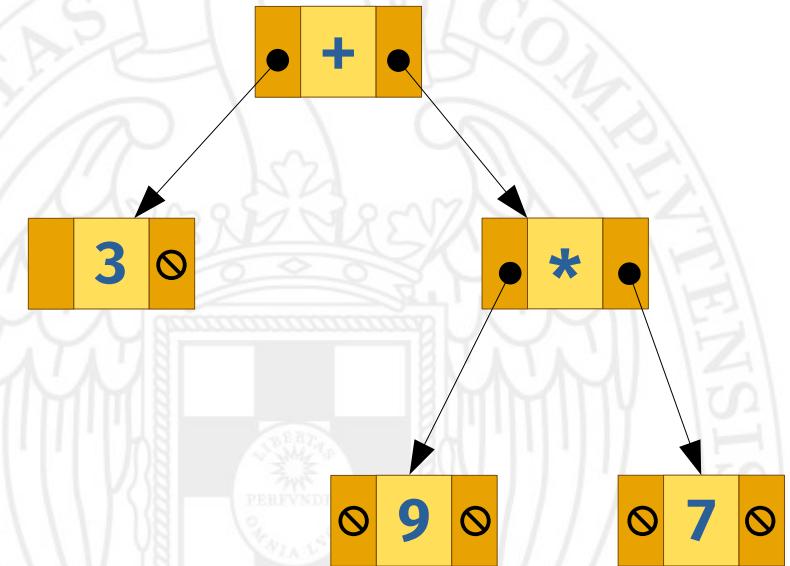
((. 3 .) + ((. 9 .) \* (. 7 .)))



# Ejemplo

```
int main() {
    BinTree<std::string> tree = {{"3"} , "+", {{"9"} , "*", {"7"} } };
    std::cout << tree << std::endl;
    return 0;
}
```

((. 3 .) + ((. 9 .) \* (. 7 .)))



# Leer un árbol por entrada

```
template<typename T>
BinTree<T> read_tree(std::istream &in) {
    char c;
    in >> c;
    if (c == '.') {
        return BinTree<T>();
    } else {
        assert (c == '(');
        BinTree<T> left = read_tree<T>(in);
        T elem;
        in >> elem;
        BinTree<T> right = read_tree<T>(in);
        in >> c;
        assert (c == ')');
        BinTree<T> result(left, elem, right);
        return result;
    }
}
```

((. 3 .) + ((. 9 .) \* (. 7 .))))

# Destrucción de memoria

# Problema importante

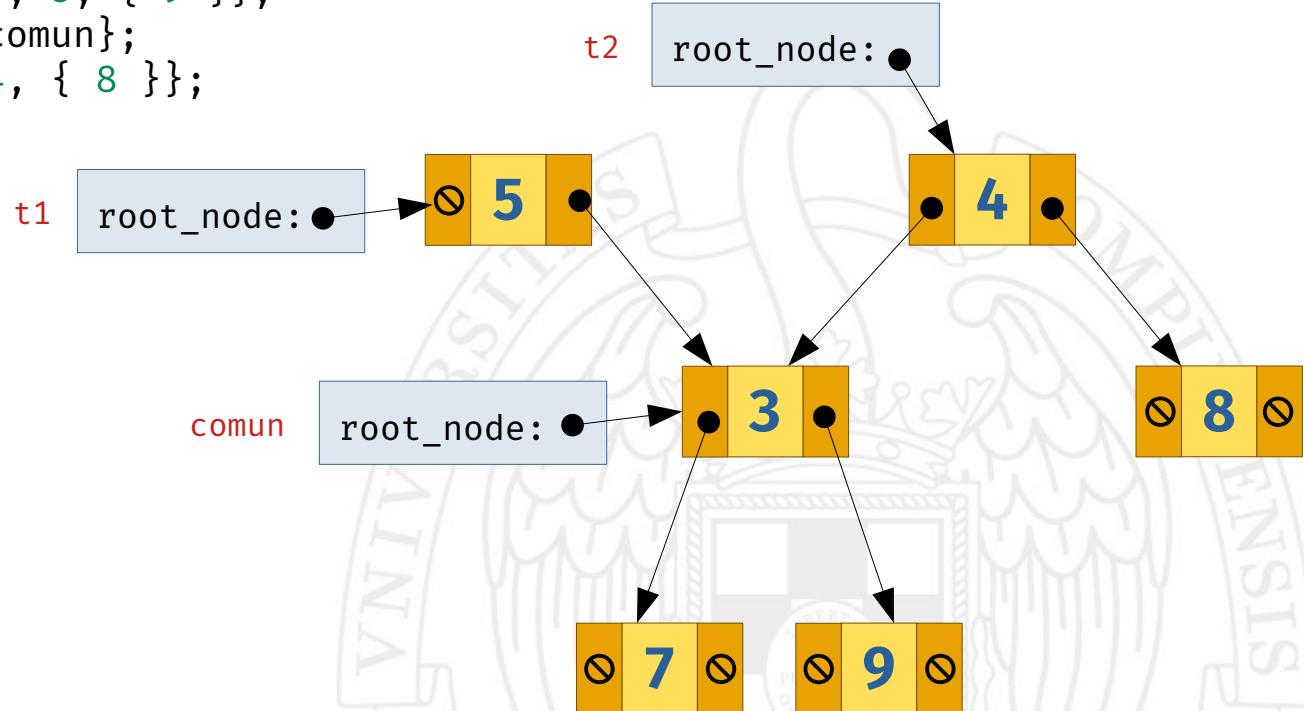
- ¡No estamos liberando la memoria ocupada por los nodos!
- Hay que hacerlo con cuidado...



# El problema de la compartición en árboles

```
BinTree<int> comun = {{ 7 }, 3, { 9 }};
BinTree<int> t1 = {{}, 5, comun};
BinTree<int> t2 = {comun, 4, { 8 }};
```

¿Cómo liberamos la memoria ocupada por los nodos?



# Intento fallido de destructor

```
template<class T>
class BinTree {
public:
...
~BinTree() {
    delete_with_children(root_node);
}

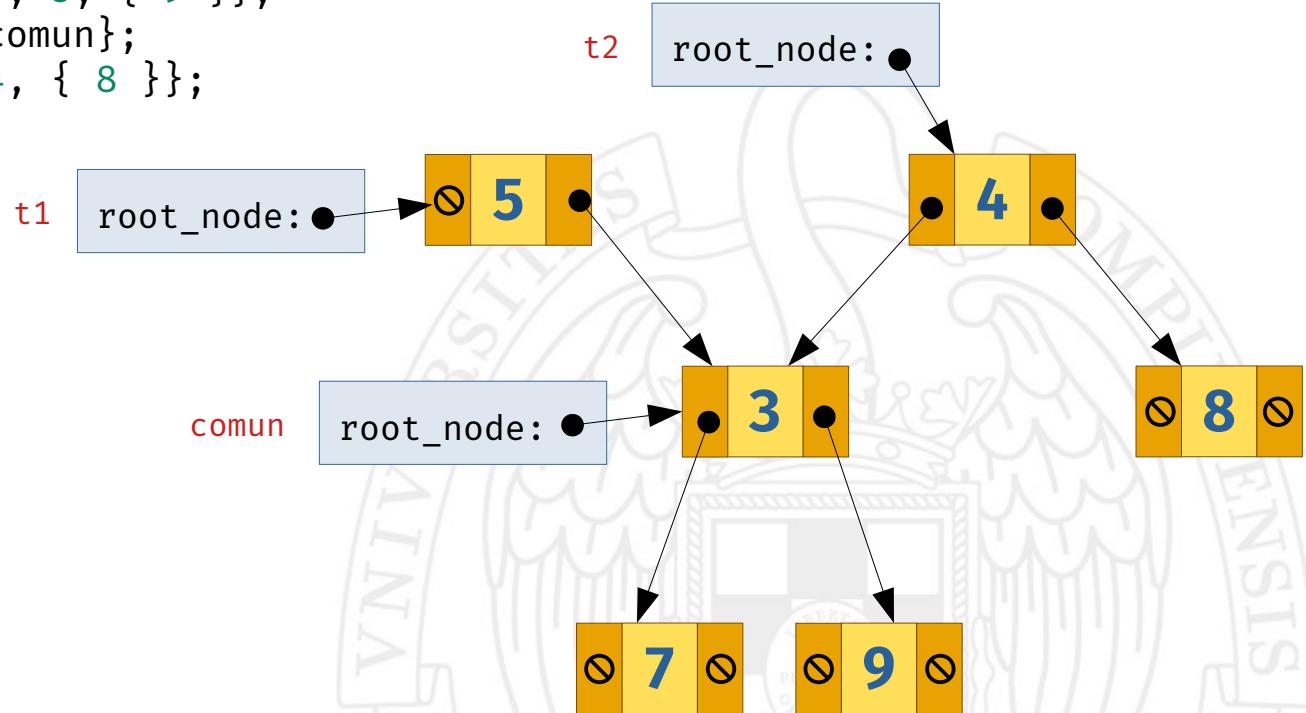
private:
    static void delete_with_children(const TreeNode *node) {
        if (node != nullptr) {
            delete_with_children(node->left);
            delete_with_children(node->right);
            delete node;
        }
    }
};
```



# En nuestro ejemplo...

```
BinTree<int> comun = {{ 7 }, 3, { 9 }};
BinTree<int> t1 = {{}, 5, comun};
BinTree<int> t2 = {comun, 4, { 8 }};
```

¿Qué pasa cuando t1, t2 y comun salen de ámbito?



# Soluciones

Para evitar liberar nodos más de una vez, podemos optar por alguna de las siguientes alternativas:

- 1) *Evitar la compartición de nodos entre árboles.*

Ejercicio

Cada vez que construyamos un árbol a partir de otros, debemos hacer una copia de los nodos de estos últimos.

- 2) *Aceptar la compartición de nodos entre árboles.*

Otro video

Utilizamos mecanismos de conteo de referencias para saber cuándo liberar la memoria.

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

# Compartición en árboles binarios

Manuel Montenegro Montes

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

# Definición actual de TreeNode

```
struct TreeNode {  
    T elem;  
    TreeNode *left, *right;  
  
    TreeNode(const TreeNode *left,  
             const T &elem,  
             const TreeNode *right)  
        : elem(elem), left(left),  
              right(right) {}  
};
```

- Vamos a sustituir los punteros estándar de C++ por *smart pointers*.
- En lugar de `TreeNode *` utilizamos `std::shared_ptr<TreeNode>`

# Cambios en TreeNode

```
struct TreeNode {  
    T elem;  
    std::shared_ptr<TreeNode> left, right;  
  
    TreeNode(const std::shared_ptr<TreeNode> &left,  
             const T &elem,  
             const std::shared_ptr<TreeNode> &right)  
        : elem(elem), left(left),  
          right(right) {}  
};
```



# Cambios en TreeNode

```
using NodePointer = std::shared_ptr<TreeNode>;  
  
struct TreeNode {  
    T elem;  
    NodePointer left, right;  
  
    TreeNode(const NodePointer &left,  
             const T &elem,  
             const NodePointer &right)  
        : elem(elem), left(left),  
              right(right) { }  
};
```



# Cambios en BinTree

```
template<class T>
class BinTree {
public:

    BinTree(): root_node(nullptr) { }
    BinTree(const T &elem)
        : root_node(std::make_shared<TreeNode>(nullptr, elem, nullptr)) { }
    BinTree(const BinTree &left, const T &elem, const BinTree &right)
        : root_node(std::make_shared<TreeNode>(left.root_node, elem, right.root_node)) { }

    ...

private:
    using NodePointer = std::shared_ptr<TreeNode>;
    struct TreeNode { ... }

    NodePointer root_node;

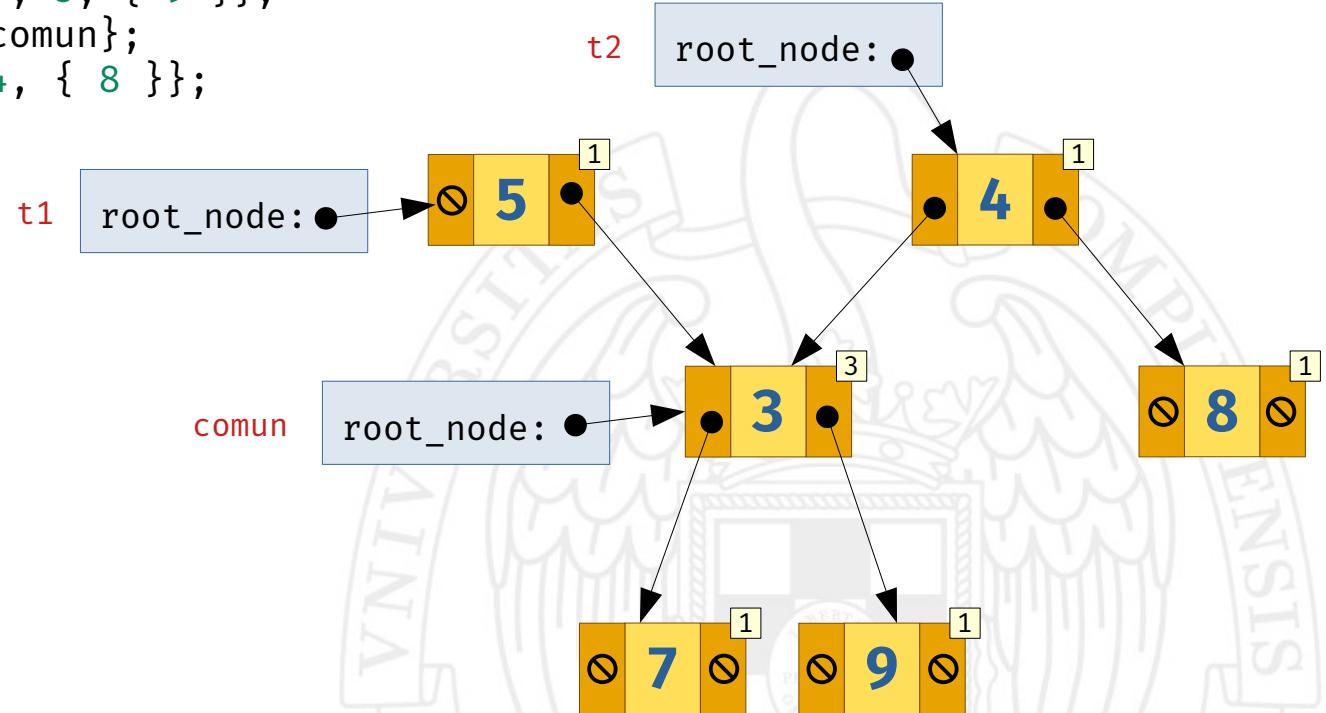
    static void display_node(const NodePointer &root, std::ostream &out) { ... }
};
```

# No necesitamos...

- Destructor
  - Cuando se elimina un objeto `BinTree` se llama automáticamente al destructor de `root_node`.
  - El destructor de `root_node` decrementa el contador de referencias del nodo raíz, y lo libera, en caso de llegar a 0.
- Constructor de copia
  - El constructor de copia por defecto `BinTree` nos sirve, ya que llama al constructor de copia de `root_node`.
  - El constructor de copia de `root_node` incrementa el contador de referencias del nodo raíz.

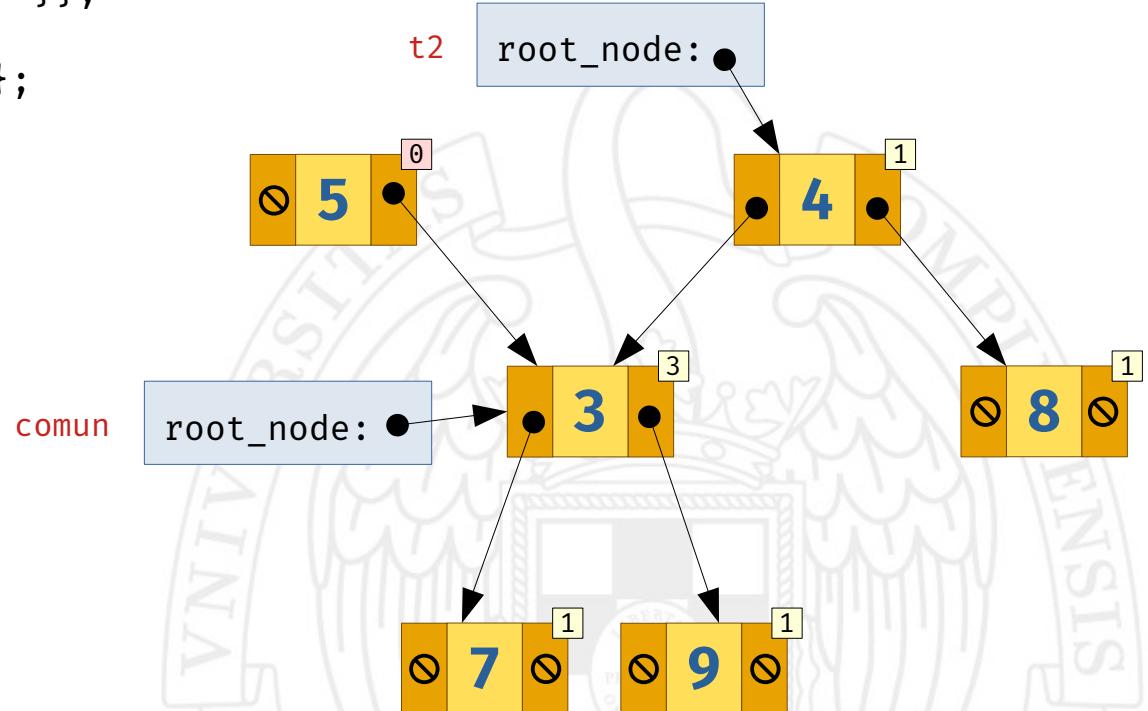
# Ejemplo

```
BinTree<int> comun = {{ 7 }, 3, { 9 }};
BinTree<int> t1 = {{}, 5, comun};
BinTree<int> t2 = {comun, 4, { 8 }};
```



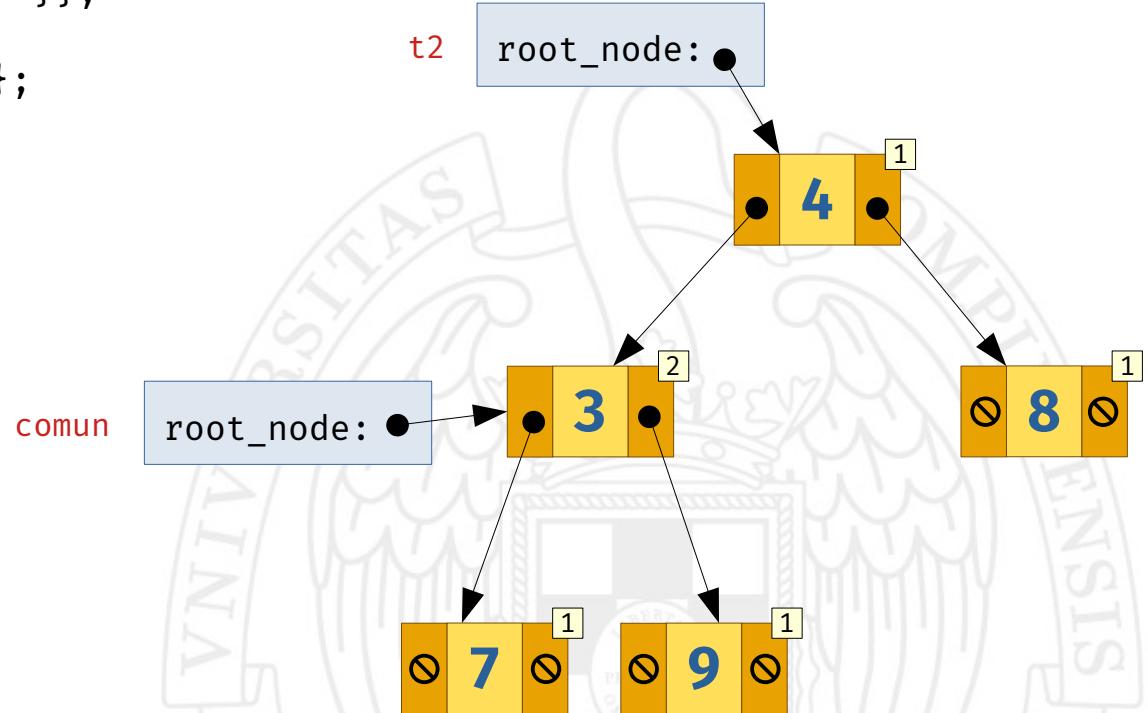
# Ejemplo

```
BinTree<int> comun = {{ 7 }, 3, { 9 }};
BinTree<int> t1 = {{}, 5, comun};
BinTree<int> t2 = {comun, 4, { 8 }};
```



# Ejemplo

```
BinTree<int> comun = {{ 7 }, 3, { 9 }};
BinTree<int> t1 = {{}, 5, comun};
BinTree<int> t2 = {comun, 4, { 8 }};
```

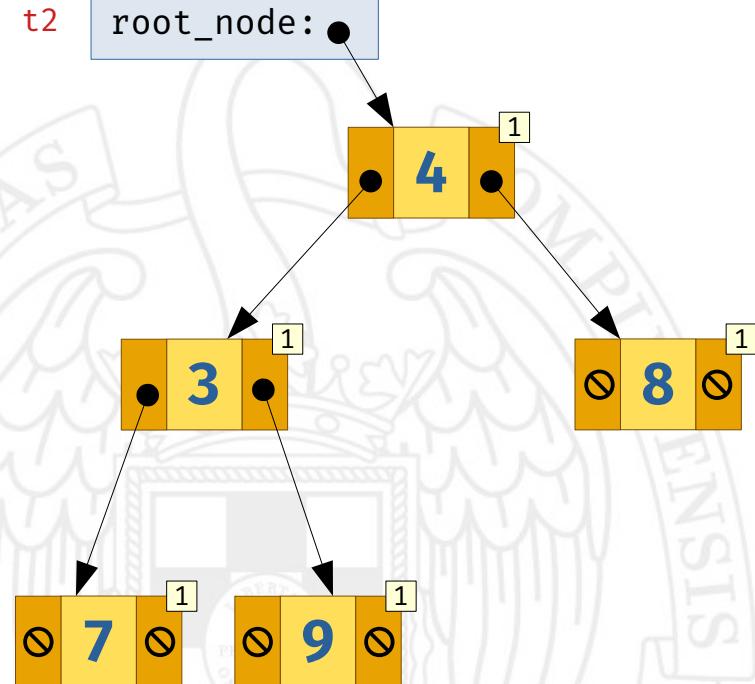


# Ejemplo

```
BinTree<int> comun = {{ 7 }, 3, { 9 }};
BinTree<int> t1 = {{}, 5, comun};
BinTree<int> t2 = {comun, 4, { 8 }};
```

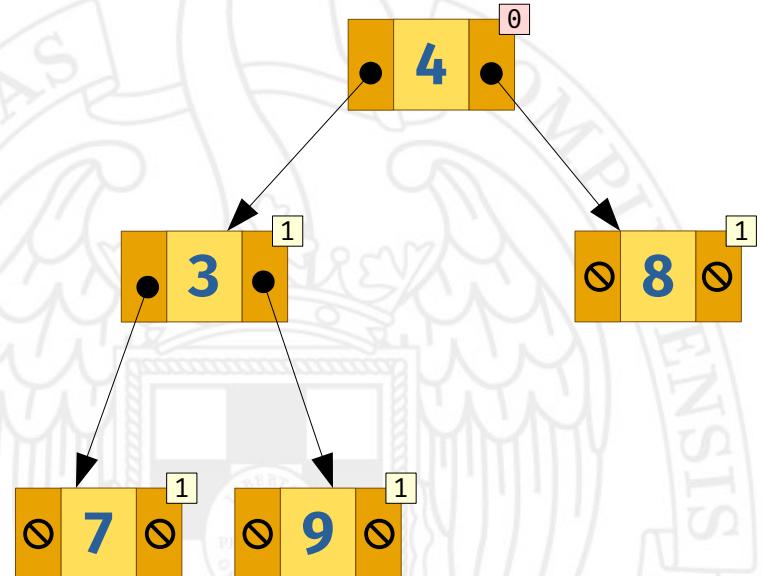
t2

root\_node:



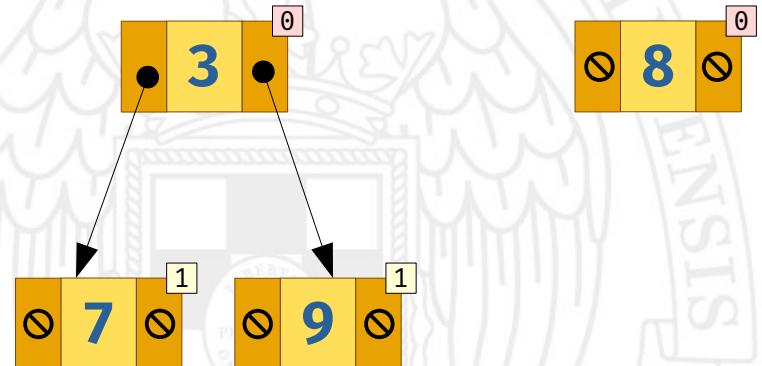
# Ejemplo

```
BinTree<int> comun = {{ 7 }, 3, { 9 }};
BinTree<int> t1 = {{}, 5, comun};
BinTree<int> t2 = {comun, 4, { 8 }};
```



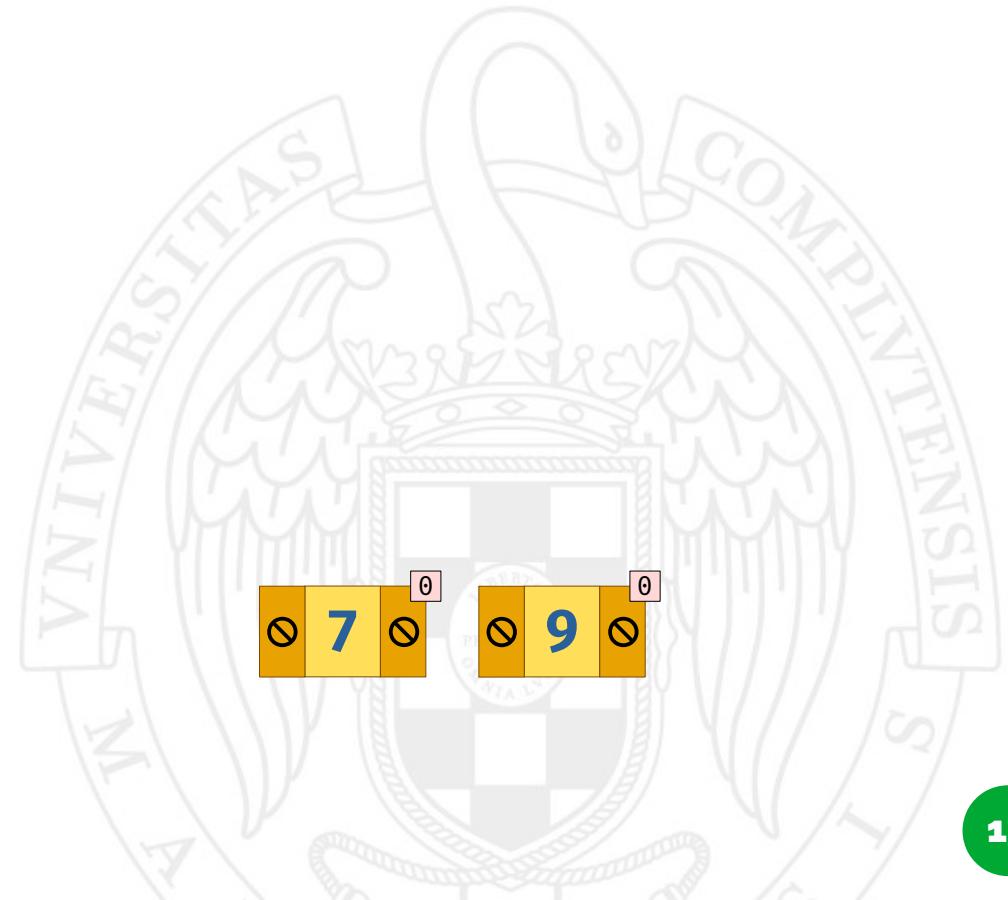
# Ejemplo

```
BinTree<int> comun = {{ 7 }, 3, { 9 }};
BinTree<int> t1 = {{}, 5, comun};
BinTree<int> t2 = {comun, 4, { 8 }};
```



# Ejemplo

```
BinTree<int> comun = {{ 7 }, 3, { 9 }};
BinTree<int> t1 = {{}, 5, comun};
BinTree<int> t2 = {comun, 4, { 8 }};
```



# Ejemplo

```
BinTree<int> comun = {{ 7 }, 3, { 9 }};
BinTree<int> t1 = {{}, 5, comun};
BinTree<int> t2 = {comun, 4, { 8 }};
```



ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

# Funciones sobre árboles binarios

Manuel Montenegro Montes

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

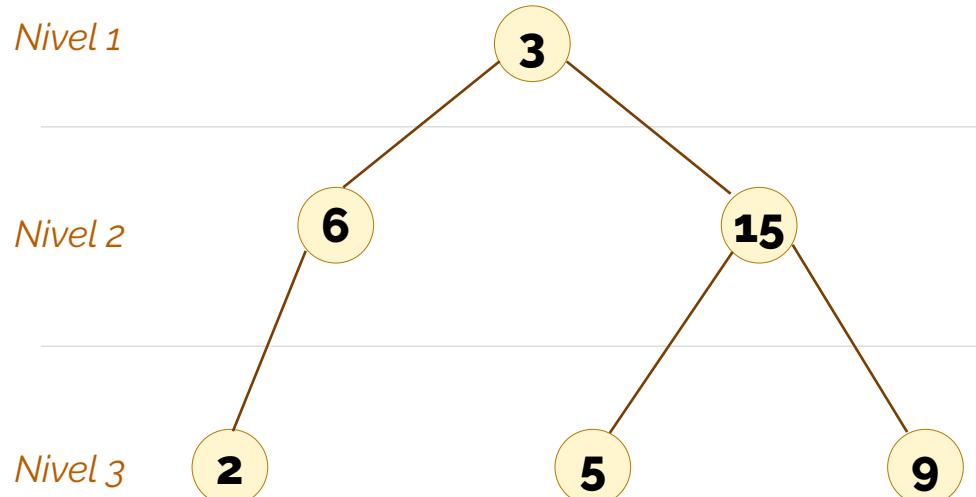
# Recordatorio: interfaz de BinTree<T>

```
template<class T>
class BinTree {
public:
    BinTree();
    BinTree(const T &elem);
    BinTree(const BinTree &left, const T &elem, const BinTree &right);

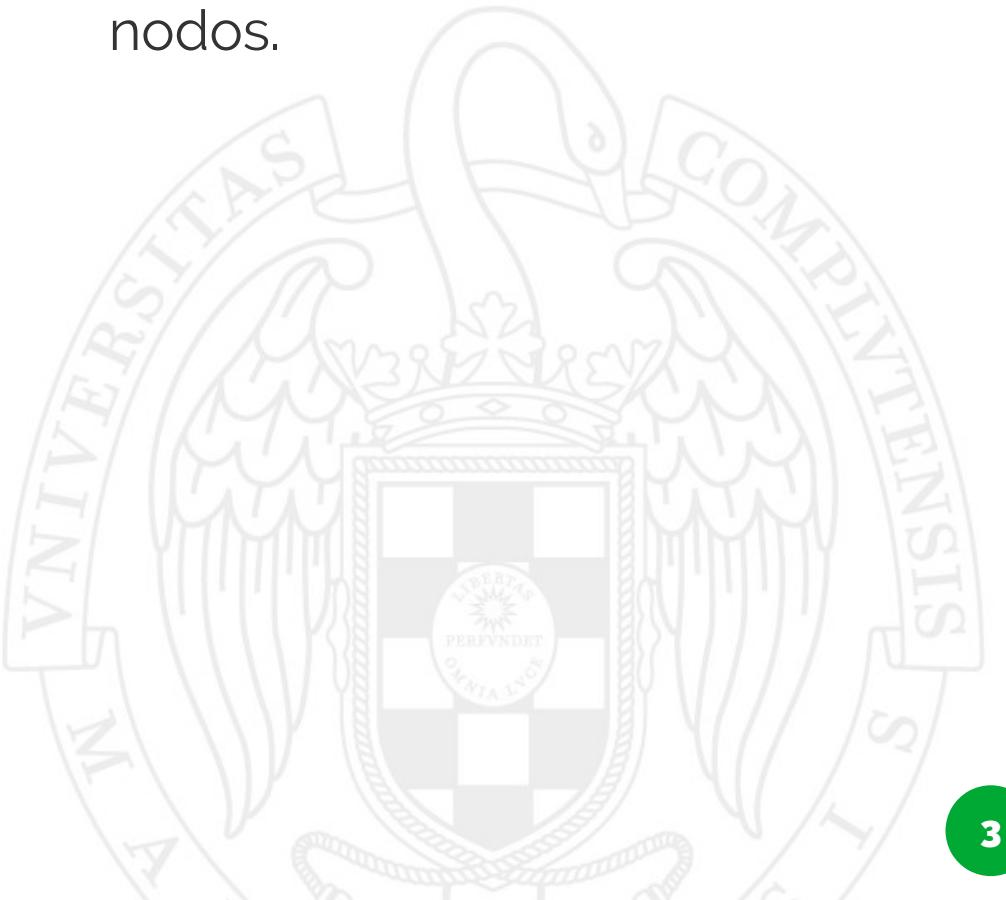
    const T & root() const;
    BinTree left() const;
    BinTree right() const;
    bool empty() const;

private:
    ...
};
```

# Recordatorio: altura de un árbol binario



- La **altura** de un árbol es el máximo de los niveles de los nodos.



# Definición recursiva de altura

- Es posible definir recursivamente la altura de un árbol binario:

$$\text{height}(\text{---}) = 0$$

$$\text{height}\left( \begin{array}{c} \text{x} \\ | \\ t_1 \quad t_2 \end{array} \right) = 1 + \max(\text{height}\left[ t_1 \right], \text{height}\left[ t_2 \right])$$

# Función height

$\text{height}(\text{---}) = 0$

$$\text{height}\left( \begin{array}{c} \text{x} \\ | \\ t_1 \quad t_2 \end{array} \right) = 1 + \max(\text{height}\left[ t_1 \right], \text{height}\left[ t_2 \right])$$

```
template<typename T>
int height(const BinTree<T> &tree) {
    if (tree.empty()) {
        return 0;
    } else {
        return 1 + std::max(height(tree.left()), height(tree.right()));
    }
}
```

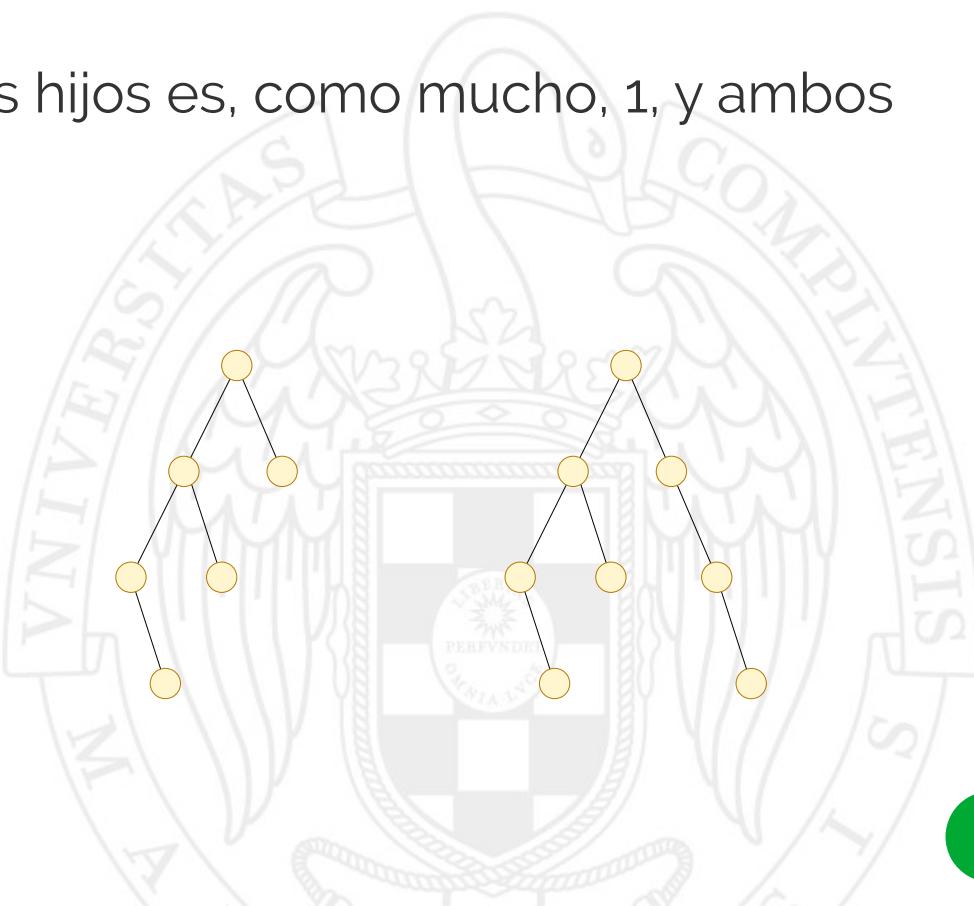
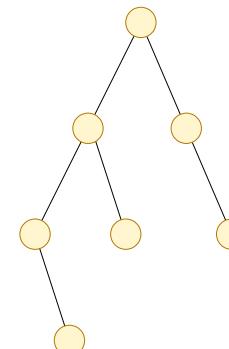
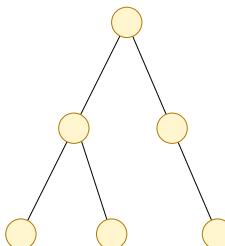
# Coste en tiempo

```
template<typename T>
int height(const BinTree<T> &tree) {
    if (tree.empty()) {
        return 0;
    } else {
        return 1 + std::max(height(tree.left()), height(tree.right()));
    }
}
```

# Árboles equilibrados en altura

Un árbol está **equilibrado en altura** si:

- Es el árbol vacío, o bien
- La diferencia entre las alturas de sus hijos es, como mucho, 1, y ambos están equilibrados en altura.



# Definición recursiva

***balanced***( $\text{---}$ ) = true

$$\mathbf{balanced} \left( \begin{array}{c} \text{x} \\ | \\ t_1 \quad t_2 \end{array} \right) = \mathbf{balanced} \left( t_1 \right) \wedge \mathbf{balanced} \left( t_2 \right)$$
$$\wedge \left| \mathbf{height} \left( t_1 \right) - \mathbf{height} \left( t_2 \right) \right| \leq 1$$

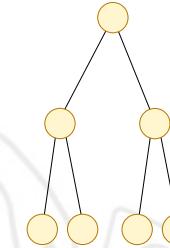
# Función balanced

```
template<typename T>
bool balanced(const BinTree<T> &tree) {
    if (tree.empty()) {
        return true;
    } else {
        bool bal_left = balanced(tree.left());
        bool bal_right = balanced(tree.right());
        int height_left = height(tree.left());
        int height_right = height(tree.right());
        return bal_left && bal_right && abs(height_left - height_right) ≤ 1;
    }
}
```

¿Cuál es el coste en tiempo?

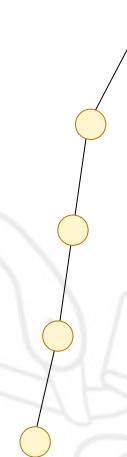
# Función balanced: caso mejor

```
template<typename T>
bool balanced(const BinTree<T> &tree) {
    if (tree.empty()) {
        return true;
    } else {
        bool bal_left = balanced(tree.left());
        bool bal_right = balanced(tree.right());
        int height_left = height(tree.left());
        int height_right = height(tree.right());
        return bal_left && bal_right && abs(height_left - height_right) ≤ 1;
    }
}
```

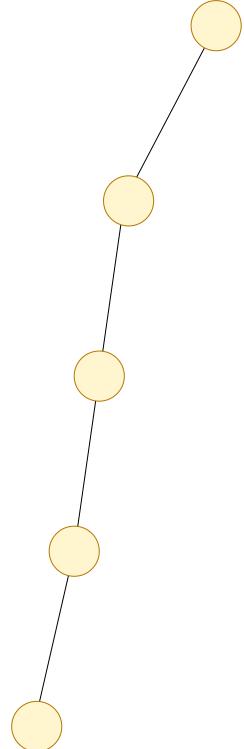


# Función balanced: caso peor

```
template<typename T>
bool balanced(const BinTree<T> &tree) {
    if (tree.empty()) {
        return true;
    } else {
        bool bal_left = balanced(tree.left());
        bool bal_right = balanced(tree.right());
        int height_left = height(tree.left());
        int height_right = height(tree.right());
        return bal_left && bal_right && abs(height_left - height_right) ≤ 1;
    }
}
```



# Problema de llamar a height



# ¿Cómo solucionarlo?

- Implementando una función auxiliar recursiva que **simultáneamente** calcule la altura y determine si un árbol está equilibrado.
- Esta función devuelve dos valores:
  - `balanced (bool)` – si el árbol está equilibrado o no.
  - `height (int)` – altura del árbol.
- La función `balanced_height` devolverá ambos valores como parámetros de salida.

# Función balanced\_height

```
template<typename T>
void balanced_height(const BinTree<T> &tree, bool &balanced, int &height) {
    if (tree.empty()) {
        balanced = true;
        height = 0;
    } else {
        bool bal_left, bal_right;
        int height_left, height_right;
        balanced_height(tree.left(), bal_left, height_left);
        balanced_height(tree.right(), bal_right, height_right);
        balanced = bal_left && bal_right && abs(height_left - height_right) ≤ 1;
        height = 1 + std::max(height_left, height_right);
    }
}
```

# Función balanced

```
template<typename T>
bool balanced(const BinTree<T> &tree) {
    bool balanced;
    int height;
    balanced_height(tree, balanced, height);
    return balanced;
}
```



# Moraleja

- La mayoría de las funciones que operan sobre árboles son recursivas.
- En muchos casos estas funciones deben devolver valores auxiliares adicionales para evitar costes en tiempo elevados.



ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

# Recorridos de árboles binarios

Manuel Montenegro Montes

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

# ¿Qué es un recorrido?

- **Recorrer** un árbol significa visitar los nodos de un árbol, de modo que cada nodo es visitado exactamente una vez.
- **Visitar** un nodo significa realizar una acción específica, que puede depender del valor contenido dentro de ese nodo.
  - Imprimir por pantalla el valor del nodo.
  - Sumar el valor del nodo a una variable externa.
  - Escribir el valor del nodo en un fichero.
  - Incrementar un contador externo.

Comenzaremos aquí

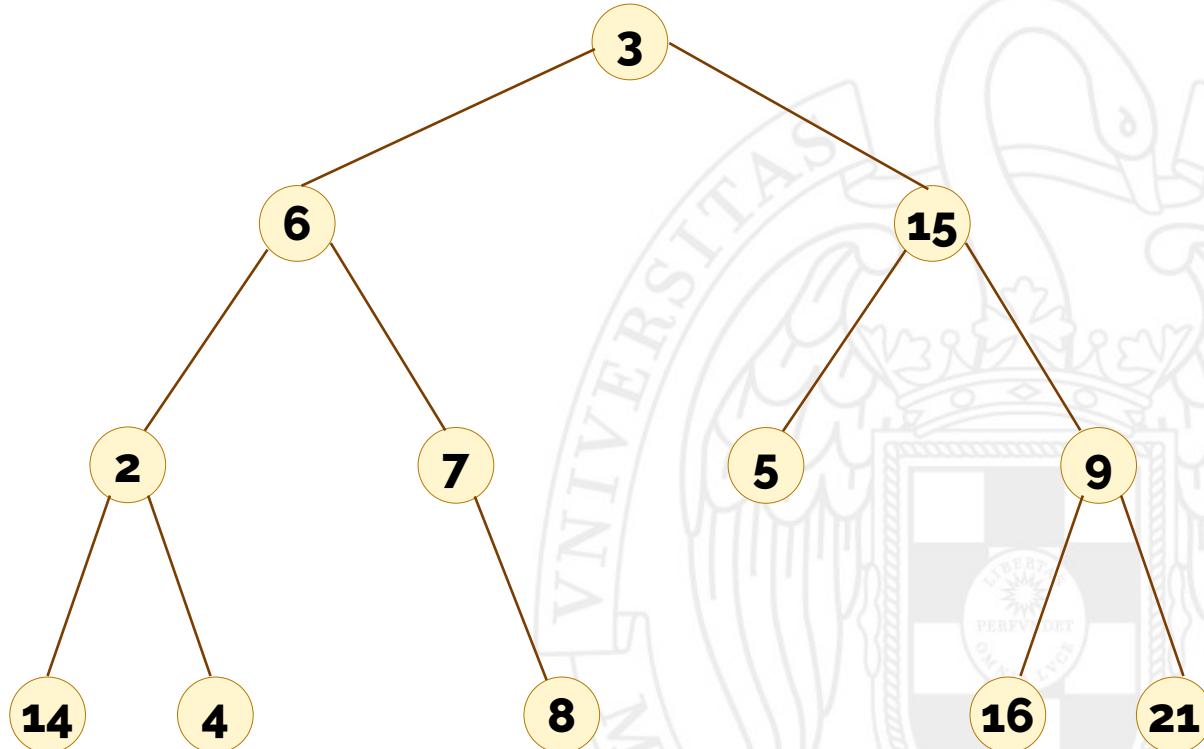
# Tipos de recorridos

- Recorrido en profundidad  
*Depth First Search (DFS)*
  - Recorrido en anchura  
*Breadth First Search (BFS)*
- 

- Preorden
- Inorden
- Postorden

# Recorridos en profundidad

- Se explora completamente un hijo antes de pasar al siguiente.

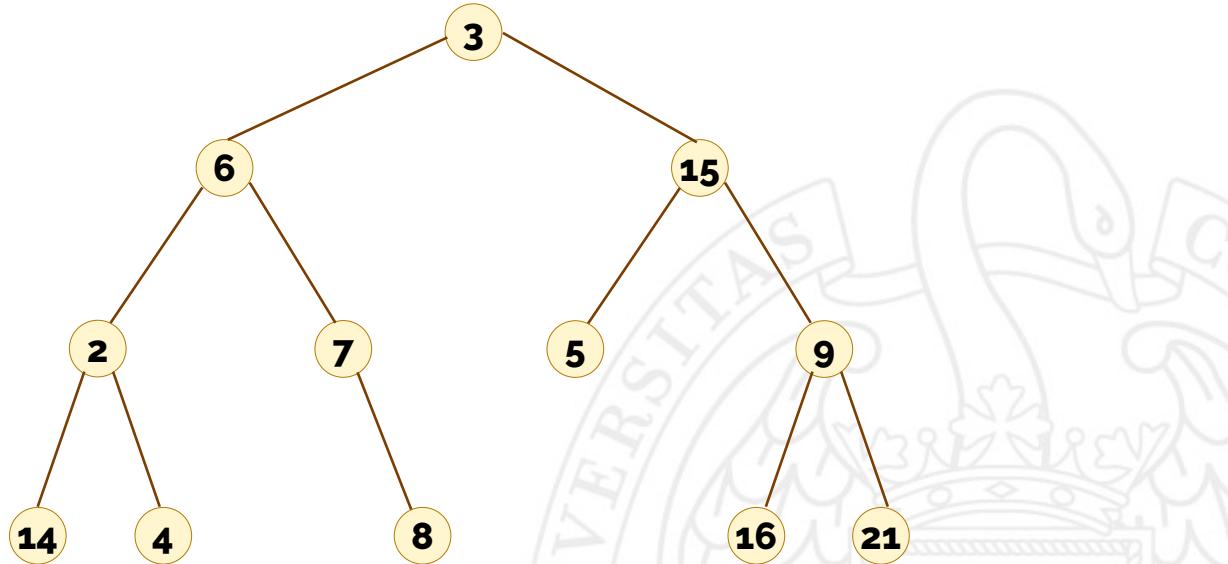


# Recorridos en profundidad

- Se explora completamente un hijo antes de pasar al siguiente.
  - **Preorden:** Visitar raíz, luego recorrer hijo izquierdo, luego recorrer hijo derecho.



# Recorrido en preorden

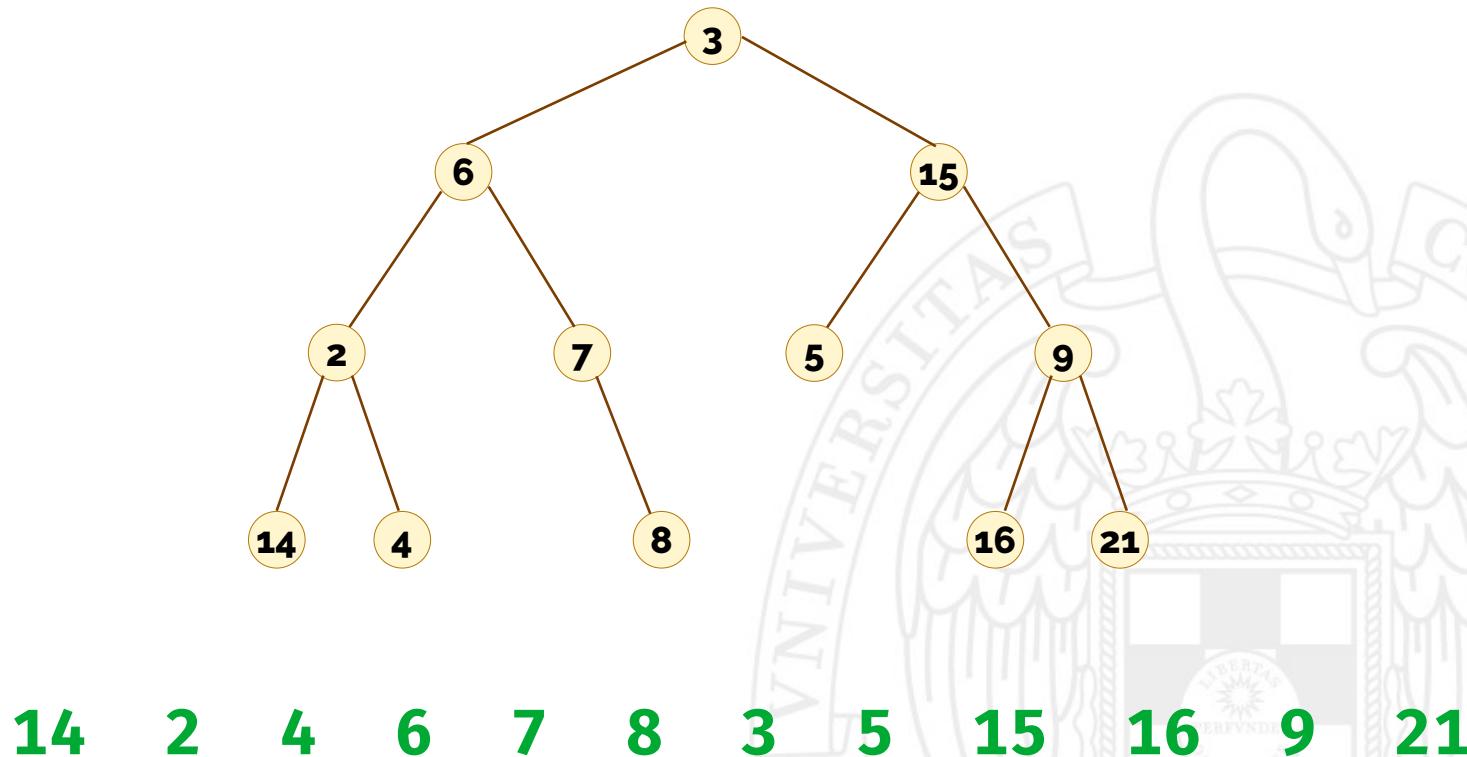


3 6 2 14 4 7 8 15 5 9 16 21

# Recorridos en profundidad

- Se explora completamente un hijo antes de pasar al siguiente.
  - **Preorden:** Visitar raíz, luego recorrer hijo izquierdo, luego recorrer hijo derecho.
  - **Inorden:** Recorrer hijo izquierdo, visitar raíz, luego recorrer hijo derecho.

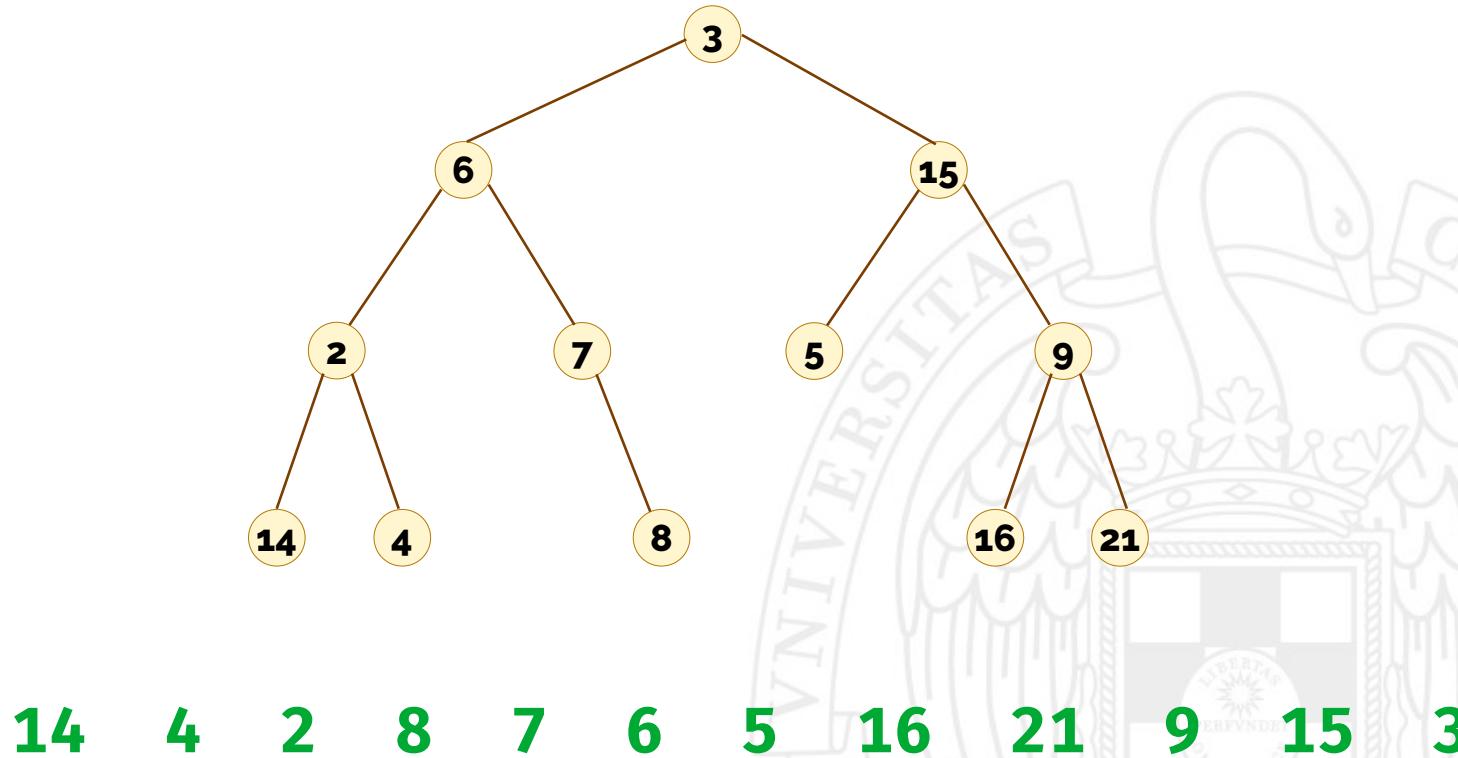
# Recorrido en inorden



# Recorridos en profundidad

- Se explora completamente un hijo antes de pasar al siguiente.
  - **Preorden:** Visitar raíz, luego recorrer hijo izquierdo, luego recorrer hijo derecho.
  - **Inorden:** Recorrer hijo izquierdo, visitar raíz, luego recorrer hijo derecho.
  - **Postorden:** Recorrer hijo izquierdo, luego recorrer hijo derecho, luego visitar raíz.

# Recorrido en postorden



# Recorridos en profundidad

- Se explora completamente un hijo antes de pasar al siguiente.
  - **Preorden:** Visitar raíz, luego recorrer hijo izquierdo, luego recorrer hijo derecho.
  - **Inorden:** Recorrer hijo izquierdo, visitar raíz, luego recorrer hijo derecho.
  - **Postorden:** Recorrer hijo izquierdo, luego recorrer hijo derecho, luego visitar raíz.

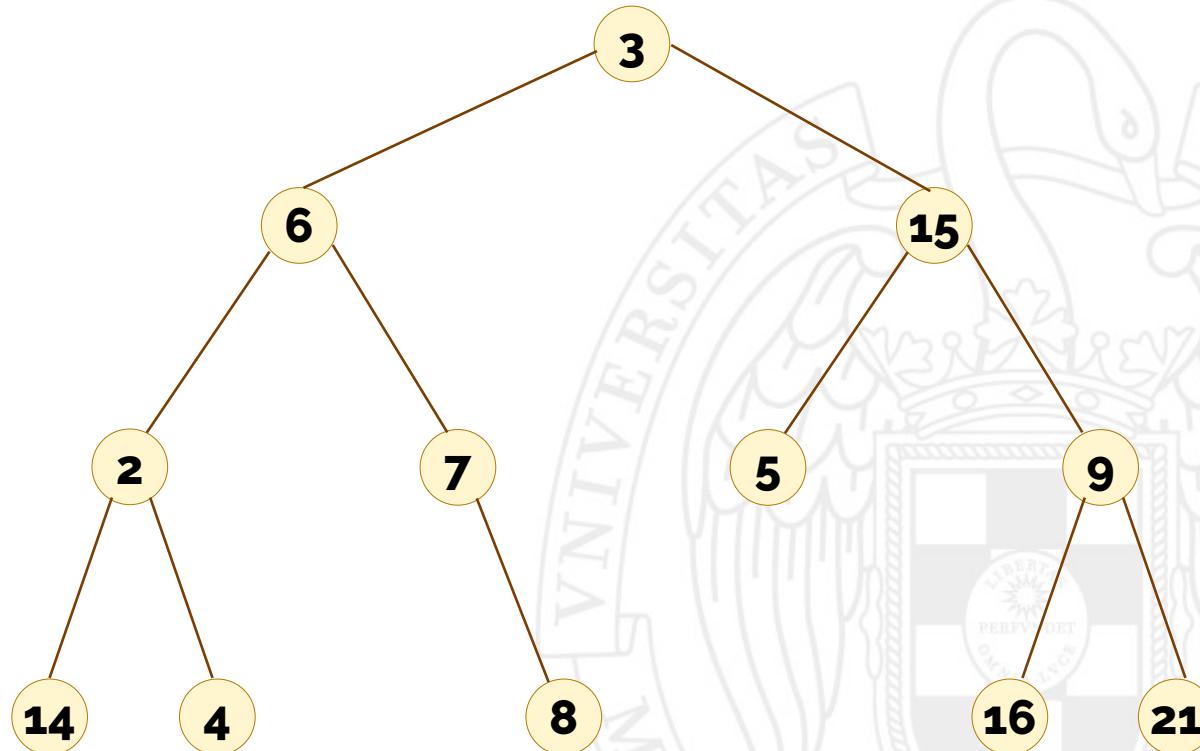
# Tipos de recorridos

- Recorrido en profundidad  
*Depth First Search (DFS)*
  - Recorrido en anchura  
*Breadth First Search (BFS)*
- 

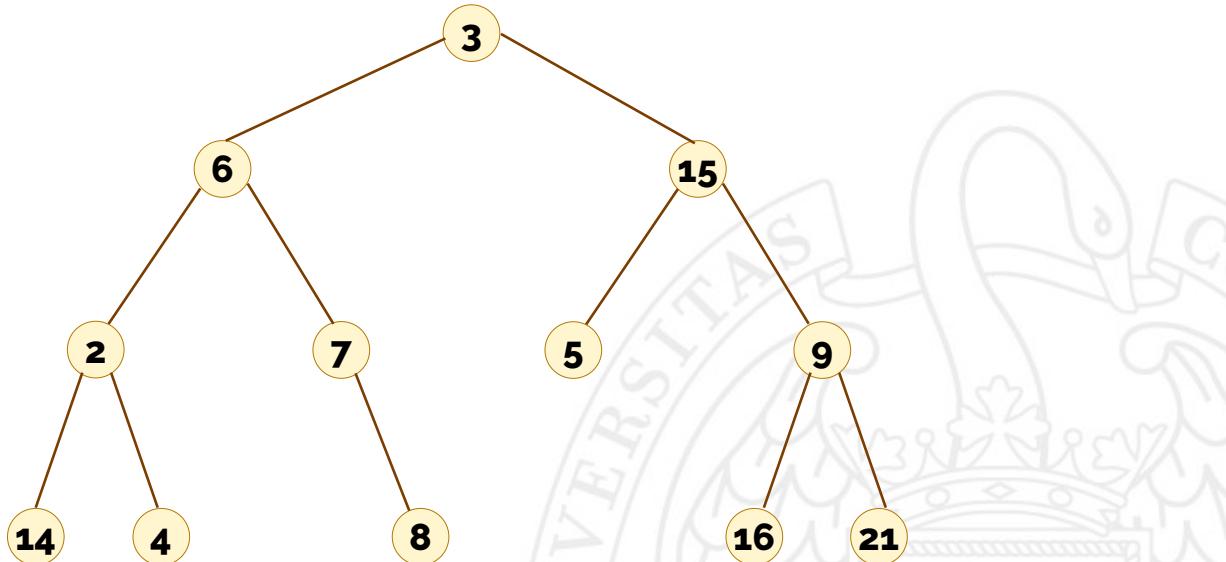
- Preorden
- Inorden
- Postorden

# Recorridos en anchura (*por niveles*)

- Se explora completamente un nivel antes de pasar al siguiente.



# Recorrido en anchura (*por niveles*)



3    6    15    2    7    5    9    14    4    8    16    21

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

# Implementando recorridos en profundidad (*DFS*)

Manuel Montenegro Montes

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

# Tipos de recorridos

- Recorrido en profundidad  
*Depth First Search (DFS)*
  - Recorrido en anchura  
*Breadth First Search (BFS)*
- 

- Preorden
- Inorden
- Postorden

# Recordatorio: interfaz de BinTree<T>

```
template<class T>
class BinTree {
public:
    BinTree();
    BinTree(const T &elem);
    BinTree(const BinTree &left, const T &elem, const BinTree &right);

    const T & root() const;
    BinTree left() const;
    BinTree right() const;
    bool empty() const;

private:
    ...
};
```

# Recordatorio: interfaz de BinTree<T>

```
template<class T>
class BinTree {
public:
    // ...
    void preorder() const;
    void inorder() const;
    void postorder() const;

private:
    ...
};
```

- Añadimos tres nuevos métodos a BinTree<T>.



# Recordatorio: interfaz de BinTree<T>

```
template<class T>
class BinTree {
public:
    // ...
    void preorder() const {
        preorder(root_node);
    }

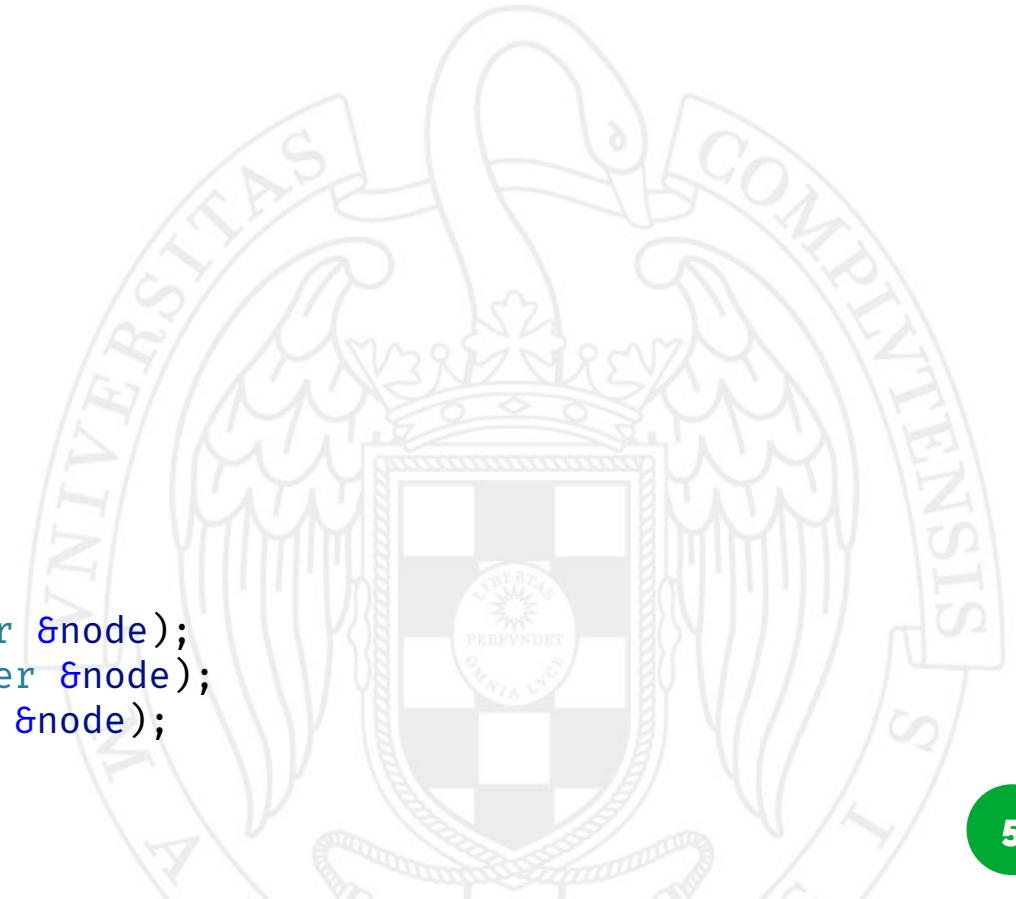
    void inorder() const {
        inorder(root_node);
    }

    void postorder() const {
        postorder(root_node);
    }

private:
    static void preorder(const NodePointer &node);
    static void postorder(const NodePointer &node);
    static void inorder(const NodePointer &node);

    ...
};
```

- Estos métodos harán uso de otros tres métodos privados auxiliares.



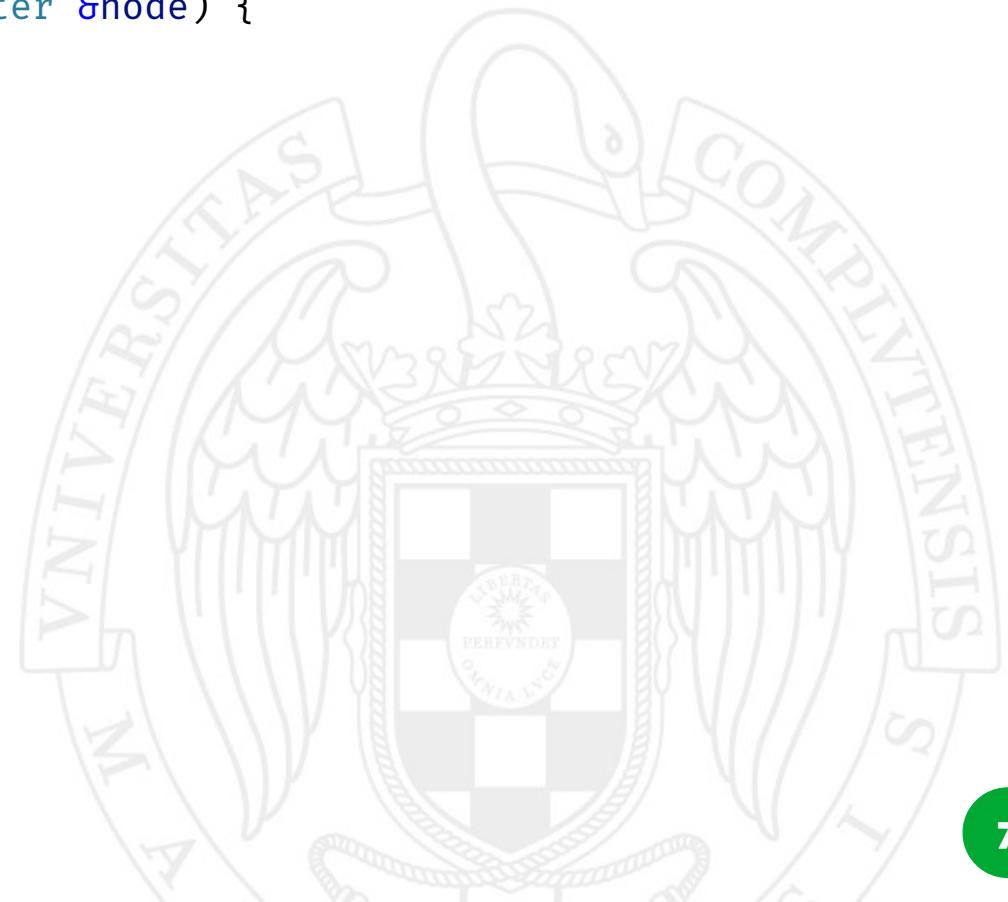
# Método auxiliar preorder

```
template<typename T>
void BinTree<T>::preorder(const NodePointer &node) {
    if (node != nullptr) {
        std::cout << node->elem << " ";
        preorder(node->left);
        preorder(node->right);
    }
}
```



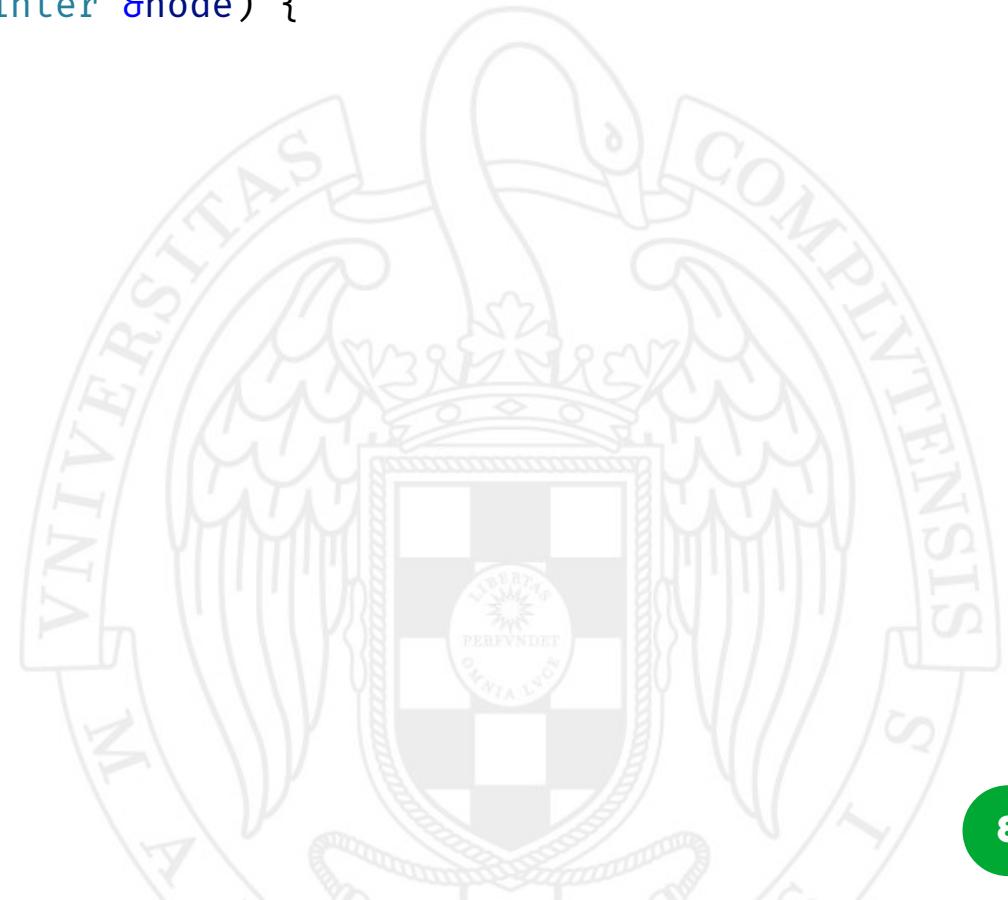
# Método auxiliar inorder

```
template<typename T>
void BinTree<T>::inorder(const NodePointer &node) {
    if (node != nullptr) {
        inorder(node->left);
        std::cout << node->elem << " ";
        inorder(node->right);
    }
}
```



# Método auxiliar postorder

```
template<typename T>
void BinTree<T>::postorder(const NodePointer &node) {
    if (node != nullptr) {
        postorder(node->left);
        postorder(node->right);
        std::cout << node->elem << " ";
    }
}
```



# Ejemplo

```
int main() {
    BinTree<int> tree = {{{ 9 }, 4, { 5 }}, 7, {{ 10 }, 4, { 6 }}};

    std::cout << "Recorrido en preorden: " << std::endl;
    tree.preorder();
    std::cout << std::endl;

    std::cout << "Recorrido en inorder: " << std::endl;
    tree.inorder();
    std::cout << std::endl;

    std::cout << "Recorrido en postorden: " << std::endl;
    tree.postorder();
    std::cout << std::endl;

    return 0;
}
```

Recorrido en preorden:  
7 4 9 5 4 10 6  
Recorrido en inorder:  
9 4 5 7 10 4 6  
Recorrido en postorden:  
9 5 4 10 6 4 7

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

# Implementando recorridos en anchura (*BFS*)

Manuel Montenegro Montes

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

# Tipos de recorridos

- Recorrido en profundidad  
*Depth First Search (DFS)*
  - Recorrido en anchura  
*Breadth First Search (BFS)*
- 

- Preorden
- Inorden
- Postorden

# Nuevo método

```
template<class T>
class BinTree {
public:
    // ...
    void preorder() const;
    void inorder() const;
    void postorder() const;

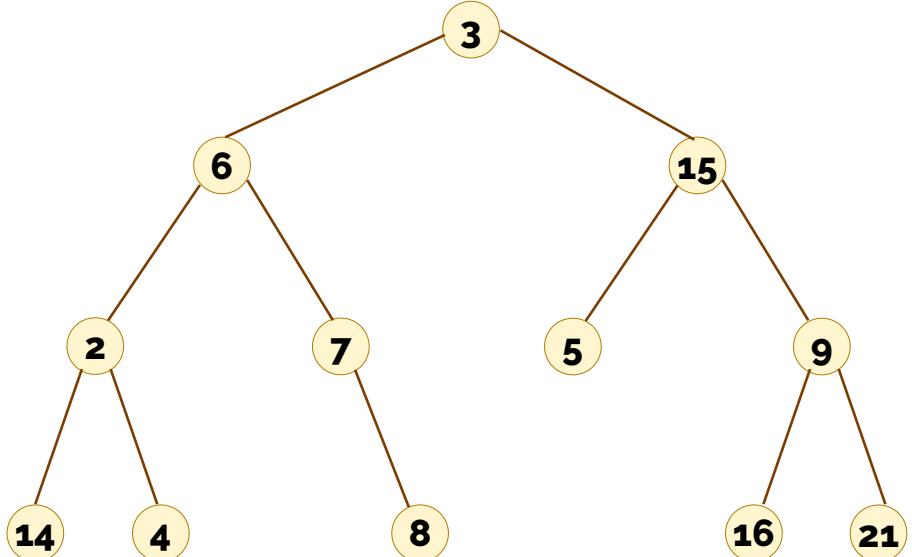
    void levelorder() const;

private:
    ...
};
```

- Implementamos el recorrido en profundidad mediante un nuevo método.

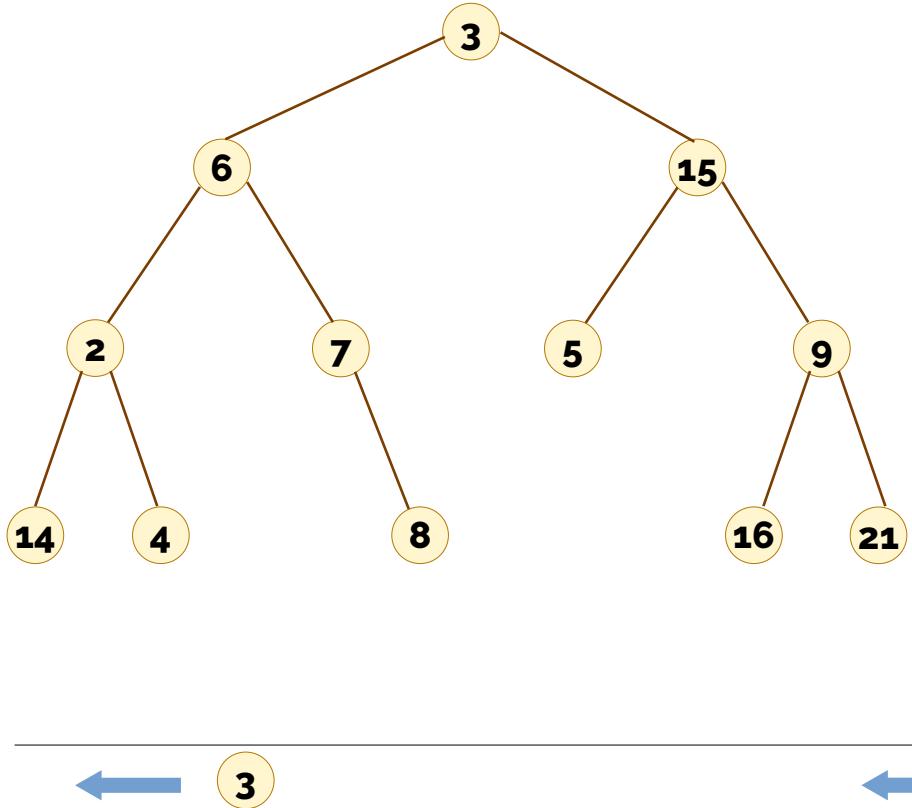


# Algoritmo de recorrido en anchura



- Utilizaremos una **cola** que contiene punteros a nodos.
- Esta cola representa los nodos pendientes que nos quedan por visitar.

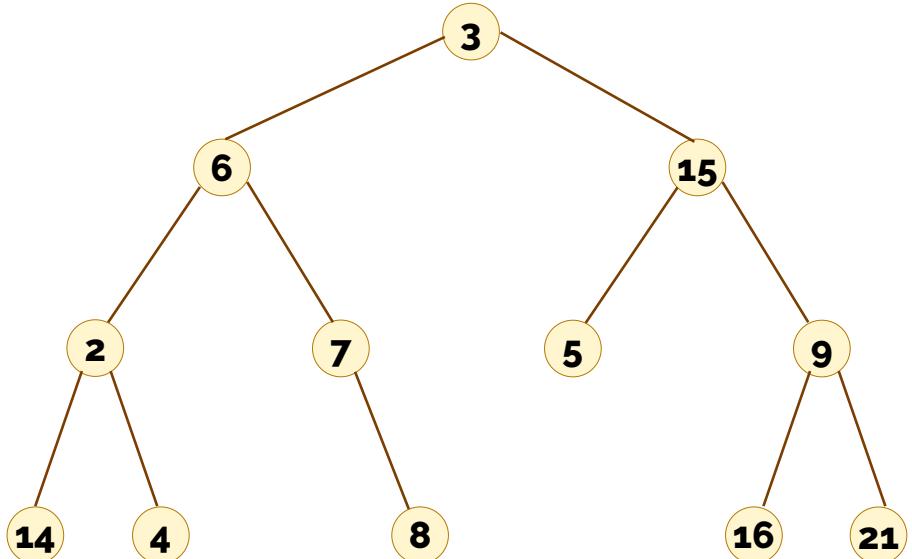
# Algoritmo de recorrido en anchura



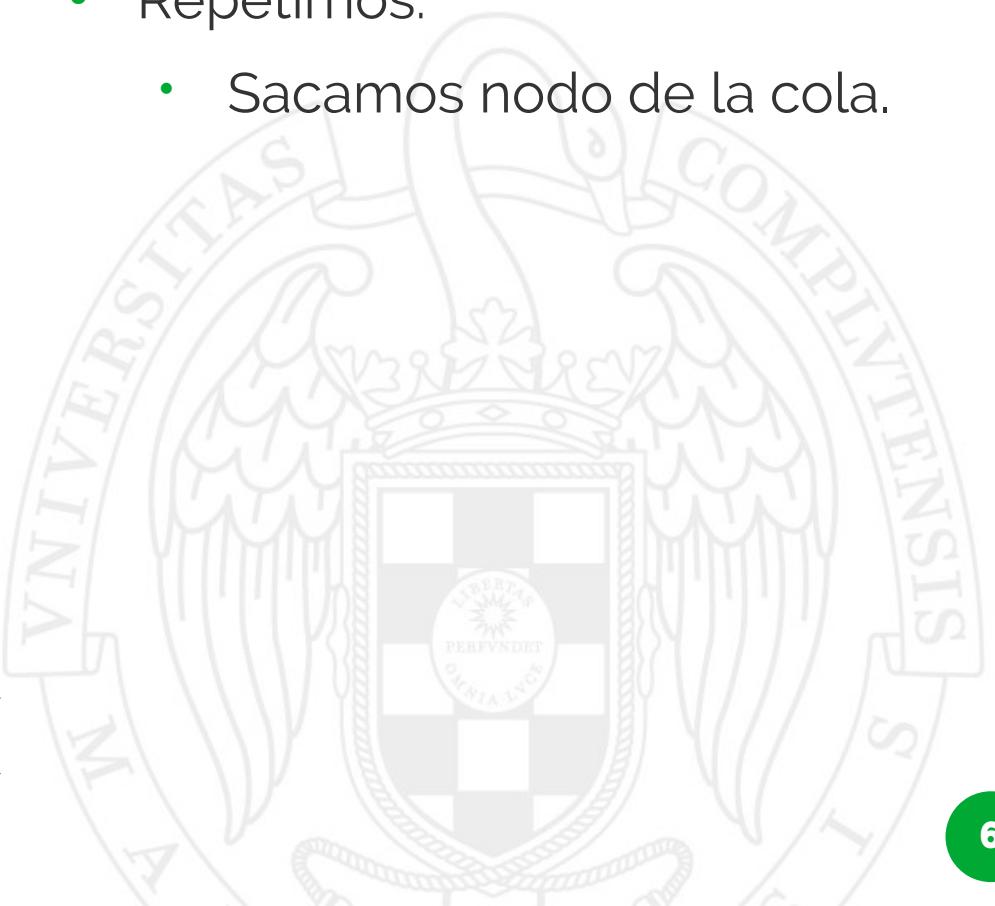
- Insertamos la raíz en la cola.
- Repetimos:



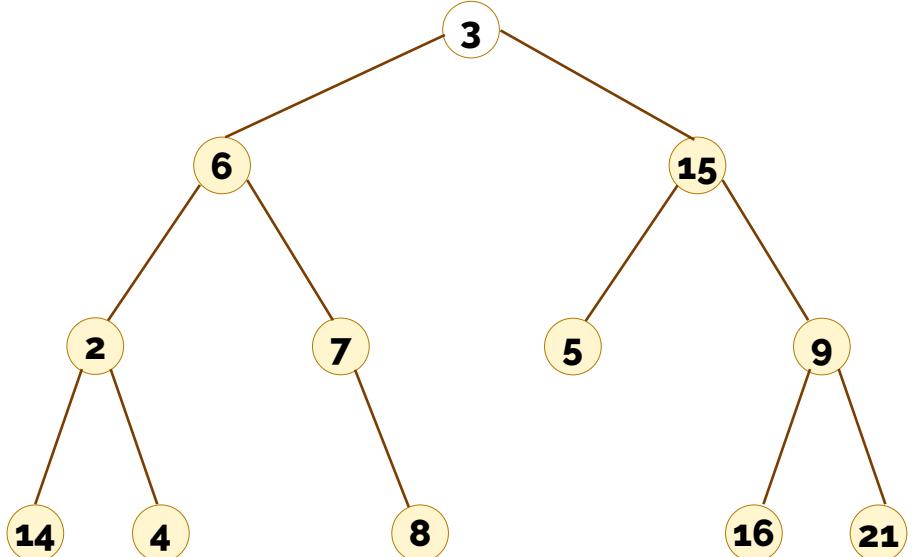
# Algoritmo de recorrido en anchura



- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.

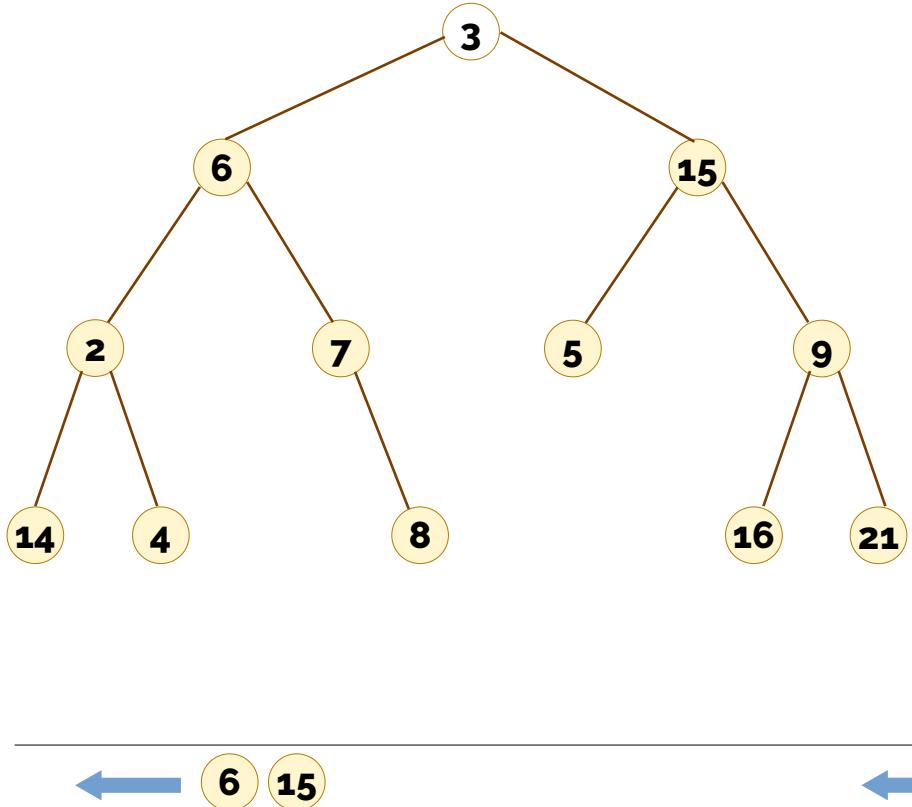


# Algoritmo de recorrido en anchura



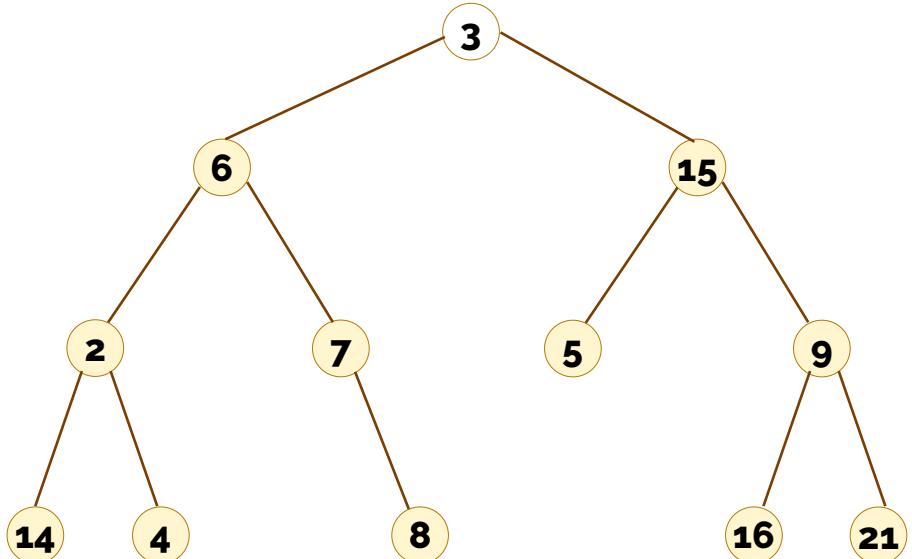
- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.

# Algoritmo de recorrido en anchura



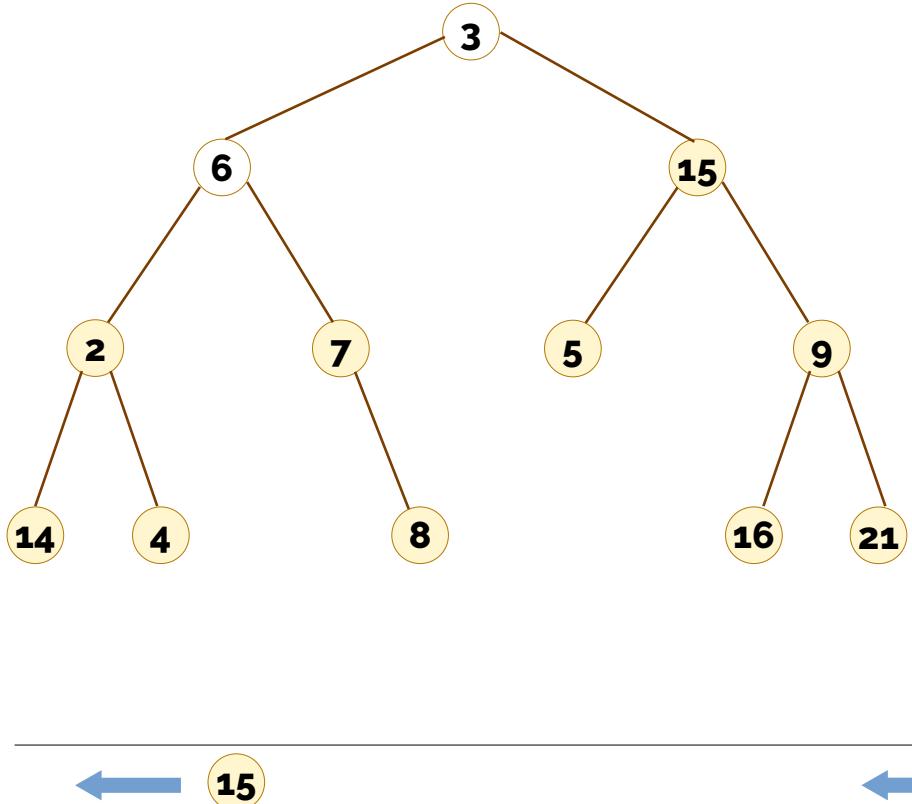
- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

# Algoritmo de recorrido en anchura



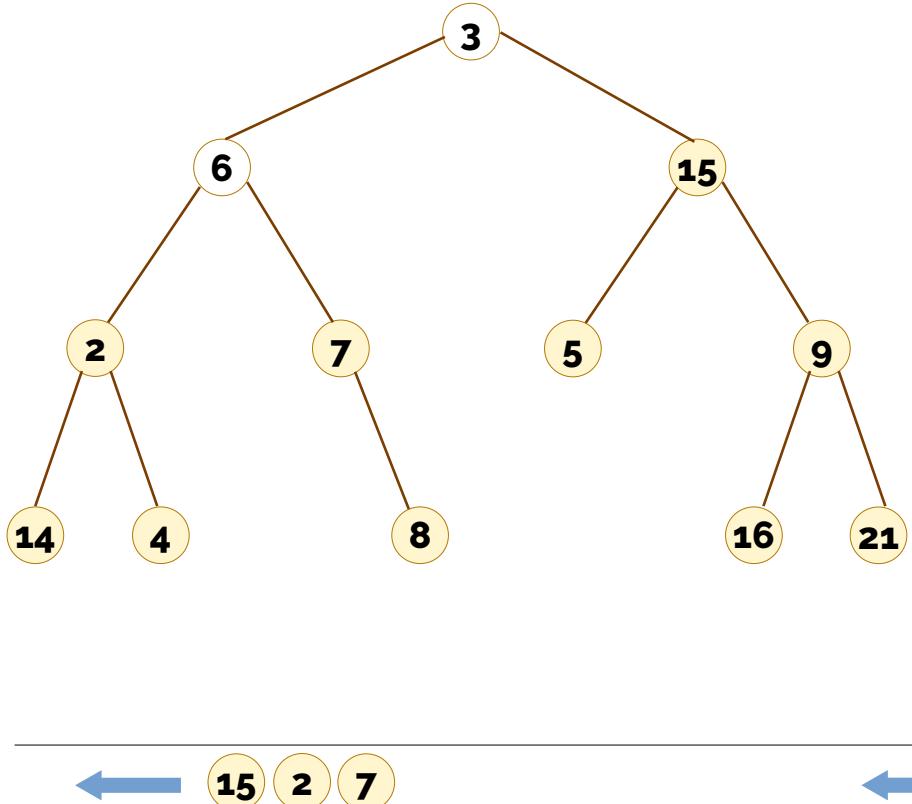
- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

# Algoritmo de recorrido en anchura



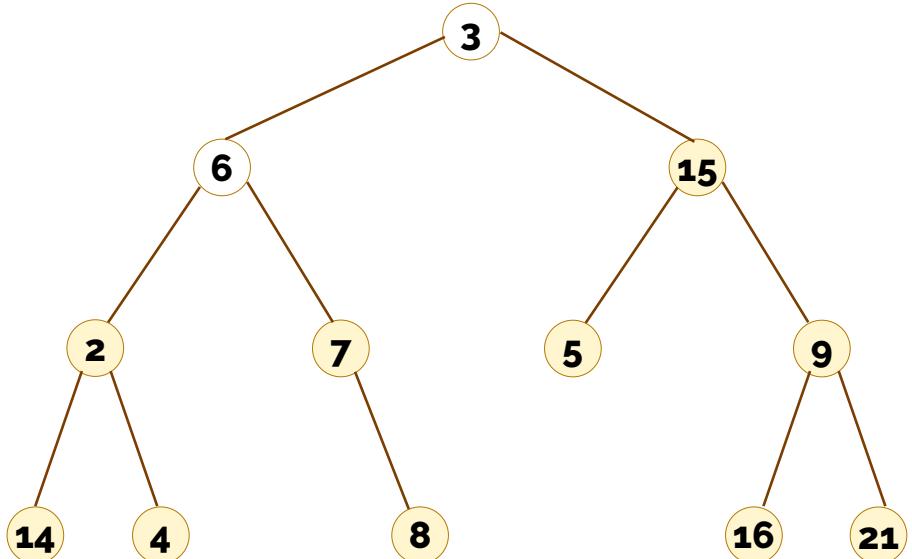
- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

# Algoritmo de recorrido en anchura



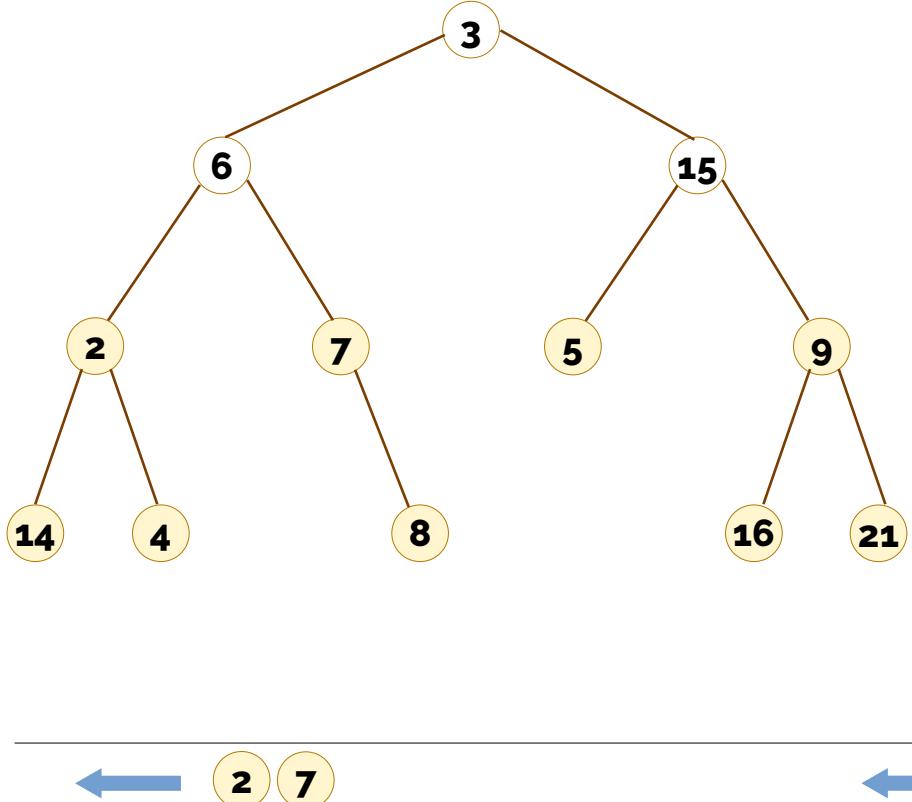
- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

# Algoritmo de recorrido en anchura



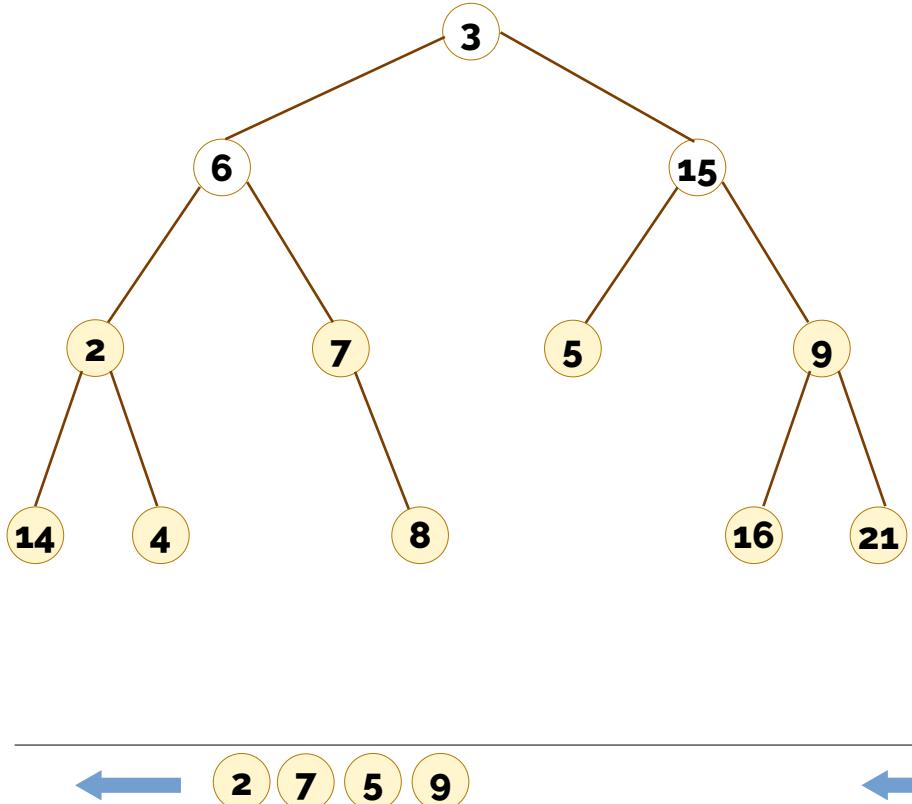
- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

# Algoritmo de recorrido en anchura



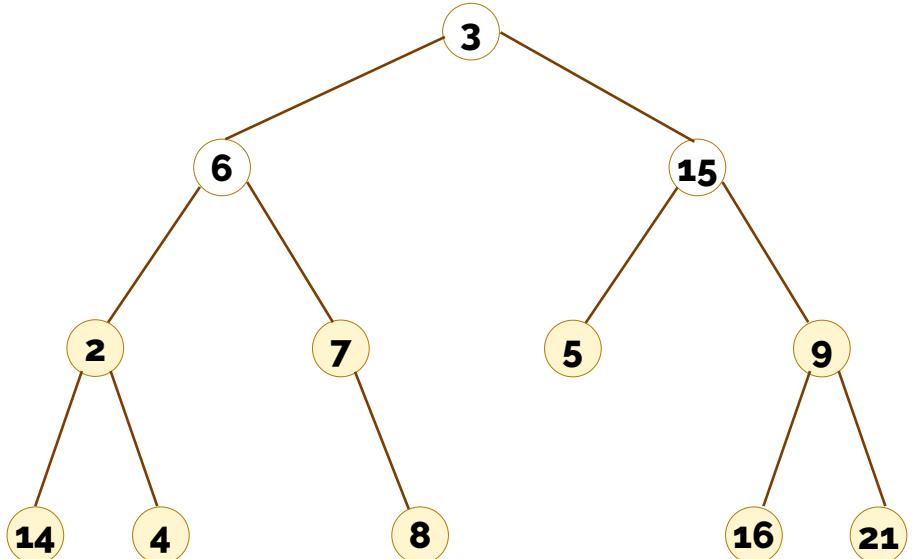
- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

# Algoritmo de recorrido en anchura



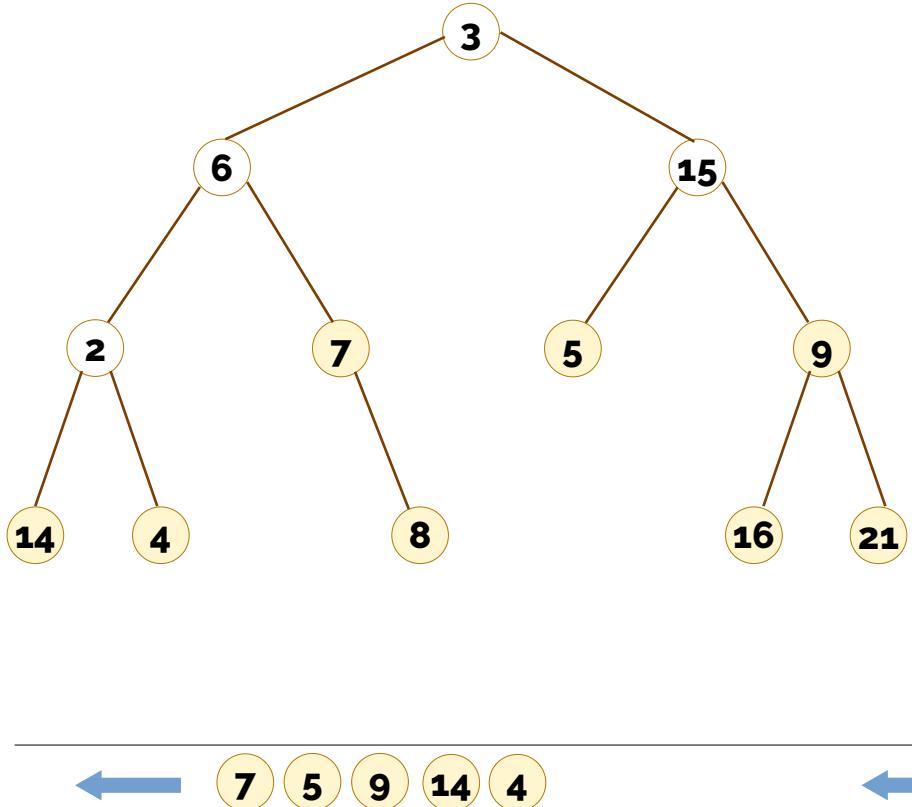
- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

# Algoritmo de recorrido en anchura



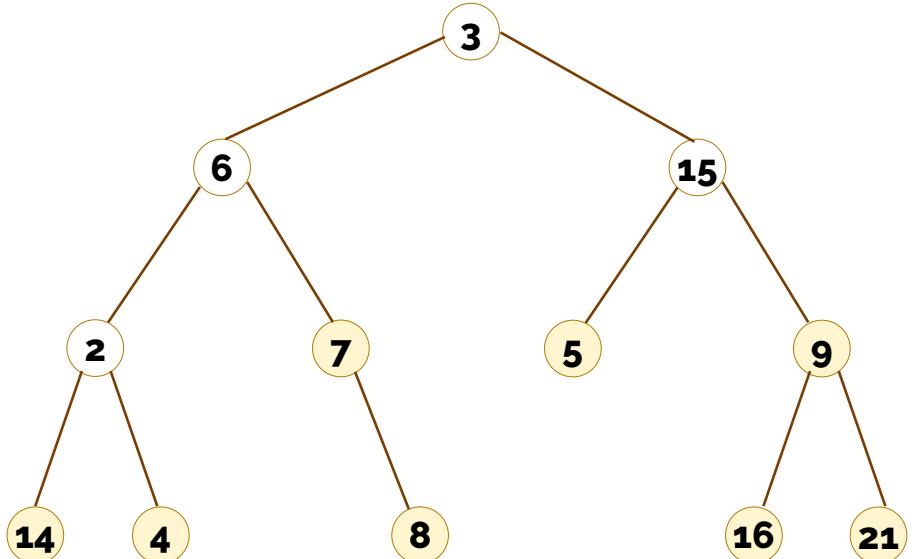
- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

# Algoritmo de recorrido en anchura



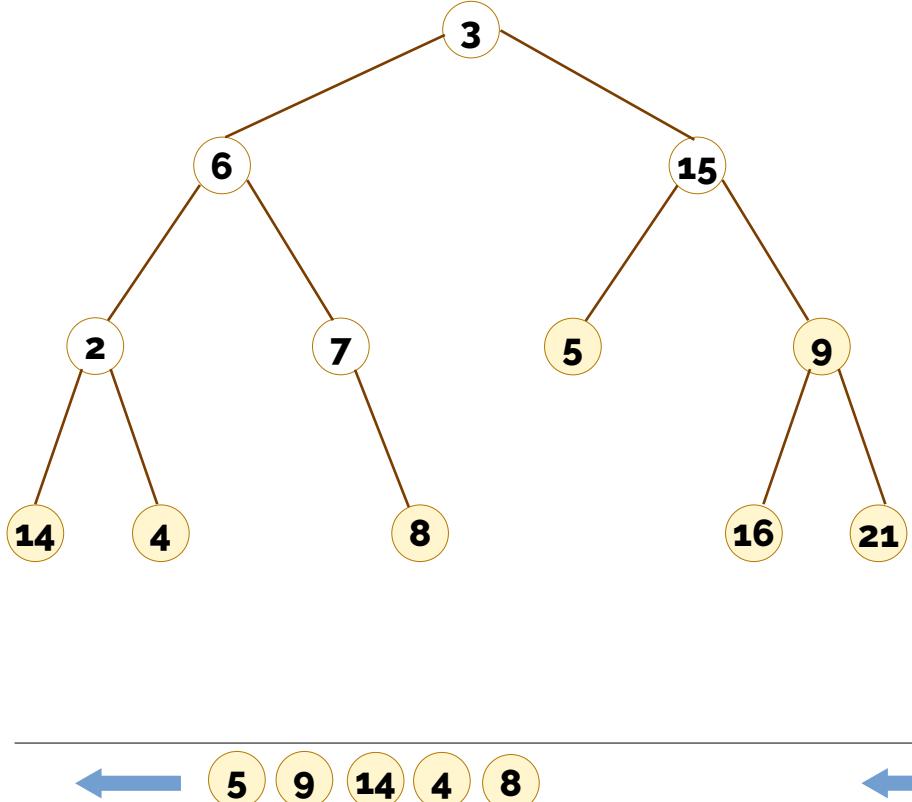
- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

# Algoritmo de recorrido en anchura



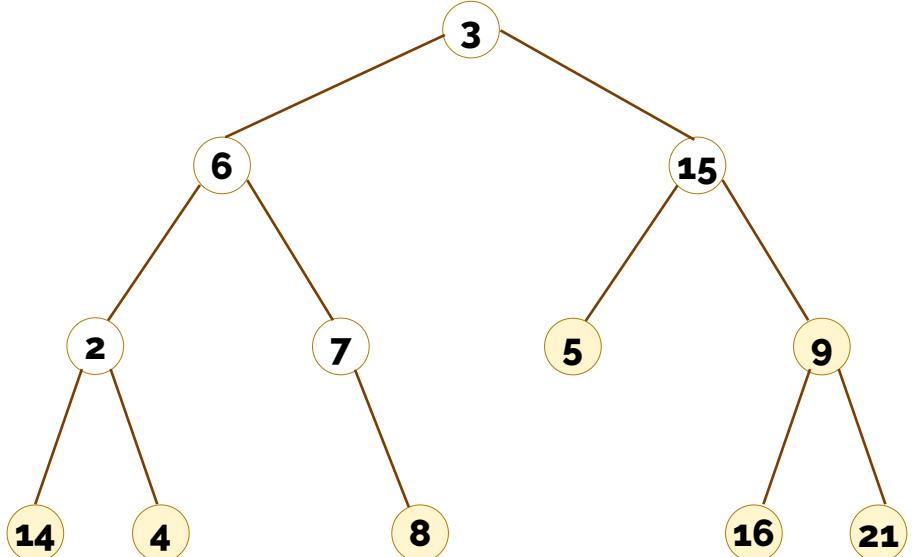
- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

# Algoritmo de recorrido en anchura



- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

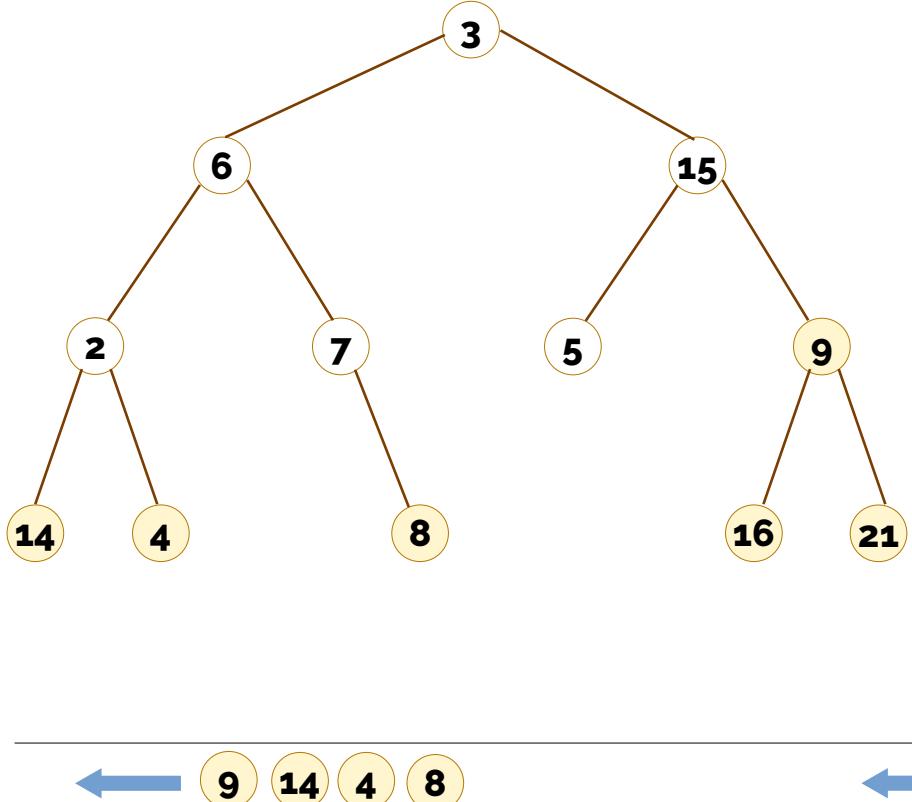
# Algoritmo de recorrido en anchura



5

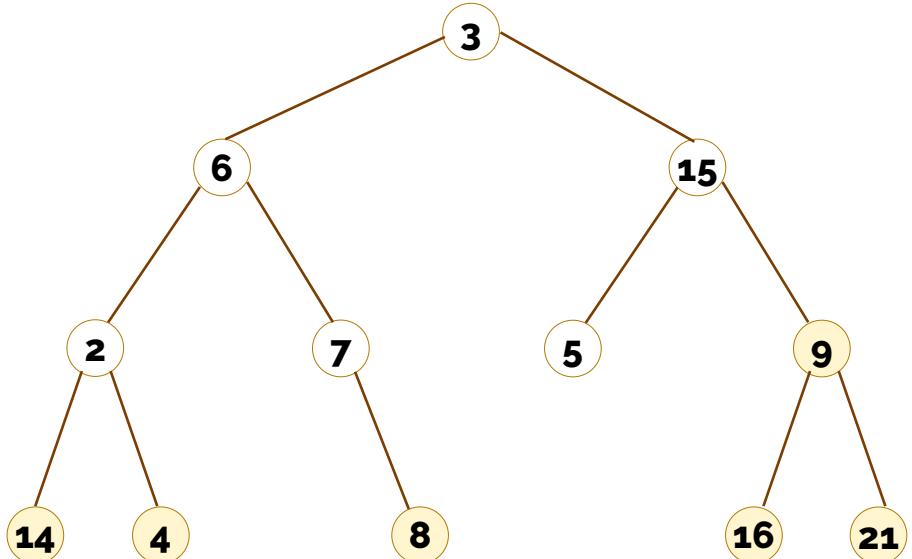
- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

# Algoritmo de recorrido en anchura



- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

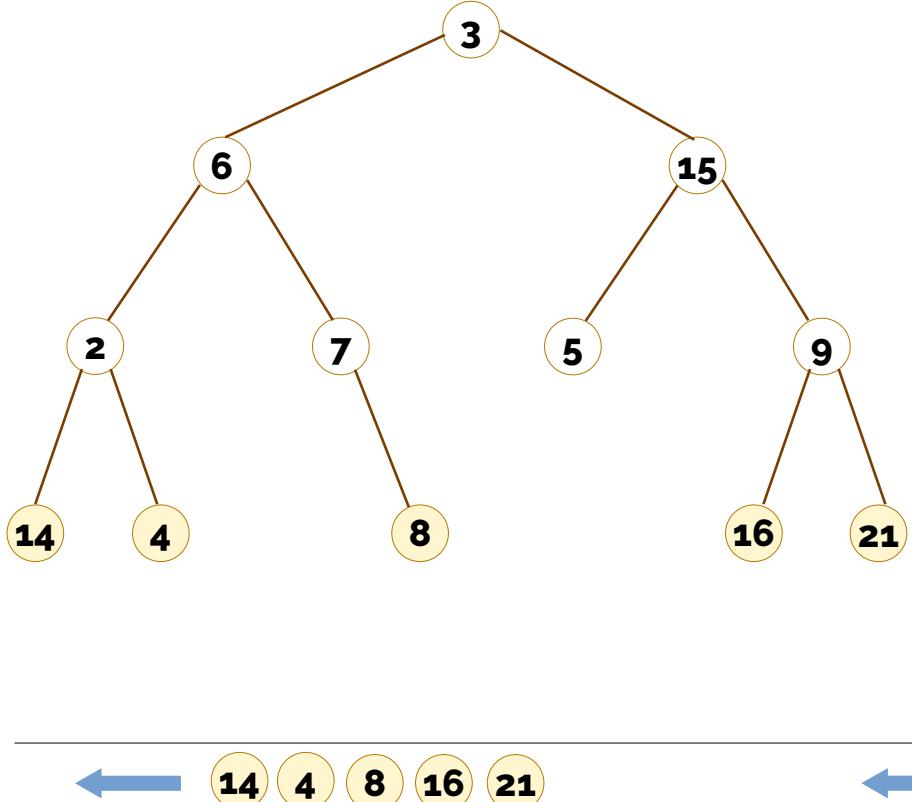
# Algoritmo de recorrido en anchura



9

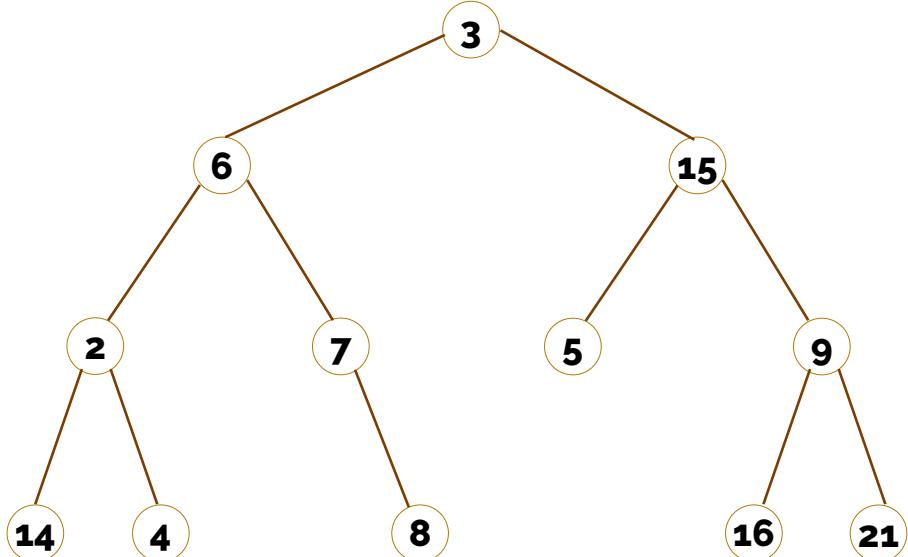
- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

# Algoritmo de recorrido en anchura



- Insertamos la raíz en la cola.
- Repetimos:
  - Sacamos nodo de la cola.
  - Visitamos ese nodo.
  - Insertamos sus hijos en la cola.

# Algoritmo de recorrido en anchura



- Insertamos la raíz en la cola.
  - Repetimos:
    - Sacamos nodo de la cola.
    - Visitamos ese nodo.
    - Insertamos sus hijos en la cola.
- hasta que la cola esté vacía.

# Código de levelorder

```
template<typename T>
void BinTree<T>::levelorder() const {
    std::queue<NodePointer> pending;
    pending.push(root_node);

    while (!pending.empty()) {
        NodePointer current = pending.front();
        pending.pop();
        std::cout << current->elem << " ";
        if (current->left != nullptr) {
            pending.push(current->left);
        }
        if (current->right != nullptr) {
            pending.push(current->right);
        }
    }
}
```

- Insertamos la raíz en la cola.
  - Repetimos:
    - Sacamos nodo de la cola.
    - Visitamos ese nodo.
    - Insertamos sus hijos en la cola.
- hasta que la cola esté vacía.

# Código de levelorder

```
template<typename T>
void BinTree<T>::levelorder() const {
    std::queue<NodePointer> pending;
    if (root_node != nullptr) {
        pending.push(root_node);
    }
    while (!pending.empty()) {
        NodePointer current = pending.front();
        pending.pop();
        std::cout << current->elem << " ";
        if (current->left != nullptr) {
            pending.push(current->left);
        }
        if (current->right != nullptr) {
            pending.push(current->right);
        }
    }
}
```

- Añadimos una guarda en el caso en el que el árbol esté vacío.



# Ejemplo

```
int main() {
    BinTree<int> tree {{{ 9 }, 4, { 5 }}, 7, {{ }, 4, { 6 }}};

    std::cout << "Recorrido por niveles: " << std::endl;
    tree.levelorder();
    std::cout << std::endl;

    return 0;
}
```

Recorrido por niveles:  
7 4 4 9 5 6

# Método auxiliar postorder

```
template<typename T>
void BinTree<T>::postorder(const NodePointer &node) {
    if (node != nullptr) {
        postorder(node->left);
        postorder(node->right);
        std::cout << node->elem << " ";
    }
}
```



# Ejemplo

```
int main() {
    BinTree<int> tree = {{{ 9 }, 4, { 5 }}, 7, {{ 10 }, 4, { 6 }}};

    std::vector<int> vec;
    std::cout << "Recorrido en preorden: " << std::endl;
    tree.preorder();
    std::cout << std::endl;

    std::cout << "Recorrido en inorder: " << std::endl;
    tree.inorder();
    std::cout << std::endl;

    std::cout << "Recorrido en postorden: " << std::endl;
    tree.postorder();
    std::cout << std::endl;

    return 0;
}
```

Recorrido en preorden:  
7 4 9 5 4 10 6  
Recorrido en inorder:  
9 4 5 7 10 4 6  
Recorrido en postorden:  
9 5 4 10 6 4 7

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

# Parametrizando el recorrido de un árbol

Manuel Montenegro Montes

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

# Recorrer un árbol

- Recorrer un árbol significa *visitar* todos sus nodos.
- **Visitar** un nodo significa realizar una acción que dependa del valor contenido en dicho nodo.

Hasta ahora:

```
template<typename T>
void BinTree<T>::preorder(const NodePointer &node) {
    if (node != nullptr) {
        std::cout << node->elem << " ";
        preorder(node->left);
        preorder(node->right);
    }
}
```

# Parametrizar el recorrido

- Podemos parametrizar el recorrido con respecto a la acción a realizar en cada nodo.

```
template<typename T>
void BinTree<T>::preorder(const NodePointer &node) {
    if (node != nullptr) {
        std::cout << node->elem << " ";
        preorder(node->left);
        preorder(node->right);
    }
}
```



# Parametrizar el recorrido

- Podemos parametrizar el recorrido con respecto a la acción a realizar en cada nodo.

```
template<typename T>
template<typename U>
void BinTree<T>::preorder(const NodePointer &node, U func) {
    if (node != nullptr) {
        func(node->elem);
        preorder(node->left, func);
        preorder(node->right, func);
    }
}
```



# Parametrizar el recorrido

- Modificamos también el método `preorden()` de la clase, que realiza la llamada inicial a la función recursiva:

```
template<class T>
class BinTree {
public:
    ...
    template <typename U>
    void preorder(U func) const {
        preorder(root_node, func);
    }
    ...
};
```



# Ejemplos

- Supongamos que tenemos el siguiente árbol:

```
BinTree<int> tree {{{ 9 }, 4, { 5 }}, 7, {{ 10 }, 4, { 6 }}};
```

- Imprimir el recorrido en preorden:

```
tree.preorder([] (int x) { std::cout << x << " "; });
```

- Imprimir solamente los elementos pares:

```
tree.preorder([] (int x) {
    if (x % 2 == 0) {
        std::cout << x << " ";
    }
});
```

# Ejemplos

- Supongamos que tenemos el siguiente árbol:

```
BinTree<int> tree {{{ 9 }, 4, { 5 }}, 7, {{ 10 }, 4, { 6 }}};
```

- Sumar los elementos del árbol:

```
int acum = 0;  
tree.preorder([&acum])(int x) { acum += x; };  
std::cout << acum << std::endl;
```

- Contar el número de elementos de un árbol:

```
int num_elems = 0;  
tree.preorder([&num_elems])(int x) { num_elems++; };  
std::cout << num_elems << std::endl;
```

# Ejemplos

- Supongamos que tenemos el siguiente árbol:

```
BinTree<int> tree {{{ 9 }, 4, { 5 }}, 7, {{ 10 }, 4, { 6 }}};
```

- Añadir los elementos del árbol a una lista:

```
std::vector<int> v;
tree.preorder([&v])(int x) { v.push_back(x); };
```

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

# EL TAD Conjunto

Manuel Montenegro Montes

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

# Conjuntos

- Un **conjunto** es una colección de elementos del mismo tipo.
- ¿Cuál es la diferencia entre un conjunto y una lista?

## Listas

- El orden de los elementos es relevante:

$$[1, 4, 5] \neq [4, 5, 1]$$

- Pueden contener elementos duplicados:

$$[1, 4, 4, 5] \neq [1, 4, 5]$$

## Conjunto

- No existe el concepto de orden entre elementos:

$$\{1, 4, 5\} = \{4, 5, 1\}$$

- La existencia de duplicados es irrelevante:

$$\{1, 4, 5\} \cup \{4\} = \{1, 4, 5\}$$

# Modelo de conjuntos

- Varias formas de implementar un conjunto.
- Cuando el conjunto está implementado y tan solo tenemos que utilizarlo, pensamos en términos del **modelo**.
- Cada instancia del TAD Conjunto representa un **conjunto finito**.

$$\{x_1, x_2, x_3, \dots, x_n\}$$

# Operaciones en el TAD conjunto

- Constructoras:
  - Crear un conjunto vacío: ***create\_empty***
- Mutadoras:
  - Añadir un elemento al conjunto: ***insert***
  - Eliminar un elemento del conjunto: ***erase***
- Observadoras:
  - Averiguar si un elemento está en el conjunto: ***contains***
  - Saber si el conjunto está vacío: ***empty***
  - Saber el tamaño del conjunto: ***size***

# Operaciones constructoras y mutadoras

{ *true* }

***create\_empty()*** → (*S*: Set)

{  $S = \emptyset$  }

{ *true* }

***insert***(*x*: Elem, *S*: Set)

{  $S = old(S) \cup \{x\}$  }

{ *true* }

***erase***(*x*: Elem, *S*: Set)

{  $S = old(S) - \{x\}$  }

# Operaciones observadoras

$\{ \text{ true } \}$

**contains**( $x: \text{Elem}$ ,  $S: \text{Set}$ )  $\rightarrow (b: \text{bool})$

$\{ b \Leftrightarrow x \in S \}$

$\{ \text{ true } \}$

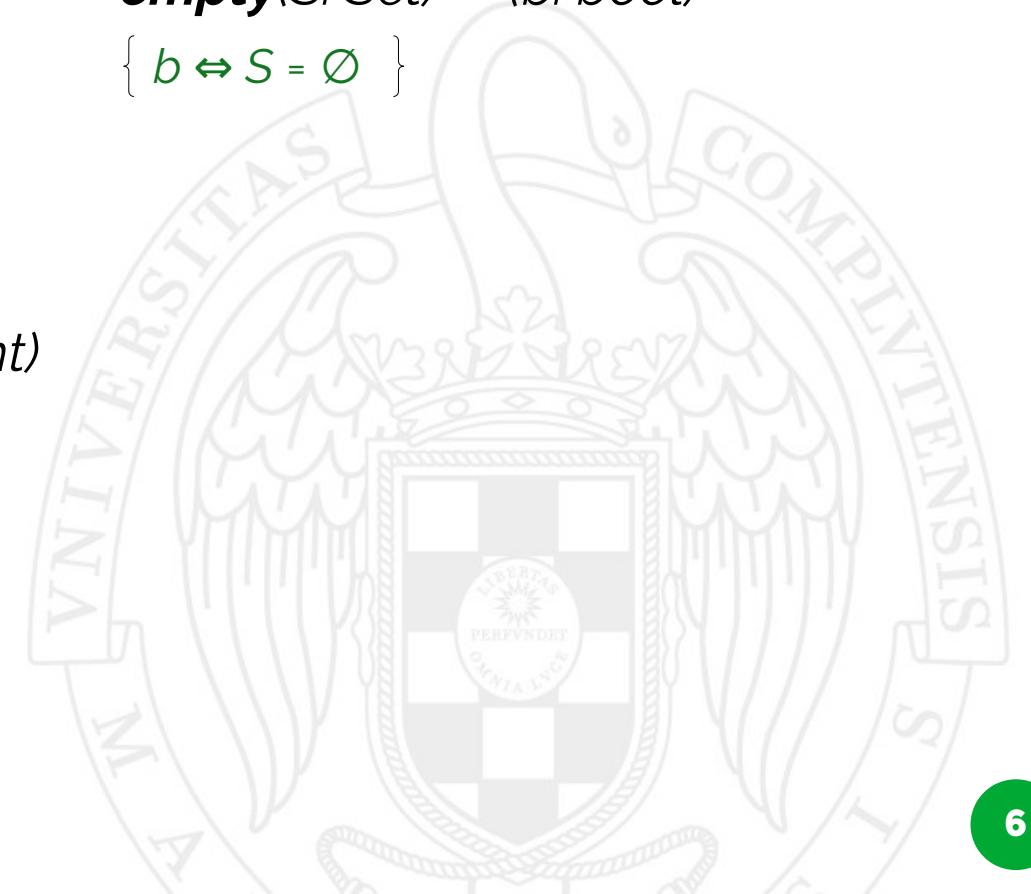
**empty**( $S: \text{Set}$ )  $\rightarrow (b: \text{bool})$

$\{ b \Leftrightarrow S = \emptyset \}$

$\{ \text{ true } \}$

**size**( $S: \text{Set}$ )  $\rightarrow (n: \text{int})$

$\{ n = |S| \}$



# Interfaz en C++

```
class set {  
public:  
    set();  
    set(const set &other);  
    ~set();  
  
    void insert(const T &elem);  
    void erase(const T &elem);  
  
    bool contains(const T &elem) const;  
    int size() const;  
    bool empty() const;  
  
private:  
    // ???  
};
```



# Dos implementaciones

- Mediante **listas**.
- Mediante **árboles binarios de búsqueda**.



ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

# Implementación del TAD Conjunto mediante listas ordenadas

Manuel Montenegro Montes

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

# Operaciones en el TAD Conjunto

- Constructoras:
  - Crear un conjunto vacío: ***create\_empty***
- Mutadoras:
  - Añadir un elemento al conjunto: ***insert***
  - Eliminar un elemento del conjunto: ***erase***
- Observadoras:
  - Averiguar si un elemento está en el conjunto: ***contains***
  - Saber si el conjunto está vacío: ***empty***
  - Saber el tamaño del conjunto: ***size***

# Representación mediante listas ordenadas

- Clase que contiene un único atributo: `list_elems`, de tipo Lista.
- El atributo `list_elems` contiene los elementos del conjunto que se quiere representar de modo que:
  - `list_elems` almacena los elementos en orden ascendente.
  - `list_elems` no almacena duplicados.

$\{4, 5, 7, 3, 1\}$

`list_elems: [1, 3, 4, 5, 7]`

# Representación mediante listas ordenadas

- Clase que contiene un único atributo: `list_elems`, de tipo Lista.
- El atributo `list_elems` contiene los elementos del conjunto que se quiere representar de modo que:
  - `list_elems` almacena los elementos en orden ascendente.
  - `list_elems` no almacena duplicados.

```
template <typename T>
class SetList {
public:
    ...
private:
    using List = ???;
    List list_elems;
};
```



The code snippet shows a template class `SetList`. It has a public section containing an ellipsis (`...`). In the private section, there is a `using` declaration followed by a question mark twice (`using List = ???;`). Below this, there is a brace group that encloses two types: `std::vector<T>` and `std::list<T>`.

# Representación mediante listas ordenadas

- Función de abstracción:

Si  $s$  es una instancia de la clase `SetList`:

$$f(s) = \{ s.list\_elems[i] \mid 0 \leq i < s.list\_elems.size() \}$$

- Invariante de representación

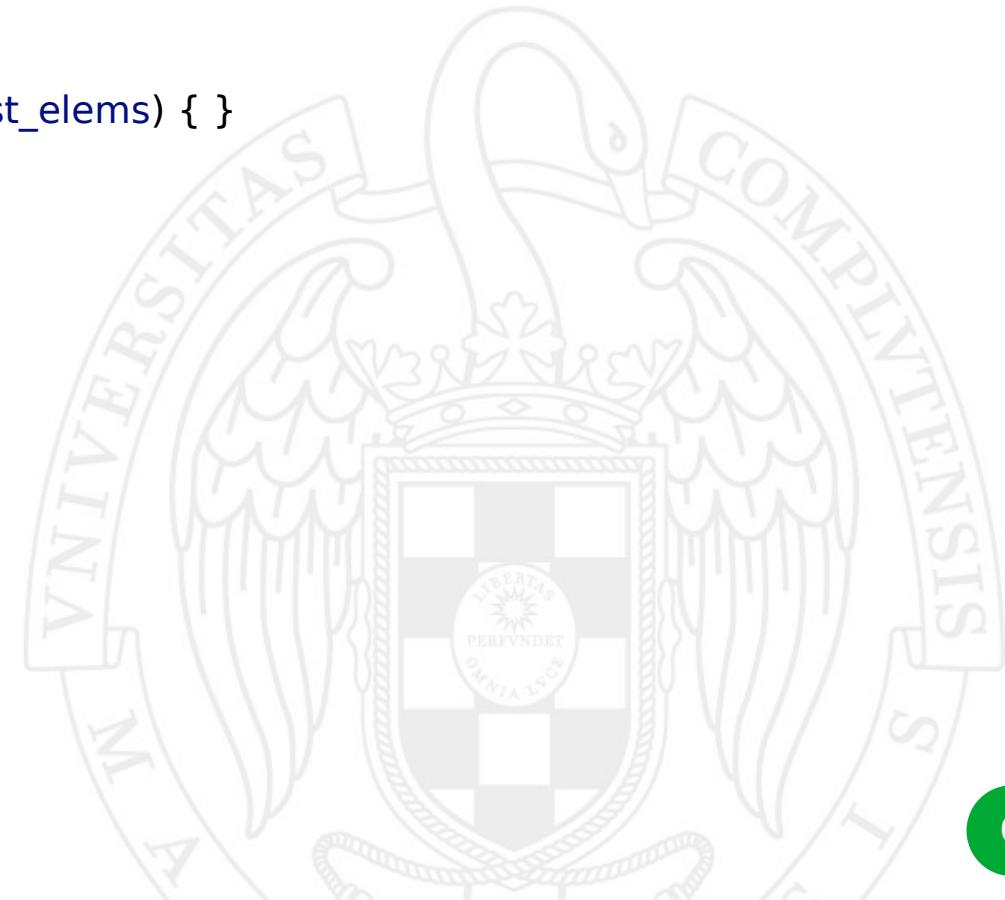
$$I(s) \equiv \forall i, j: 0 \leq i < j < s.list\_elems.size()$$

$$\implies s.list\_elems[i] < s.list\_elems[j]$$

# Operaciones constructoras

```
template <typename T>
class SetList {
public:
    SetList() { }
    SetList(const SetList &other): list_elems(other.list_elems) { }
    ~SetList() { }

private:
    ...
    List list_elems;
};
```



# Operaciones observadoras

```
template <typename T>
class SetList {
public:
...
    bool contains(const T &elem) const { ... }

    int size() const {
        return list_elems.size();
    }

    bool empty() const {
        return list_elems.empty();
    }

private:
...
    List list_elems;
};
```



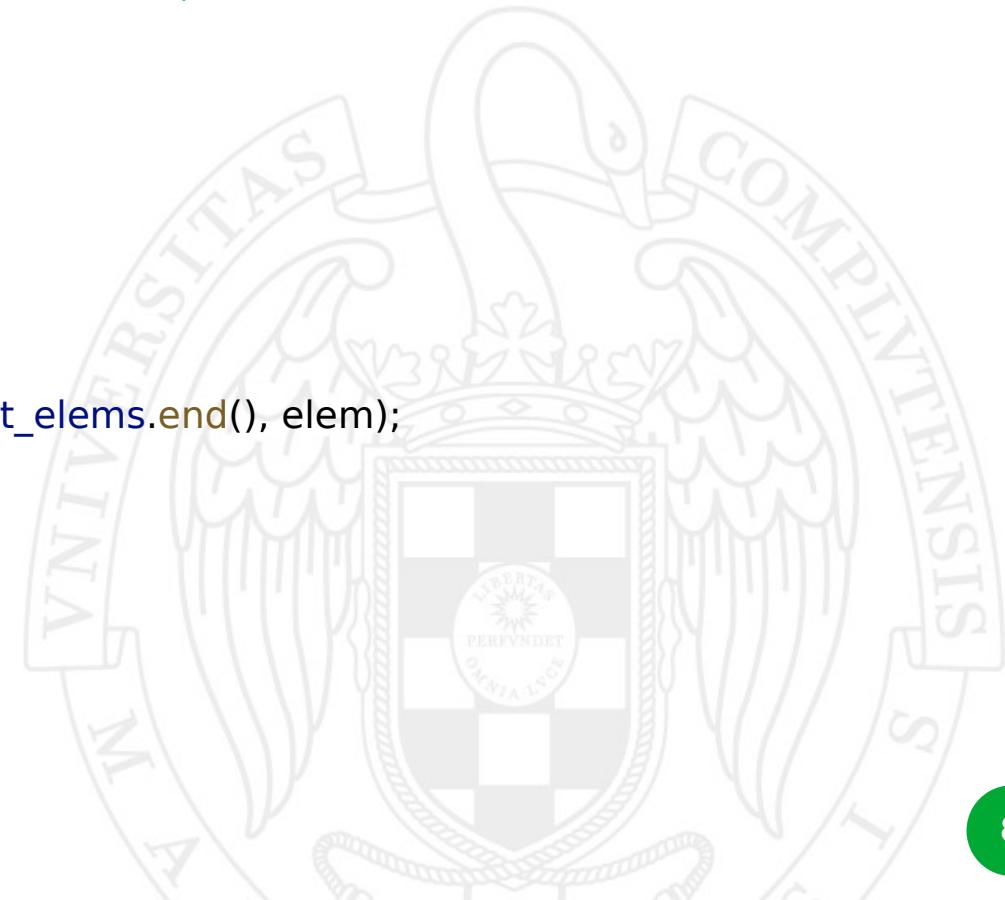
# Operación *contains*

- Utilizamos una función de búsqueda binaria

```
bool binary_search(iterator first, iterator last, const T& val)
```

definida en <algorithm>

```
template <typename T>
class SetList {
public:
    bool contains(const T &elem) const {
        return std::binary_search(list_elems.begin(), list_elems.end(), elem);
    }
    ...
};
```



# Operaciones mutadoras

```
template <typename T>
class SetList {
public:
    ...
    void insert(const T &elem) { ... }
    void erase(const T &elem) { ... }

private:
    ...
    List list_elems;
};
```



# Operación *insert*

- Necesitamos insertar el elemento en `list_elems` de modo que la lista permanezca ordenada.
- Podemos utilizar búsqueda binaria para saber dónde insertar el elemento.
- Problema: `binary_search` solamente indica si un elemento está en la lista o no.
- Pero tenemos la función `lower_bound`:

```
iterator lower_bound(iterator begin, iterator end, const T &elem)
```

Devuelve un iterador al primer elemento contenido entre `begin` y `end` que no es estrictamente menor que `elem`.

Si todos son menores que `elem`, devuelve `end`.

Los elementos que hay entre `begin` y `end` han de estar ordenados.

# Ejemplo: lower\_bound

```
std::vector<int> v = {1, 5, 8, 10, 20};  
auto it_pos = std::lower_bound(v.begin(), v.end(), 9);  
std::cout << *it_pos << std::endl;
```

# Operación *insert*

```
template <typename T>
class SetList {
public:
...
void insert(const T &elem) {
    auto position = std::lower_bound(list_elems.begin(), list_elems.end(), elem);
    if (position == list_elems.end() || *position != elem) {
        list_elems.insert(position, elem);
    }
}

private:
...
List list_elems;
};
```



# Operación *erase*

```
template <typename T>
class SetList {
public:
...
void erase(const T &elem) {
    auto position = std::lower_bound(list_elems.begin(), list_elems.end(), elem);
    if (position != list_elems.end() && *position == elem) {
        list_elems.erase(position);
    }
}
private:
...
List list_elems;
};
```

# ¿Qué utilizo?

```
template <typename T>
class SetList {
public:
    ...
private:
    using List = ???  
    List list_elems;
};
```

{ std::vector<T>  
 std::list<T>

# Coste de las operaciones auxiliares

Operación	<code>std::vector</code>	<code>std::list</code>
<code>binary_search</code>	$O(\log n)$	$O(n)$ ( <i>no es búsq. binaria</i> )
<code>lower_bound</code>	$O(\log n)$	$O(n)$ ( <i>no es búsq. binaria</i> )
<code>insert</code> ( <i>en listas</i> )	$O(n)$	$O(1)$
<code>erase</code> ( <i>en listas</i> )	$O(n)$	$O(1)$

$n$  = longitud de *list\_elems*

# Coste de las operaciones

Operación	<code>std::vector</code>	<code>std::list</code>
<i>constructor</i>	$O(1)$	$O(1)$
<code>empty</code>	$O(1)$	$O(1)$
<code>size</code>	$O(1)$	$O(1)$
<code>contains</code>	$O(\log n)$	$O(n)$
<code>insert</code>	$O(\log n) + O(n)$	$O(n) + O(1)$
<code>erase</code>	$O(\log n) + O(n)$	$O(n) + O(1)$

$n$  = número de elementos del conjunto

# Coste de las operaciones

Operación	<code>std::vector</code>	<code>std::list</code>
<i>constructor</i>	$O(1)$	$O(1)$
<code>empty</code>	$O(1)$	$O(1)$
<code>size</code>	$O(1)$	$O(1)$
<code>contains</code>	$O(\log n)$	$O(n)$
<code>insert</code>	$O(n)$	$O(n)$
<code>erase</code>	$O(n)$	$O(n)$

$n$  = número de elementos del conjunto

# ¿Qué utilizo?

```
template <typename T>
class SetList {
public:
    ...
private:
    using List = std::vector<T>;
    List list_elems;
};
```

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

# Árboles binarios de búsqueda

Manuel Montenegro Montes

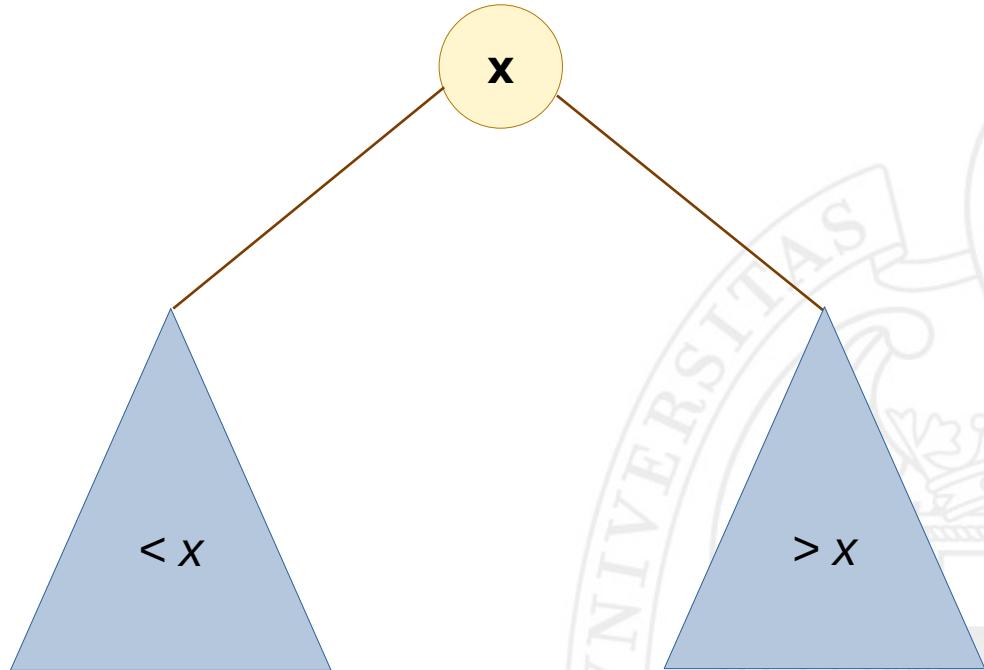
Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Árboles binarios de búsqueda (ABBs)

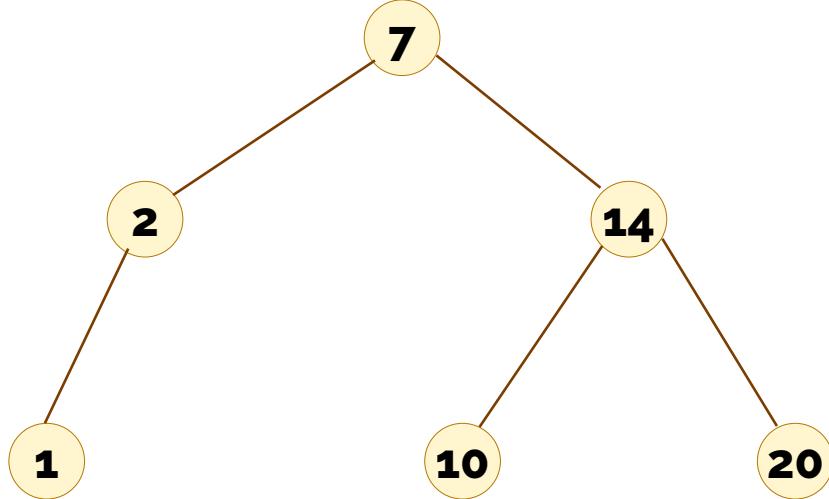
Un árbol binario es de búsqueda si:

- Es un árbol vacío, o bien,
- Es una hoja, o bien,
- Su raíz es un nodo interno, y además:
  - Todos los elementos de su **hijo izquierdo** son estrictamente **menores** que la raíz.
  - Todos los elementos de su **hijo derecho** son estrictamente **mayores** que la raíz.
  - Los hijos izquierdo y derecho son árboles de búsqueda.

# Árboles binarios de búsqueda



# Ejemplos



Árbol de  
búsqueda



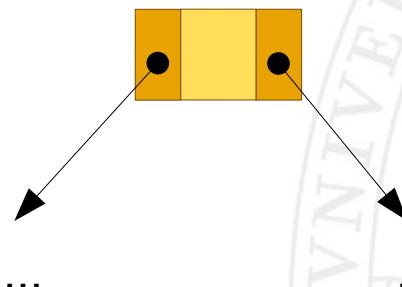
No es árbol  
de búsqueda

No es árbol  
de búsqueda

# Representación mediante nodos

```
template <typename T>
struct Node {
    T elem;
    Node *left, *right;

    Node(Node *left, const T &elem, Node *right): left(left), elem(elem), right(right) { }
};
```



# Búsqueda en un ABB

- Queremos implementar una función que determine si un elemento se encuentra en un árbol de búsqueda

```
bool search(const Node *root, const T &elem);
```

- La función determina si el `elem` se encuentra dentro del nodo `root` o en alguno de sus descendientes.
- Distinguimos cuatro casos.

# Caso 1: Árbol vacío

```
bool search(const Node *root, const T &elem);
```

- Si `root = nullptr`, el árbol es vacío.
- En ese caso, `elem` no pertenece al árbol.
- Devolvemos `false`.

```
bool search(const Node *root, const T &elem) {  
    if (root == nullptr) {  
        return false;  
    } else { ... }  
}
```

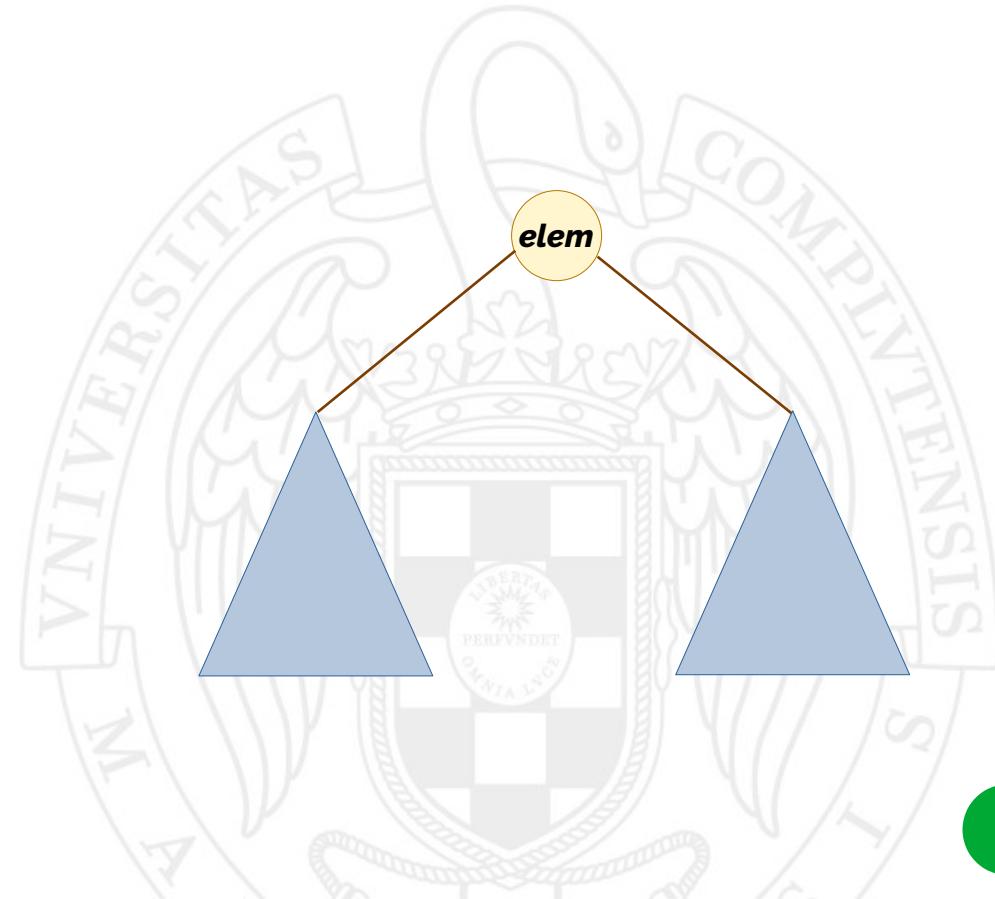


# Caso 2: elem == raíz del árbol

```
bool search(const Node *root, const T &elem);
```

- En este caso, hemos encontrado elem en el árbol. Devolvemos true.

```
bool search(const Node *root, const T &elem) {  
    if (root == nullptr) {  
        return false;  
    } else if (elem == root->elem) {  
        return true;  
    } else { ... }  
}
```

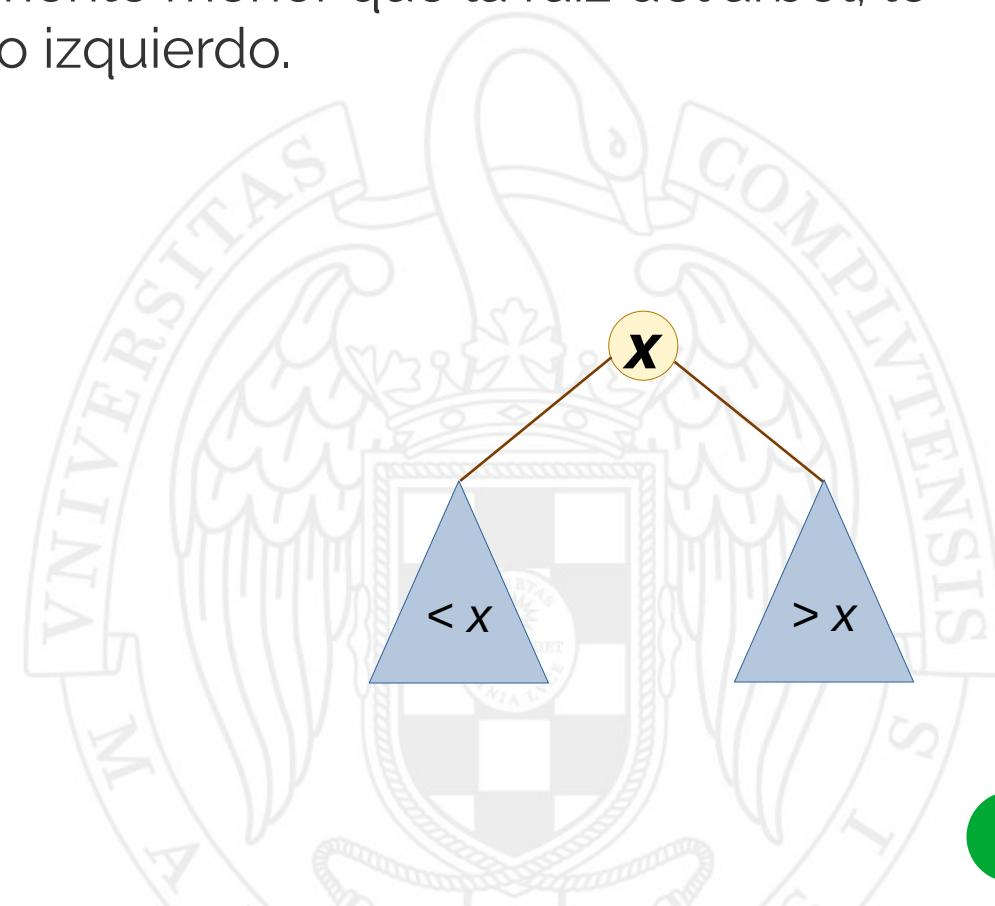


# Caso 3: elem < raíz del árbol

```
bool search(const Node *root, const T &elem);
```

- Si el elemento a buscar es estrictamente menor que la raíz del árbol, lo buscamos recursivamente en el hijo izquierdo.

```
bool search(const Node *root, const T &elem) {  
    if (root == nullptr) {  
        return false;  
    } else if (elem == root->elem) {  
        return true;  
    } else if (elem < root->elem) {  
        return search(root->left, elem);  
    } else { ... }  
}
```

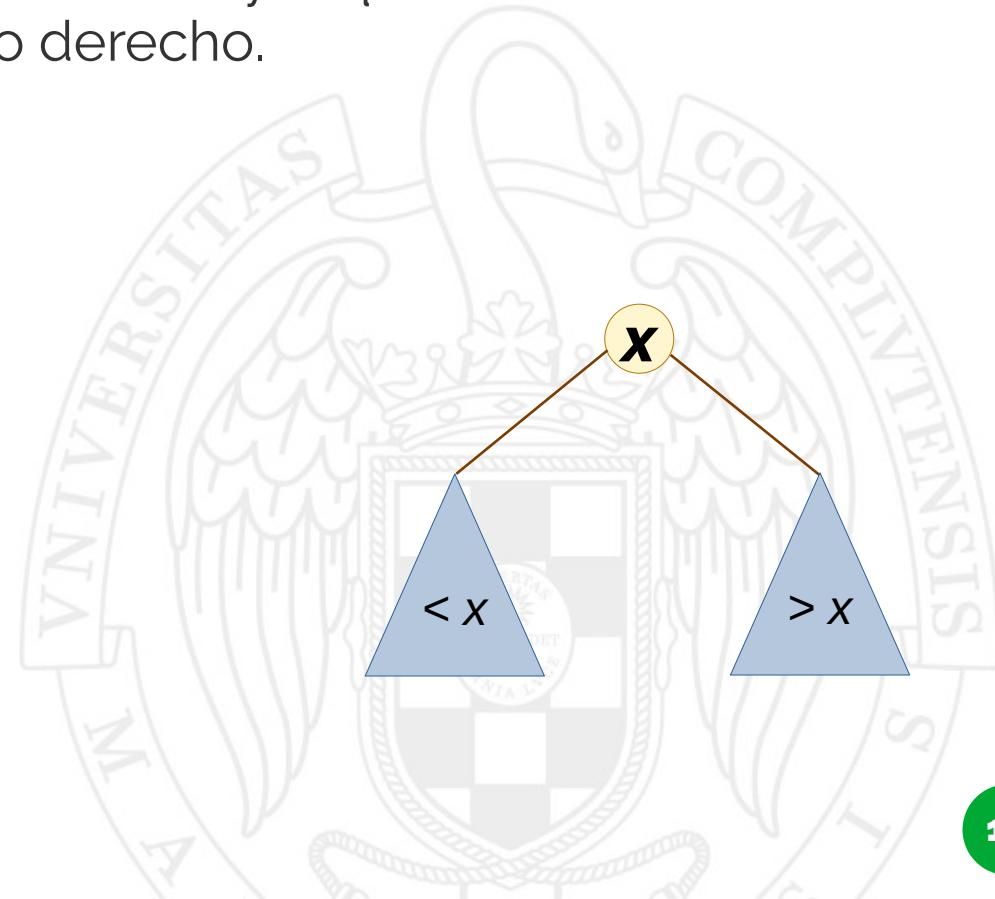


# Caso 4: elem > raíz del árbol

```
bool search(const Node *root, const T &elem);
```

- Si el elemento a buscar es estrictamente mayor que la raíz del árbol, lo buscamos recursivamente en el hijo derecho.

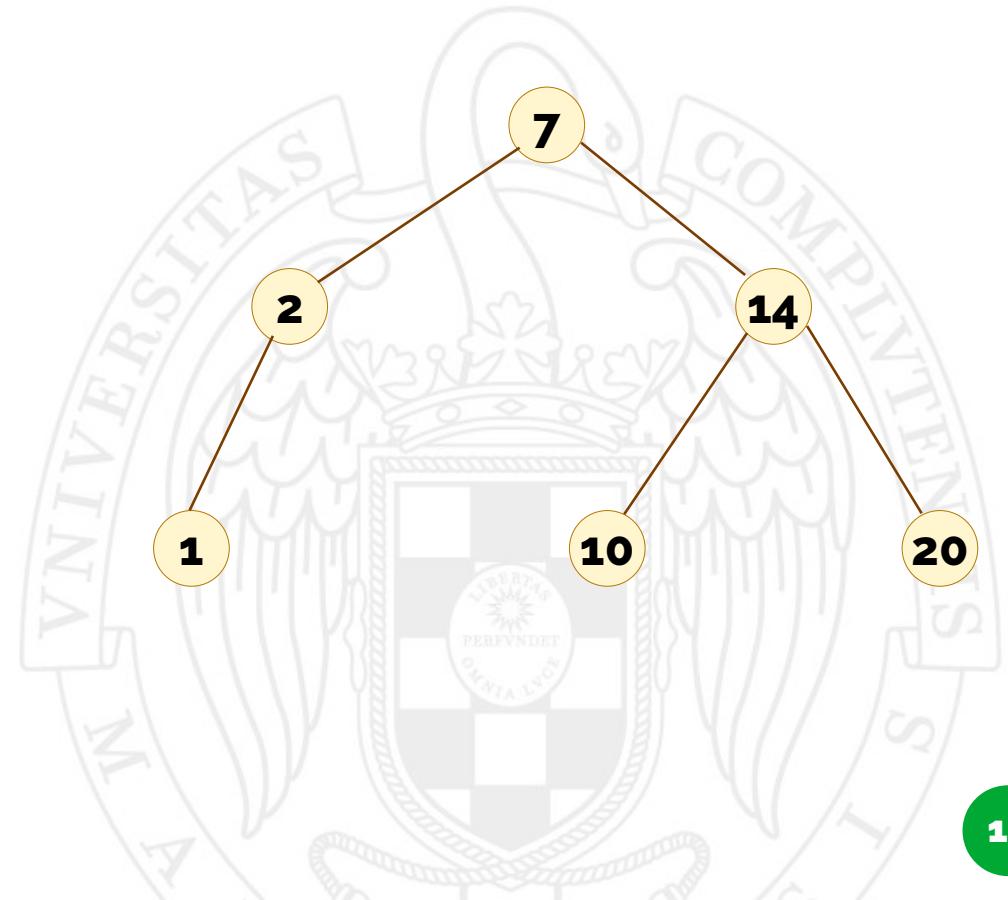
```
bool search(const Node *root, const T &elem) {  
    if (root == nullptr) {  
        return false;  
    } else if (elem == root->elem) {  
        return true;  
    } else if (elem < root->elem) {  
        return search(root->left, elem);  
    } else {  
        return search(root->right, elem);  
    }  
}
```



# Ejemplo

- Buscamos el 10

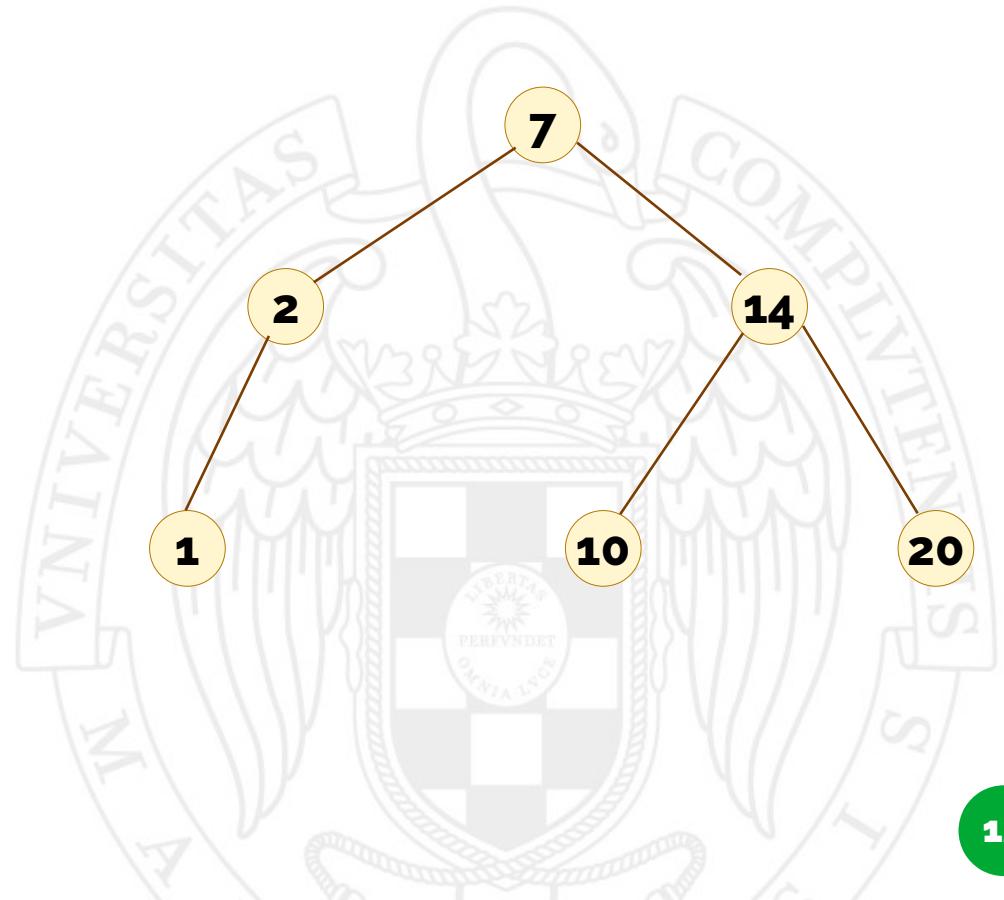
```
bool search(const Node *root, const T &elem) {  
    if (root == nullptr) {  
        return false;  
    } else if (elem == root->elem) {  
        return true;  
    } else if (elem < root->elem) {  
        return search(root->left, elem);  
    } else {  
        return search(root->right, elem);  
    }  
}
```



# Ejemplo

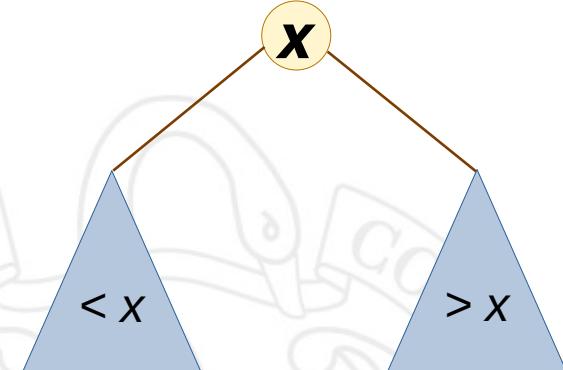
- Buscamos el 3

```
bool search(const Node *root, const T &elem) {  
    if (root == nullptr) {  
        return false;  
    } else if (elem == root->elem) {  
        return true;  
    } else if (elem < root->elem) {  
        return search(root->left, elem);  
    } else {  
        return search(root->right, elem);  
    }  
}
```



# Coste de la función search

```
bool search(const Node *root, const T &elem) {  
    if (root == nullptr) {  
        return false;  
    } else if (elem == root->elem) {  
        return true;  
    } else if (elem < root->elem) {  
        return search(root->left, elem);  
    } else {  
        return search(root->right, elem);  
    }  
}
```



# Coste de la función search

- En el caso peor, la función search desciende desde la raíz hasta las hojas.
- El coste en tiempo de la función search es **lineal** con respecto a la altura del árbol.

*¿Y con respecto al número de nodos?*

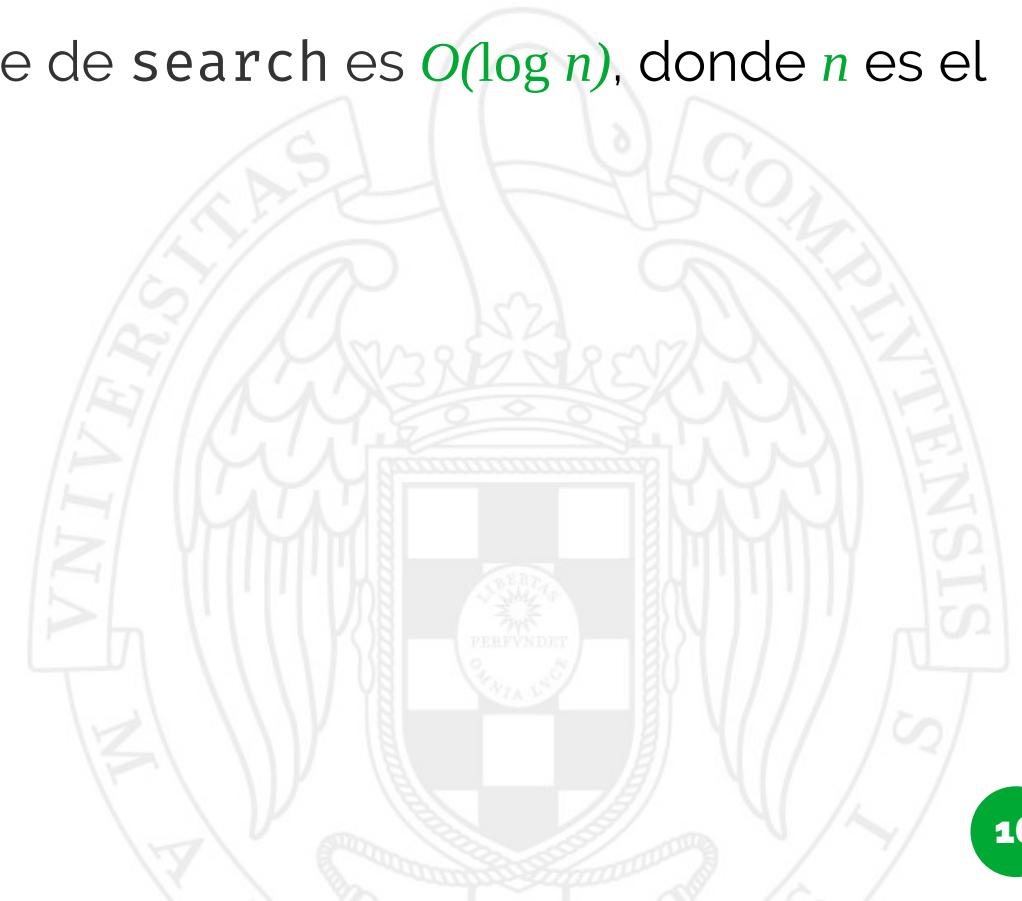
# Recordatorio

Sea  $h$  la altura de un árbol y  $n$  su número de nodos.

- Si el árbol es **degenerado**,  $h \in O(n)$
- Si el árbol es **equilibrado**,  $h \in O(\log n)$
- Si no sabemos nada acerca de si el árbol está equilibrado o no, el caso peor es el árbol degenerado.

# Coste de la función search

- Si el árbol es **degenerado**, el coste de search es  $O(n)$ , donde  $n$  es el número de nodos del árbol.
- Si el árbol está **equilibrado**, el coste de search es  $O(\log n)$ , donde  $n$  es el número de nodos del árbol.



ESTRUCTURAS DE DATOS

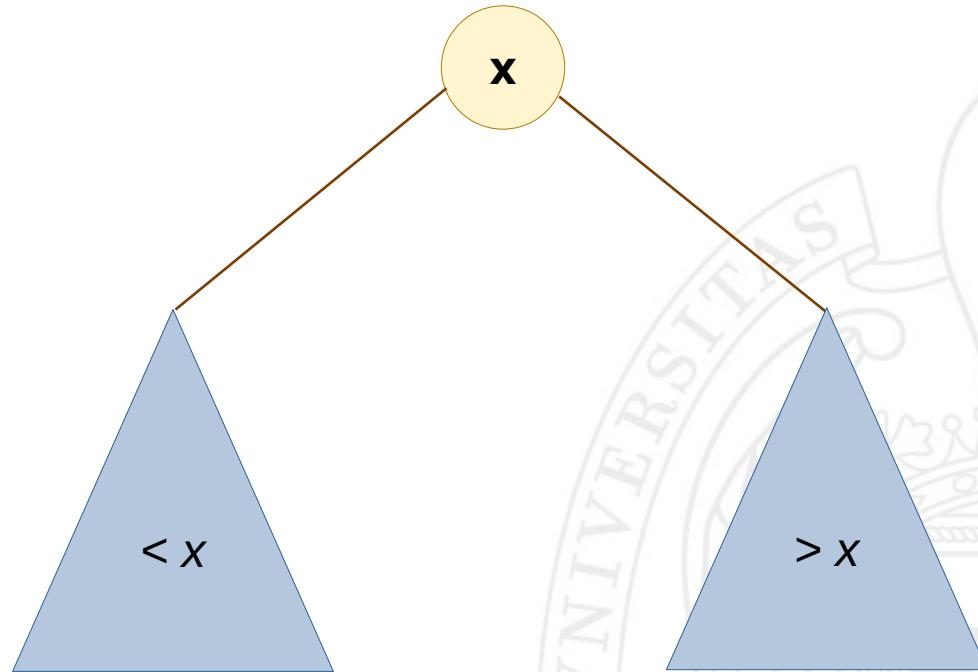
TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

# Inserción en ABBs

Manuel Montenegro Montes

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

# Recordatorio: árboles binarios de búsqueda

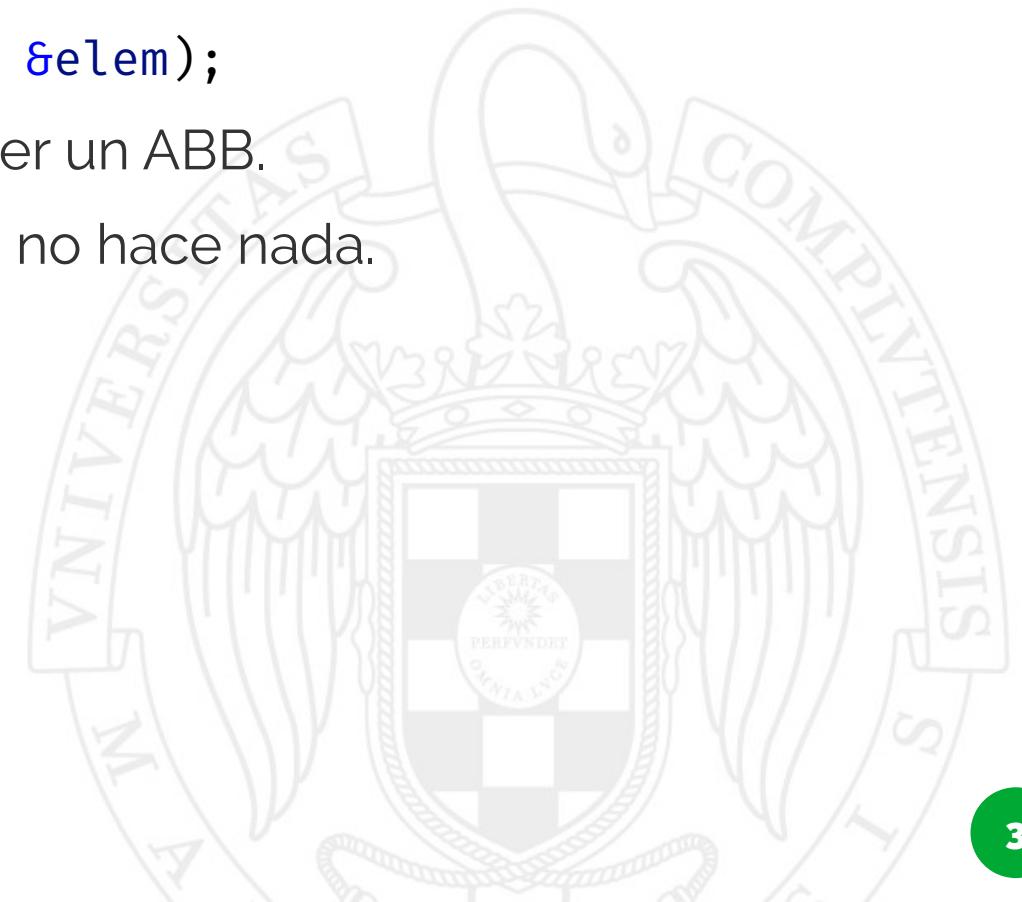


# Objetivo

- Implementar una función `insert(root, elem)`, que añada un nodo con el valor `elem` al ABB cuya raíz es `root`.

```
void insert(Node *root, const T &elem);
```

- El árbol resultante también ha de ser un ABB.
- Si `elem` ya se encuentra en el ABB, no hace nada.



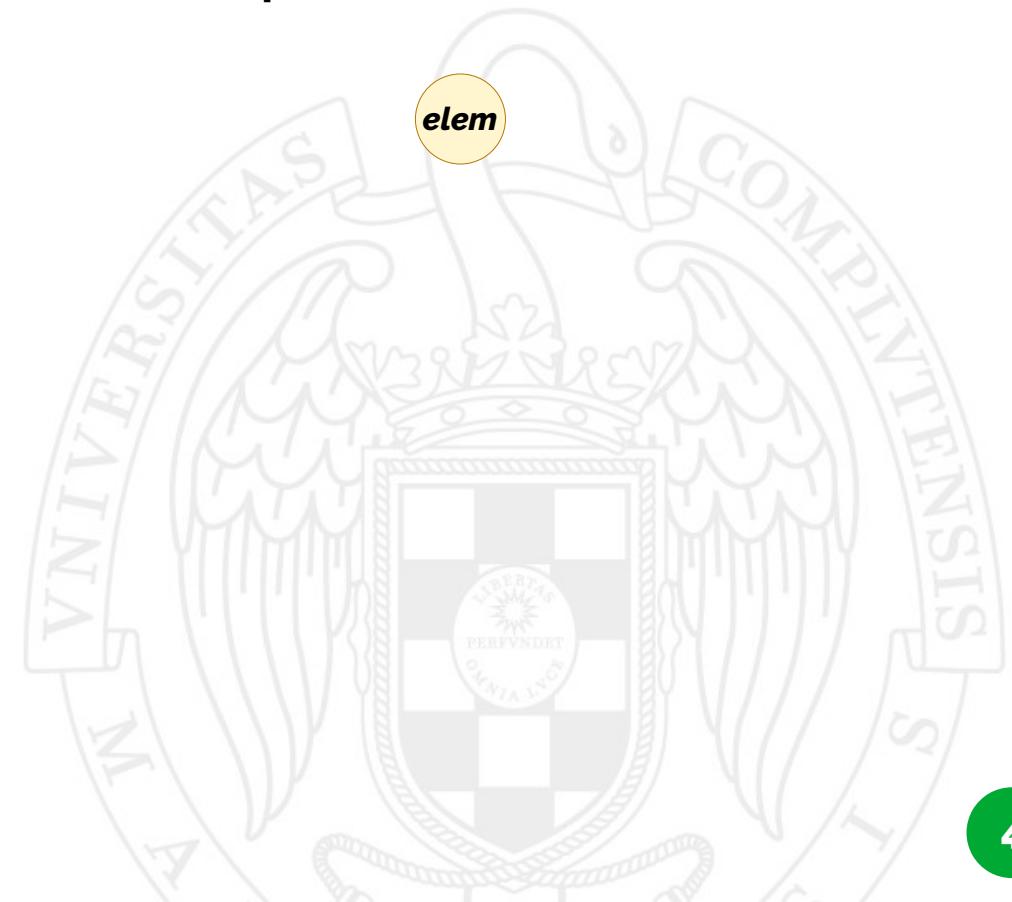
# Caso 1: Árbol vacío (root = nullptr)

Antes de la inserción

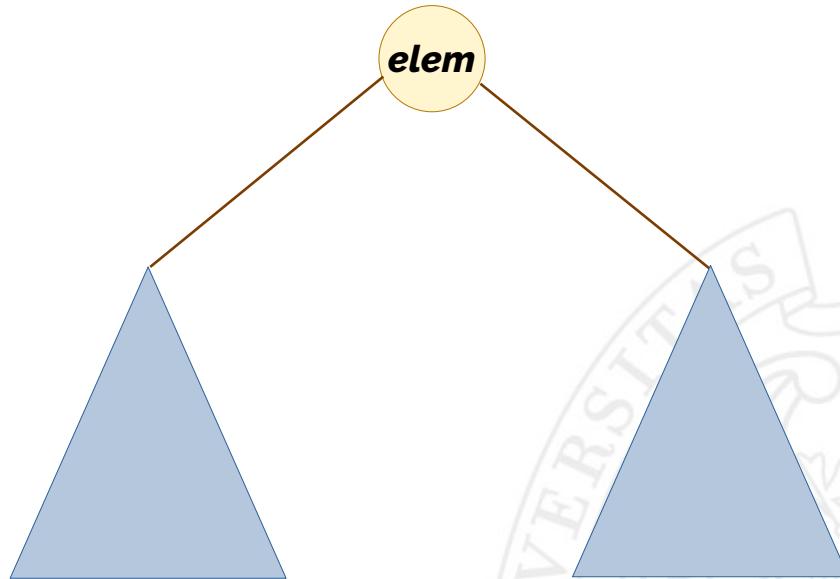
—

Después de la inserción

elem

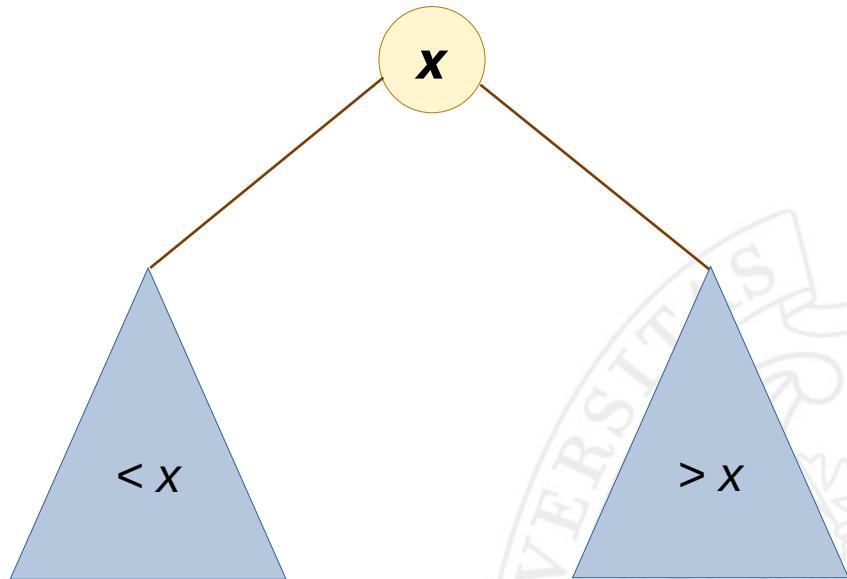


## Caso 2: elem coincide con la raíz



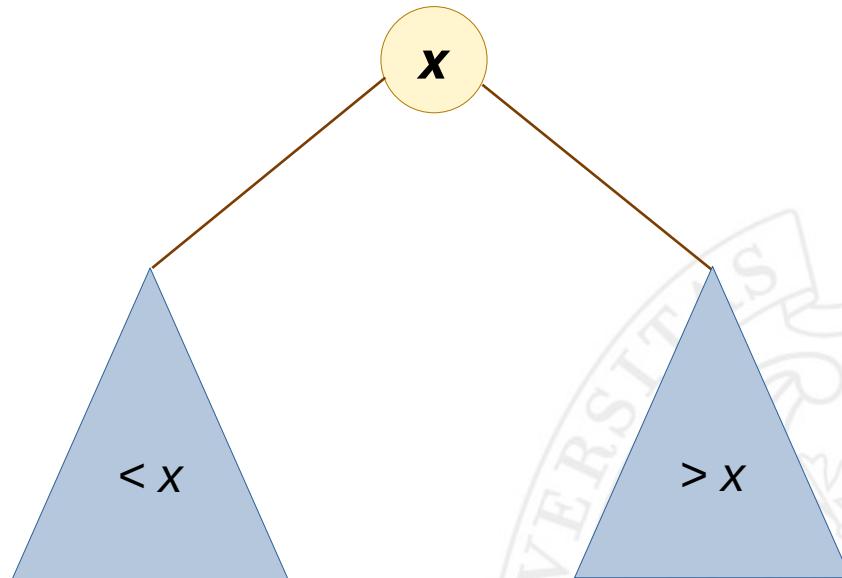
- El elemento que quiero insertar ya está en el árbol. No hacemos nada.

## Caso 3: elem < raíz



- Insertamos recursivamente elem en el hijo izquierdo de la raíz.

## Caso 4: elem > raíz



- Insertamos recursivamente elem en el hijo derecho de la raíz.

# Antes de implementar

- En uno de los casos **la raíz del árbol cambia**.  
Caso 1: si el árbol es vacío, la raíz acaba siendo el nodo recién creado.
- Por tanto, la función `insert` debe devolver también **la nueva raíz del árbol**.
- En lugar de:

```
void insert(Node *root, const T &elem);
```

tendremos:

```
Node * insert(Node *root, const T &elem);
```



Nueva raíz

# Implementación

```
Node * insert(Node *root, const T &elem) {  
    if (root == nullptr) {  
        return new Node(nullptr, elem, nullptr);  
    } else if (elem < root->elem) {  
  
    } else if (root->elem < elem) {  
  
    } else {  
    }  
}
```

- **Caso 1:** árbol vacío.

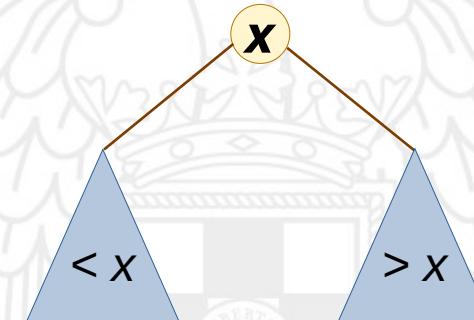
Creamos un nodo con el valor que se quiere insertar, y ese nodo es la nueva raíz del árbol.

# Implementación

```
Node * insert(Node *root, const T &elem) {  
    if (root == nullptr) {  
  
    } else if (elem < root->elem) {  
        Node *new_root_left = insert(root->left, elem);  
        root->left = new_root_left;  
        return root;  
    } else if (root->elem < elem) {  
  
    } else {  
    }  
}
```

- **Caso 3:**  $\text{elem} < \text{raiz}$

Insertamos en el hijo izquierdo.  
Conectamos la raíz con la nueva  
raíz del hijo izquierdo.

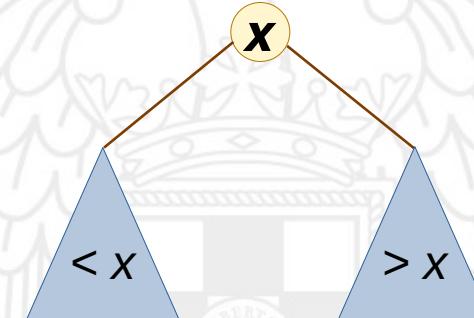


# Implementación

```
Node * insert(Node *root, const T &elem) {  
    if (root == nullptr) {  
  
    } else if (elem < root->elem) {  
  
    } else if (root->elem < elem) {  
        Node *new_root_right = insert(root->right, elem);  
        root->right = new_root_right;  
        return root;  
    } else {  
  
    }  
}
```

- **Caso 4:  $\text{elem} > \text{raiz}$**

Dual al anterior

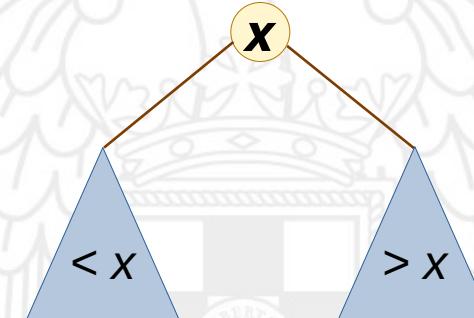


# Implementación

```
Node * insert(Node *root, const T &elem) {  
    if (root == nullptr) {  
  
    } else if (elem < root->elem) {  
  
    } else if (root->elem < elem) {  
  
    } else {  
        return root;  
    }  
}
```

- **Caso 2:**  $\text{elem} == \text{raiz}$

No se hace nada. La raíz no varía.



# Implementación

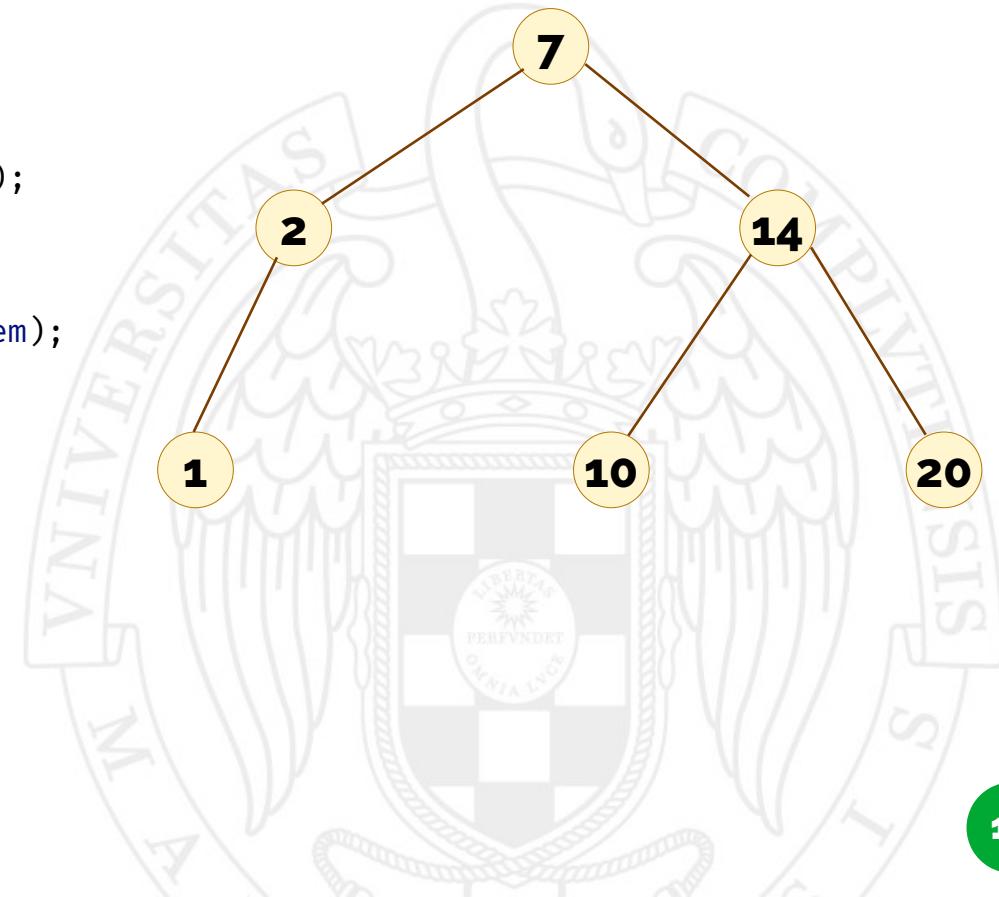
```
Node * insert(Node *root, const T &elem) {
    if (root == nullptr) {
        return new Node(nullptr, elem, nullptr);
    } else if (elem < root->elem) {
        Node *new_root_left = insert(root->left, elem);
        root->left = new_root_left;
        return root;
    } else if (root->elem < elem) {
        Node *new_root_right = insert(root->right, elem);
        root->right = new_root_right;
        return root;
    } else {
        return root;
    }
}
```



# Ejemplo

- Insertar el valor 9

```
Node * insert(Node *root, const T &elem) {  
    if (root == nullptr) {  
        return new Node(nullptr, elem, nullptr);  
    } else if (elem < root->elem) {  
        Node *new_root_left = insert(root->left, elem);  
        root->left = new_root_left;  
        return root;  
    } else if (root->elem < elem) {  
        Node *new_root_right = insert(root->right, elem);  
        root->right = new_root_right;  
        return root;  
    } else {  
        return root;  
    }  
}
```



# Coste en tiempo

- En el caso peor, el nodo se inserta en la rama más larga del árbol.
- Por tanto, si  $h$  es la altura del árbol, el coste es  $O(h)$ .
- Y si  $n$  es el número de nodos del árbol:
  - Si el árbol está equilibrado, el coste es  $O(\log n)$ .
  - Si no, el coste es  $O(n)$  en el caso peor.

ESTRUCTURAS DE DATOS

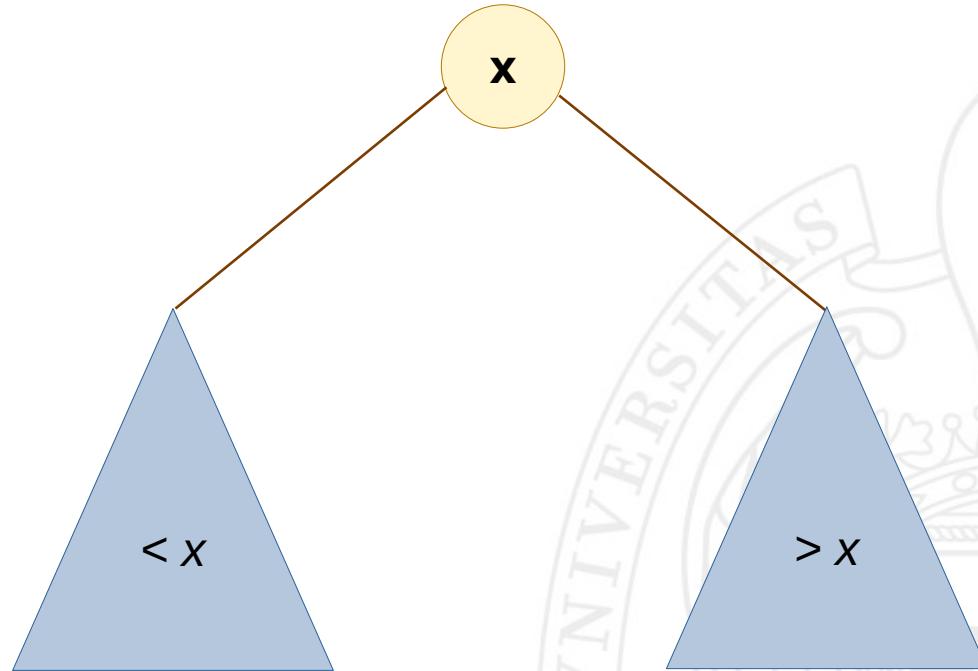
TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

# Eliminación en ABBs

Manuel Montenegro Montes

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

# Recordatorio: árboles binarios de búsqueda



# Objetivo

- Implementar una función `erase(root, elem)`, que elimine el nodo que contenga `elem` del ABB cuya raíz es `root`.
- El árbol resultante también ha de ser un ABB.
- Si `elem` no se encuentra en el ABB, no hace nada.

```
void erase(Node *root, const T &elem);
```

- En algunos casos, la raíz del árbol va a cambiar. Por tanto:

```
Node * erase(Node *root, const T &elem);
```

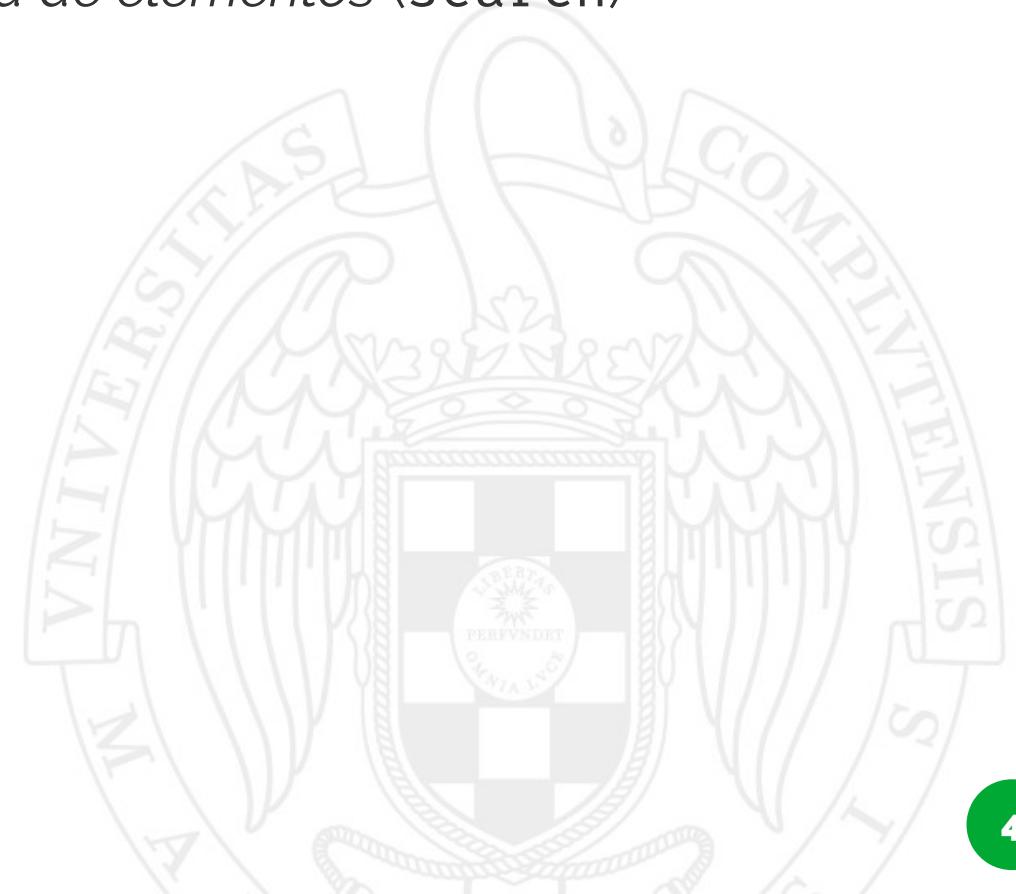
# Dos fases

- 1) Buscar el nodo a eliminar.

*Similar al algoritmo de búsqueda de elementos (search)*

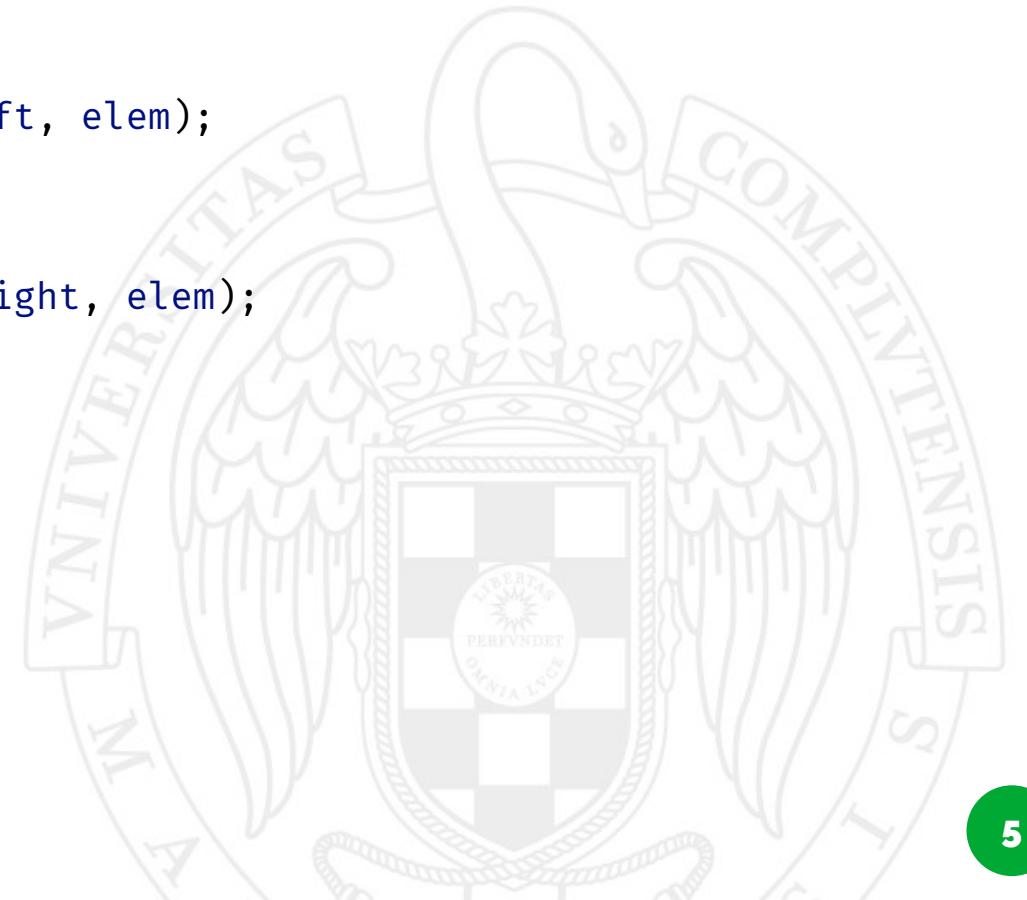
- 2) Si se encuentra, eliminarlo.

*...y poner otra cosa en su lugar.*



# Fase 1: búsqueda del nodo

```
Node * erase(Node *root, const T &elem) {
    if (root == nullptr) {
        return root;
    } else if (elem < root->elem) {
        Node *new_root_left = erase(root->left, elem);
        root->left = new_root_left;
        return root;
    } else if (root->elem < elem) {
        Node *new_root_right = erase(root->right, elem);
        root->right = new_root_right;
        return root;
    } else {
        return remove_root(root);
    }
}
```



# Fase 1: búsqueda del nodo

```
Node * erase(Node *root, const T &elem) {  
    if (root == nullptr) {  
        return root;  
    } else if (elem < root->elem) {  
  
    } else if (root->elem < elem) {  
  
    } else {  
    }  
}
```

Si llegamos al  
árbol vacío, no  
 hemos encontrado  
 el nodo a borrar.



# Fase 1: búsqueda del nodo

```
Node * erase(Node *root, const T &elem) {  
    if (root == nullptr) {  
  
    } else if (elem < root->elem) {  
        Node *new_root_left = erase(root->left, elem);  
        root->left = new_root_left;  
        return root;  
    } else if (root->elem < elem) {  
  
    } else {  
    }  
}
```

Borramos en el  
hijo izquierdo

# Fase 1: búsqueda del nodo

```
Node * erase(Node *root, const T &elem) {  
    if (root == nullptr) {  
  
    } else if (elem < root->elem) {  
  
    } else if (root->elem < elem) {  
        Node *new_root_right = erase(root->right, elem); ←  
        root->right = new_root_right;  
        return root;  
    } else {  
  
    }  
}
```

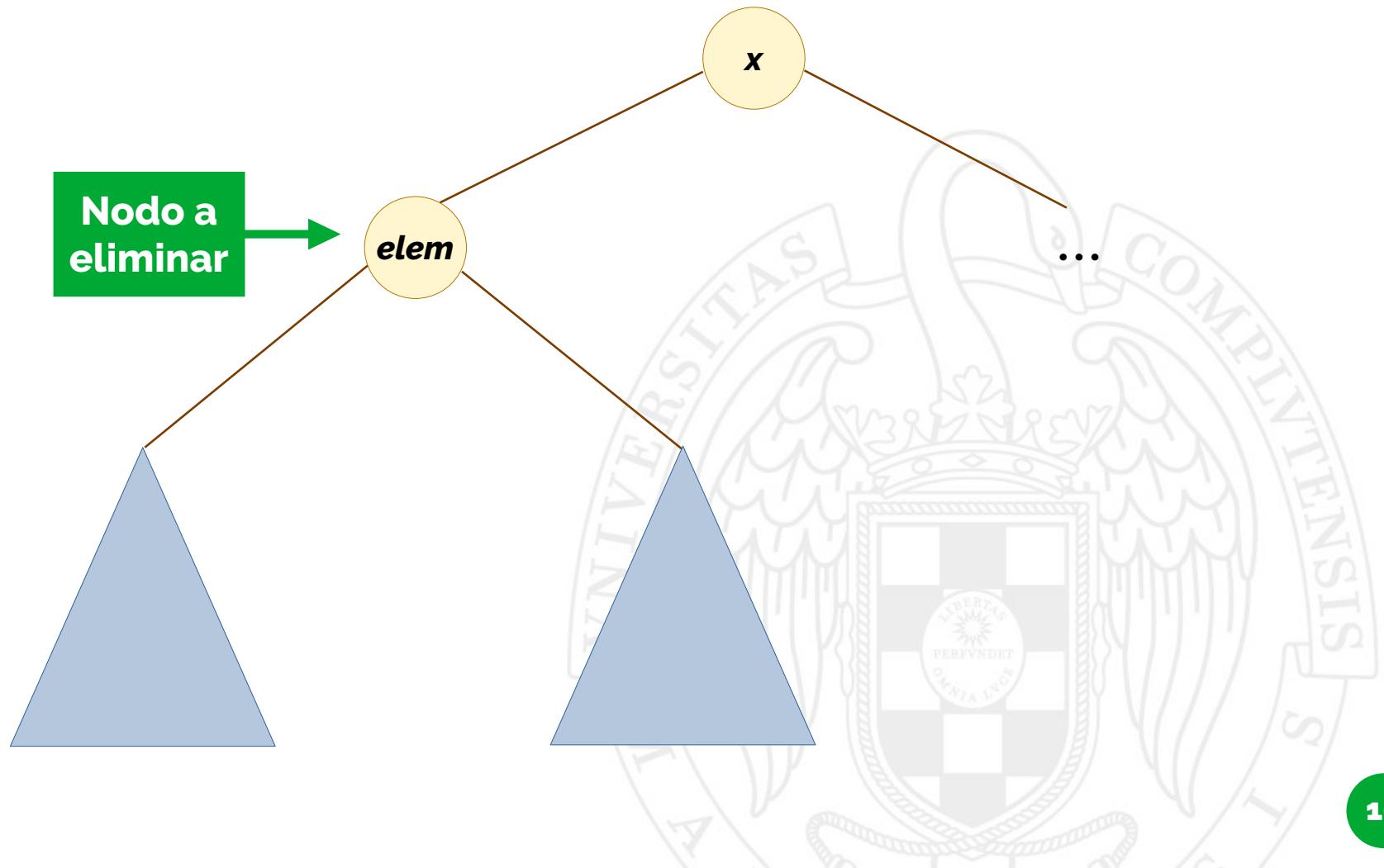
Borramos en el  
hijo derecho

# Fase 1: búsqueda del nodo

```
Node * erase(Node *root, const T &elem) {  
    if (root == nullptr) {  
  
    } else if (elem < root->elem) {  
  
    } else if (root->elem < elem) {  
  
    } else {  
        return remove_root(root);  
    }  
}
```

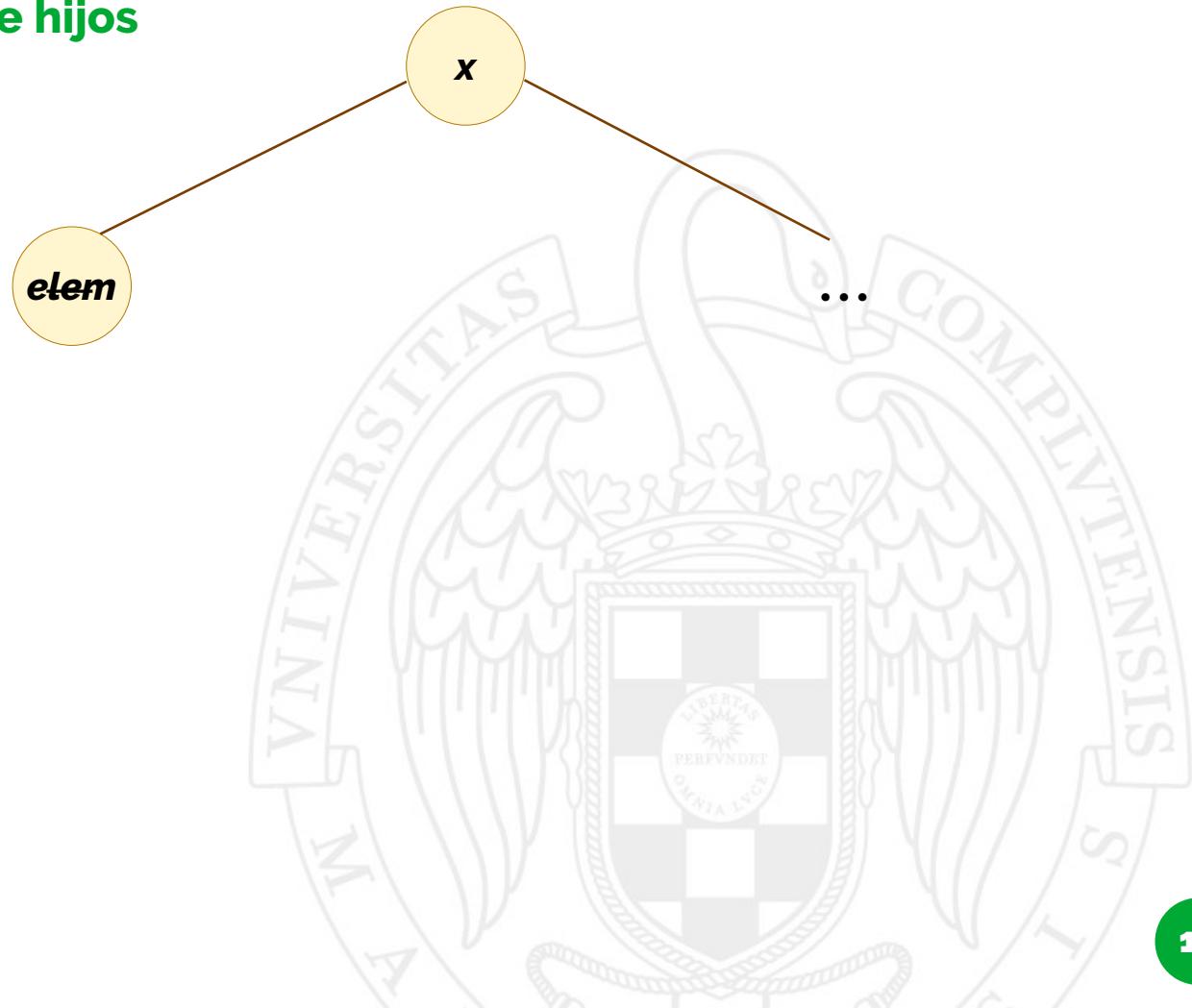
Caso  $\text{root} \rightarrow \text{elem} = \text{elem}$   
Pasamos a fase 2

## Fase 2: eliminación del nodo



# Fase 2: eliminación del nodo

Caso 1: El nodo a eliminar no tiene hijos



# Fase 2: eliminación del nodo

## Caso 1: El nodo a eliminar no tiene hijos

```
Node * remove_root(Node *root) {
    Node *left_child = root->left, *right_child = root->right;
    delete root;
    if (left_child == nullptr && right_child == nullptr) {
        return nullptr;
    } else if (left_child == nullptr) {

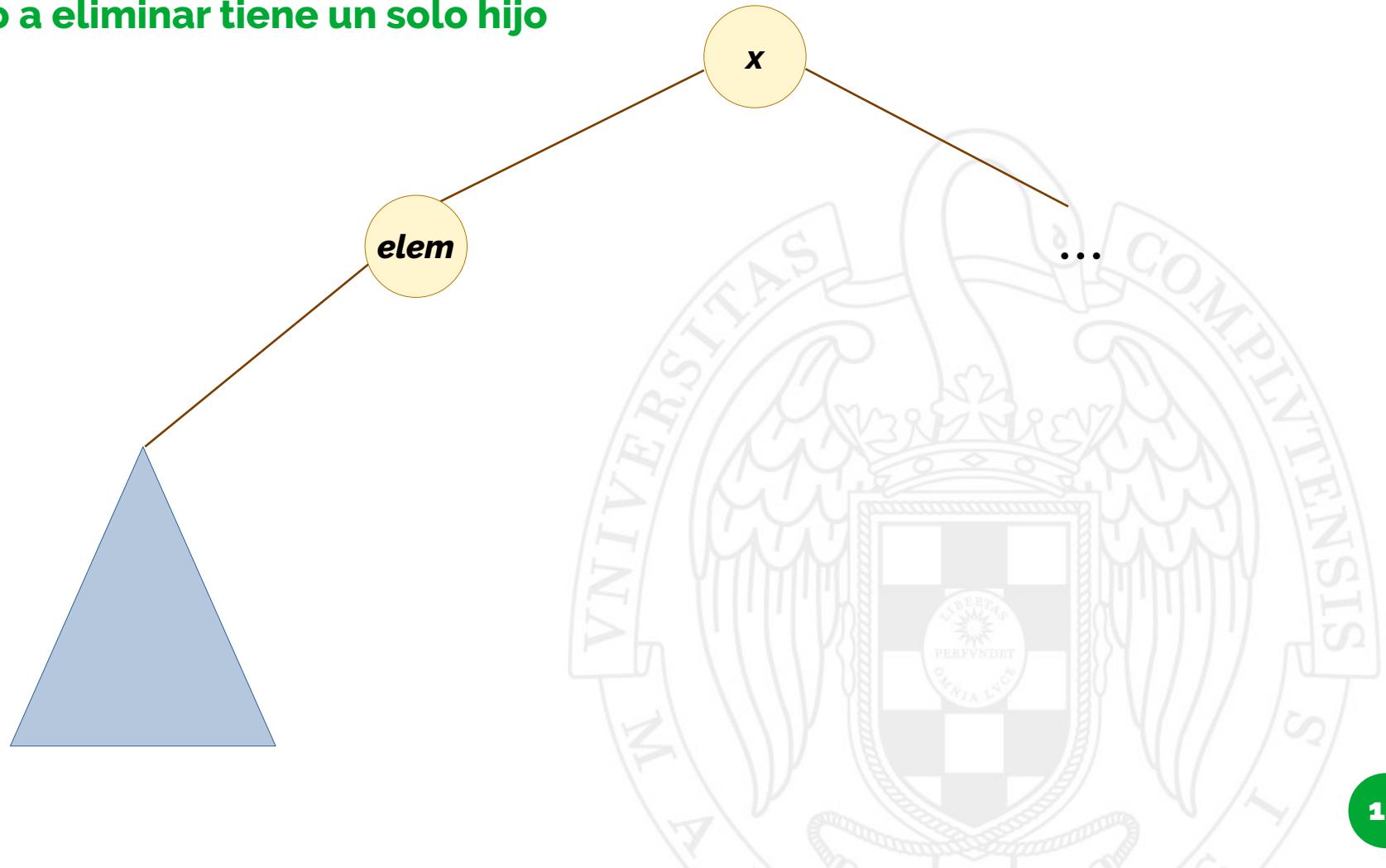
    } else if (right_child == nullptr) {

    } else {
    }
}
```



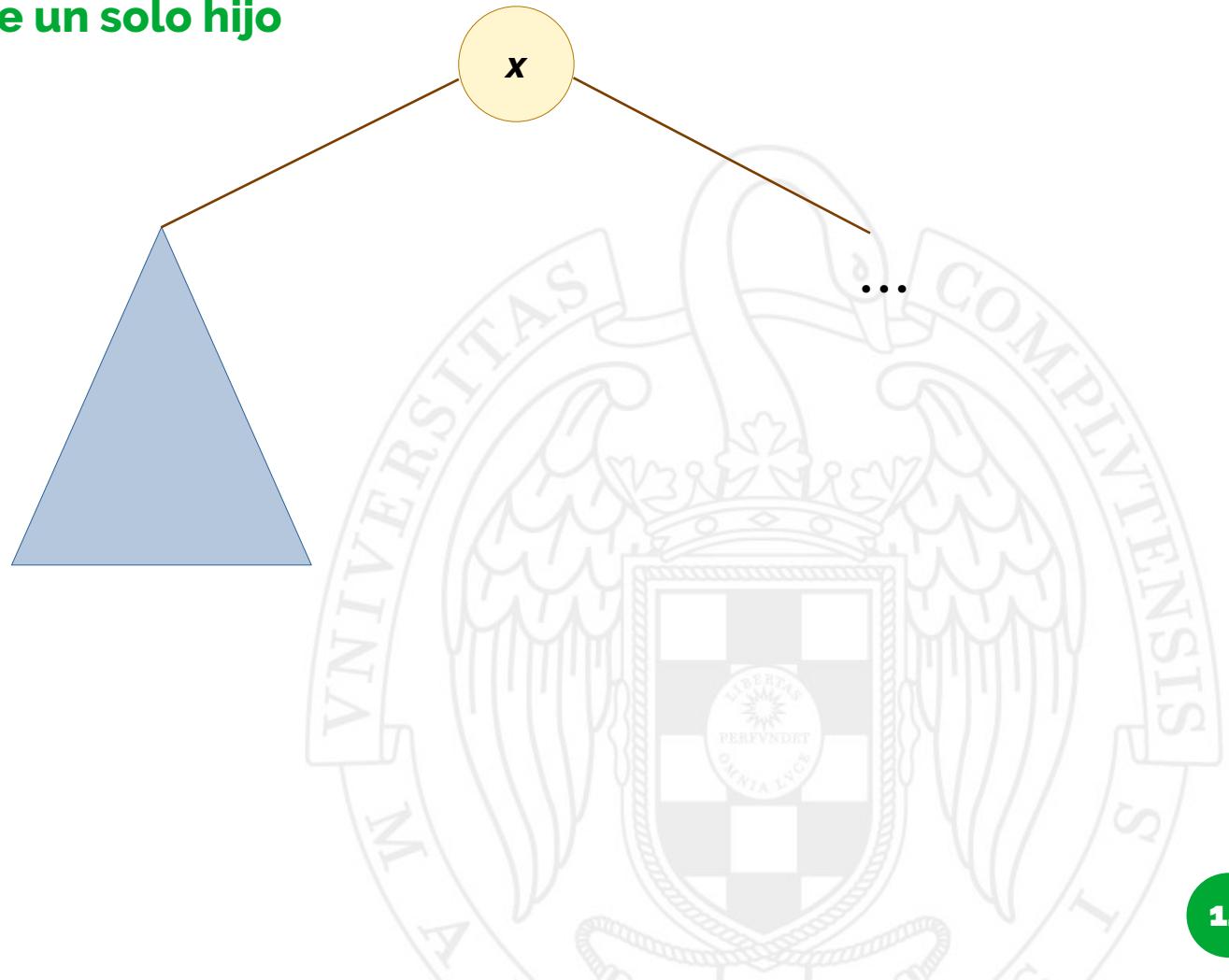
# Fase 2: eliminación del nodo

Caso 2: El nodo a eliminar tiene un solo hijo



# Fase 2: eliminación del nodo

Caso 2: El nodo a eliminar tiene un solo hijo



# Fase 2: eliminación del nodo

## Caso 2: El nodo a eliminar tiene un solo hijo

```
Node * remove_root(Node *root, Node *&new_root) {
    Node *left_child = root->left, *right_child = root->right;
    delete root;
    if (left_child == nullptr && right_child == nullptr) {

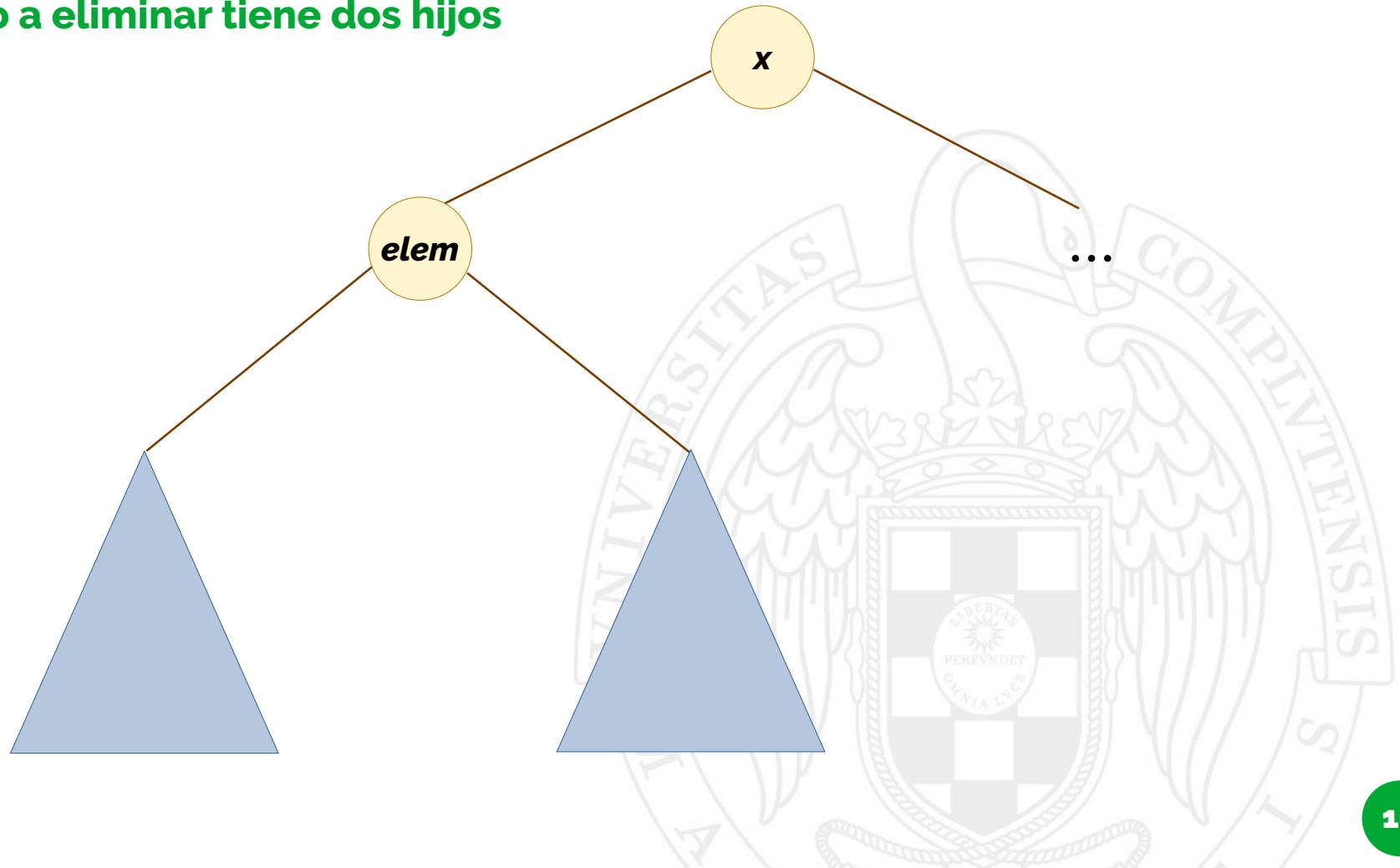
    } else if (left_child == nullptr) {
        return right_child;
    } else if (right_child == nullptr) {
        return left_child;
    } else {

    }
}
```



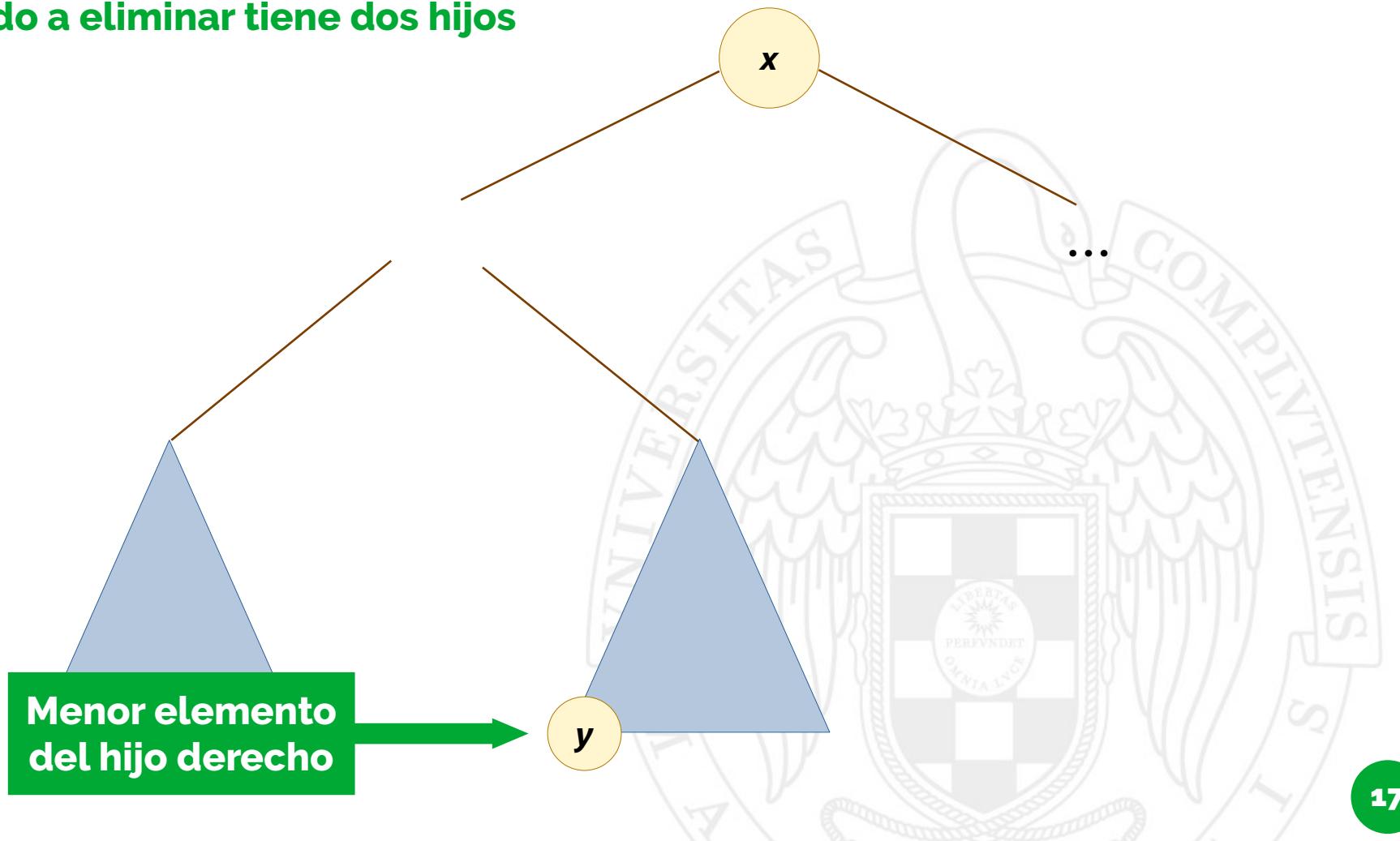
# Fase 2: eliminación del nodo

Caso 3: El nodo a eliminar tiene dos hijos



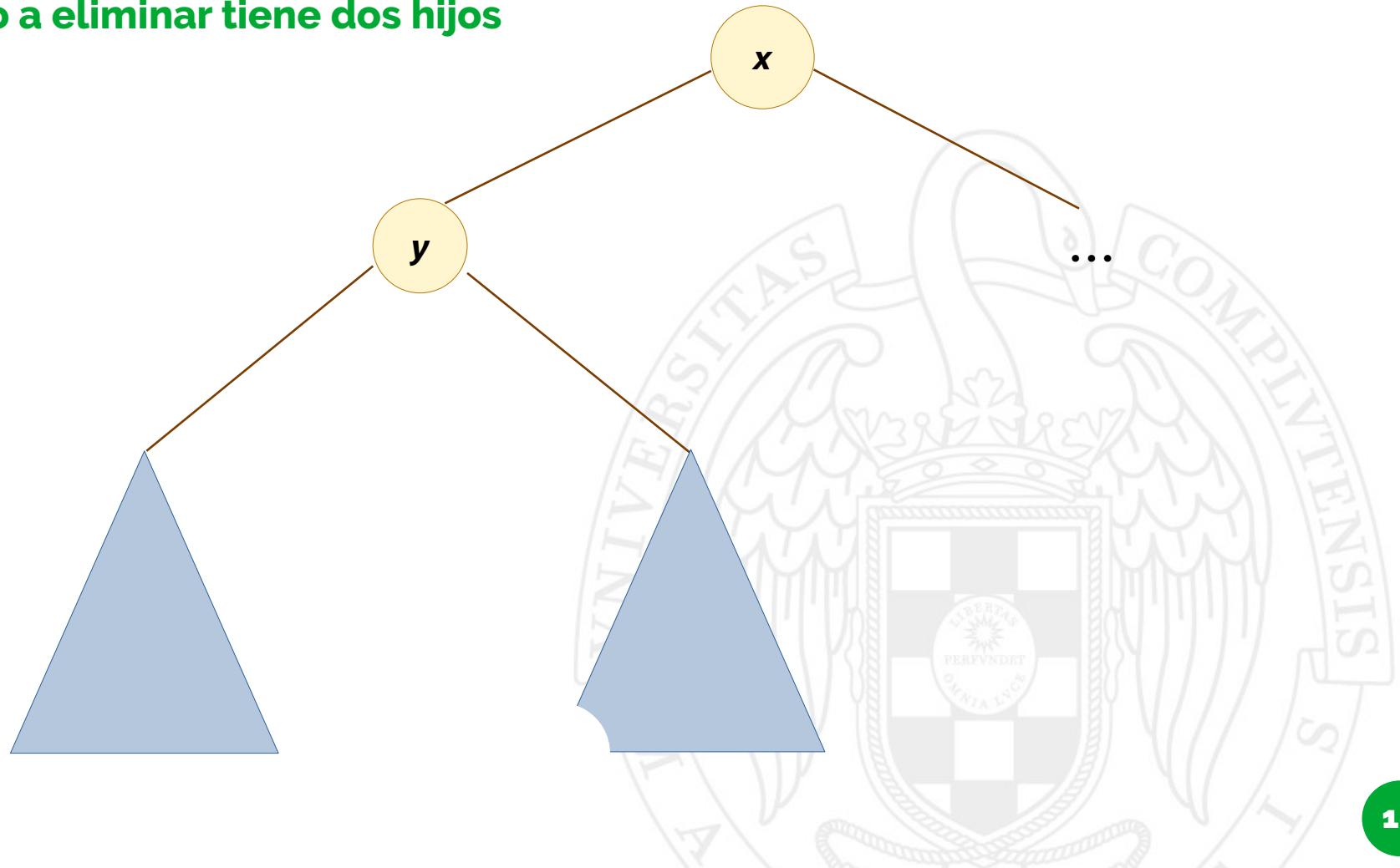
# Fase 2: eliminación del nodo

Caso 3: El nodo a eliminar tiene dos hijos



# Fase 2: eliminación del nodo

Caso 3: El nodo a eliminar tiene dos hijos



# Fase 2: eliminación del nodo

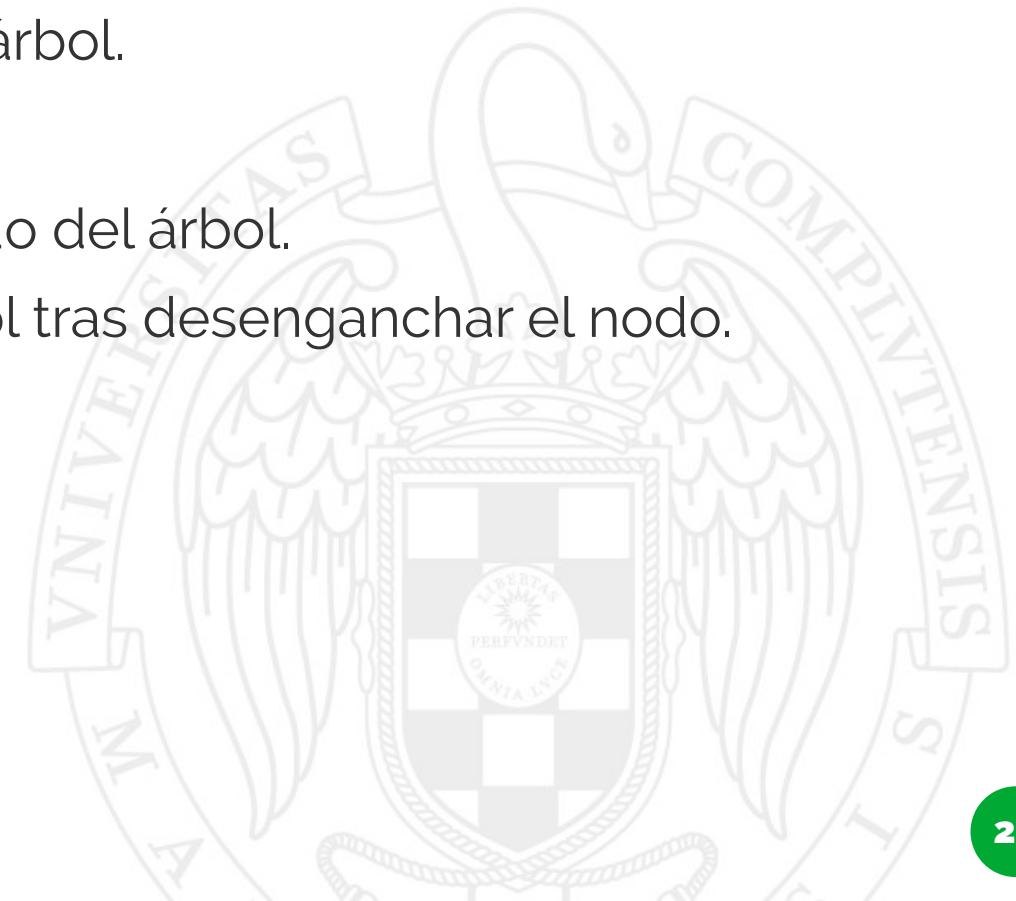
## Caso 3: El nodo a eliminar tiene dos hijos

```
Node * remove_root(Node *root, Node *&new_root) {
    Node *left_child = root->left, *right_child = root->right;
    delete root;
    if (left_child == nullptr && right_child == nullptr) {
    } else if (left_child == nullptr) {
    } else if (right_child == nullptr) {
    } else {
        auto [lowest, new_right_root] = remove_lowest(right_child);
        lowest->left = left_child;
        lowest->right = new_right_root;
        return lowest;
    }
}
```

# Eliminar el nodo más pequeño de un árbol

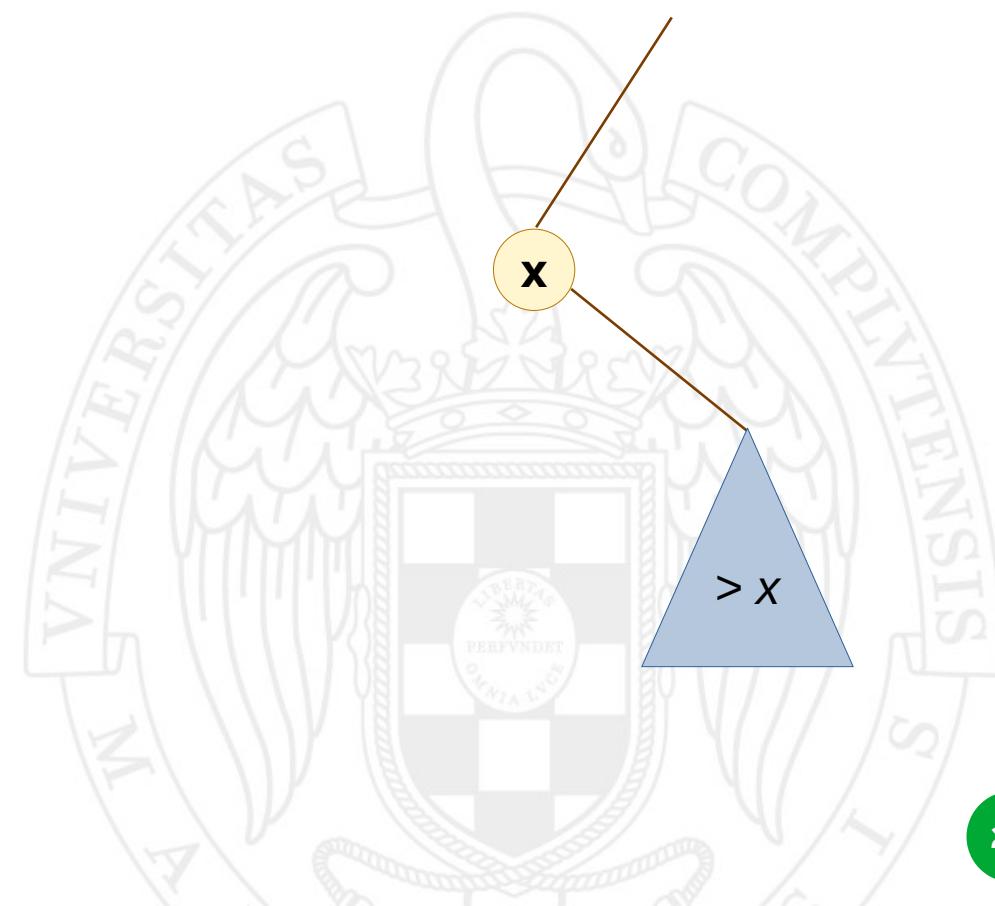
```
std::pair<Node *, Node *> remove_lowest(Node *root)
```

- Dado un árbol cuya raíz es `root`, devuelve el nodo con el valor más pequeño y lo «desengancha» del árbol.
- Devuelve dos punteros:
  - Puntero al nodo desenganchado del árbol.
  - Puntero a la nueva raíz del árbol tras desenganchar el nodo.



# Eliminar el nodo más pequeño de un árbol

```
std::pair<Node *, Node *> remove_lowest(Node *root) {
    assert (root != nullptr);
    if (root->left == nullptr) {
        return {root, root->right};
    } else {
    }
}
```



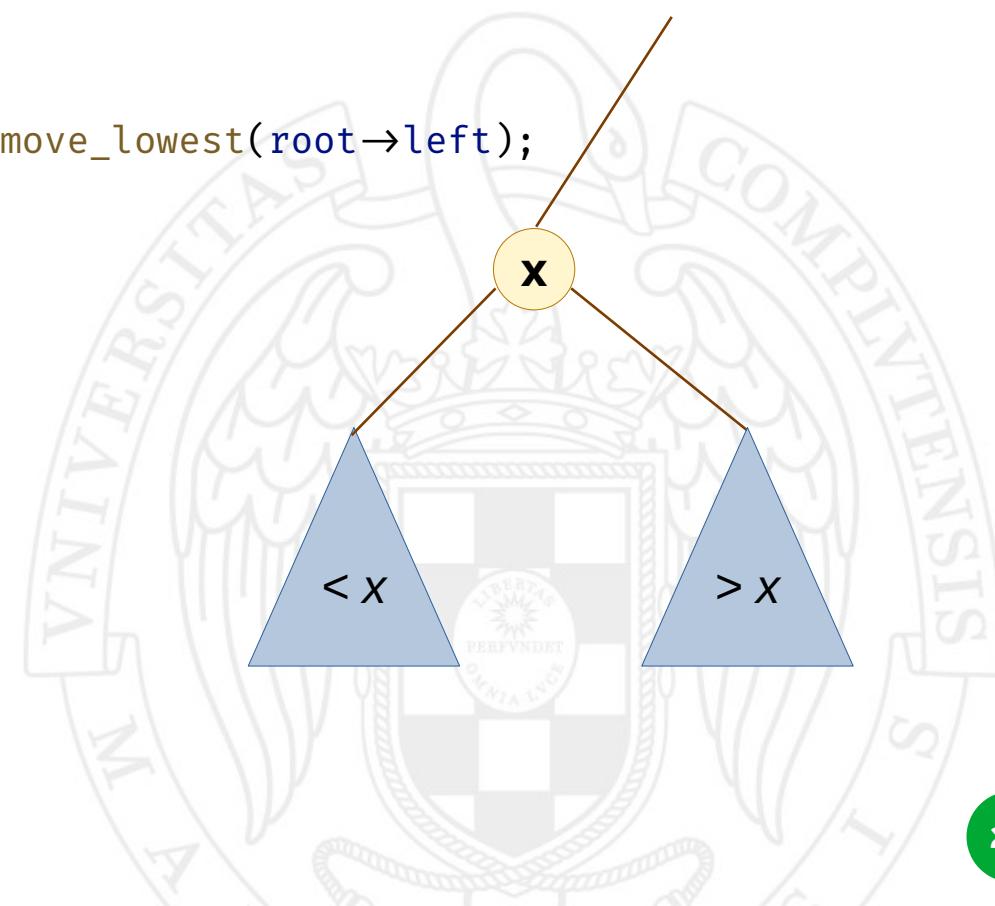
# Eliminar el nodo más pequeño de un árbol

```
std::pair<Node *, Node *> remove_lowest(Node *root) {
    assert (root != nullptr);
    if (root->left == nullptr) {
        return {root, root->right};
    } else {
    }
}
```



# Eliminar el nodo más pequeño de un árbol

```
std::pair<Node *, Node *> remove_lowest(Node *root) {
    assert (root != nullptr);
    if (root->left == nullptr) {
        return {root, root->right};
    } else {
        auto [removed_node, new_root_left] = remove_lowest(root->left);
        root->left = new_root_left;
        return {removed_node, root};
    }
}
```



# Recapitulando

- `erase(root, elem)`

Busca el elemento que se quiere eliminar. Cuando se encuentra, llama a `remove_root` sobre el nodo que contiene el elemento.

- `remove_root(root)`

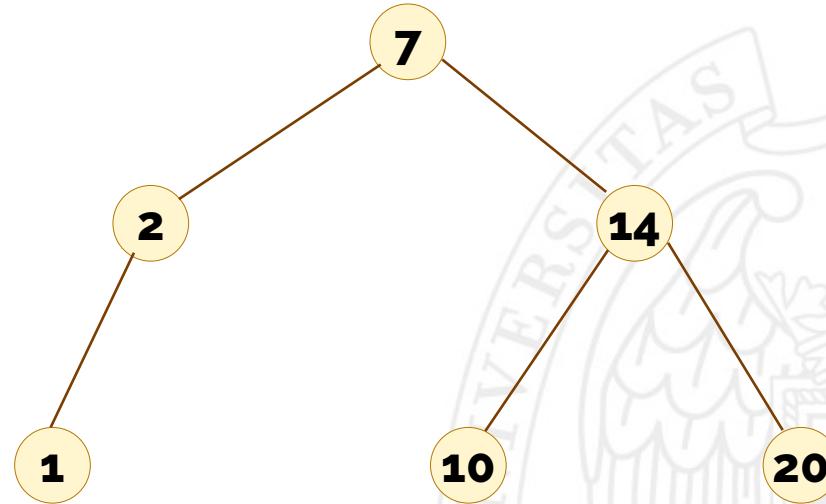
Elimina la raíz del árbol, devolviendo la nueva raíz. Si la raíz tiene dos hijos, la nueva raíz es el nodo con el menor valor del hijo derecho. Se llama a `remove_lowest` para obtener este último nodo.

- `remove_lowest(root)`

Devuelve el nodo que contiene el valor más pequeño del árbol cuya raíz es `root` y lo desengancha del árbol.

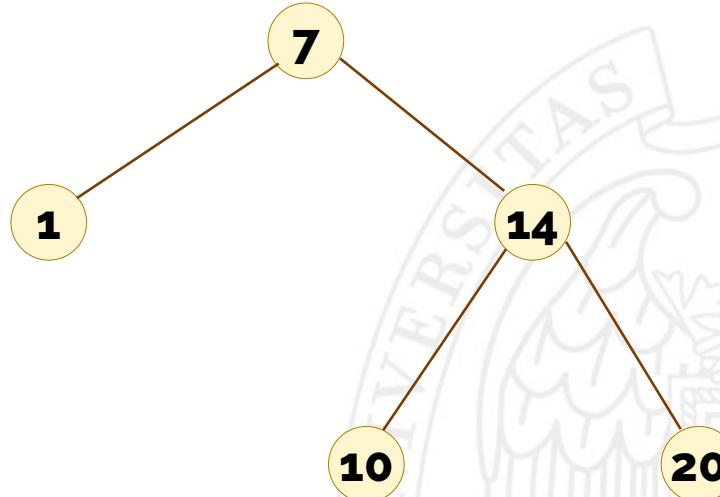
# Ejemplo

- Eliminar el valor **2**



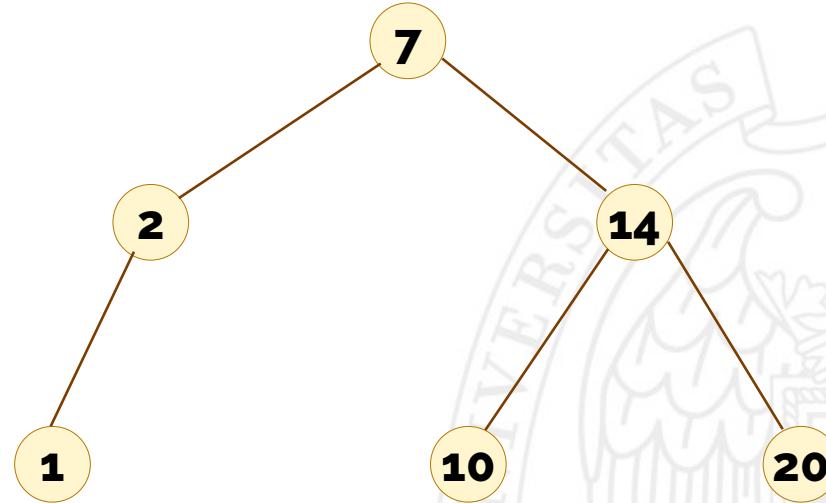
# Ejemplo

- Eliminar el valor **2**



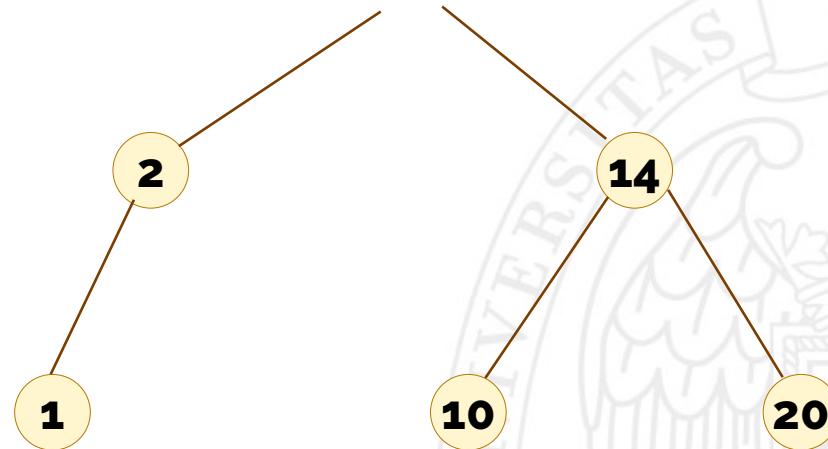
# Ejemplo

- Eliminar el valor 7



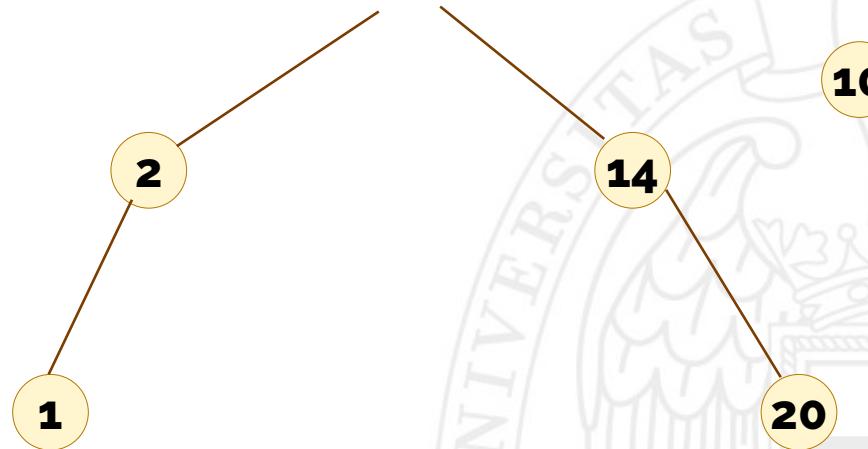
# Ejemplo

- Eliminar el valor 7



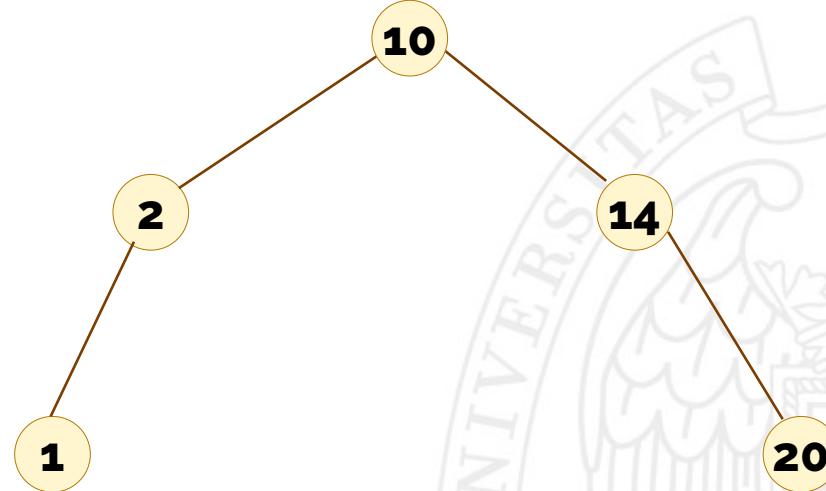
# Ejemplo

- Eliminar el valor 7



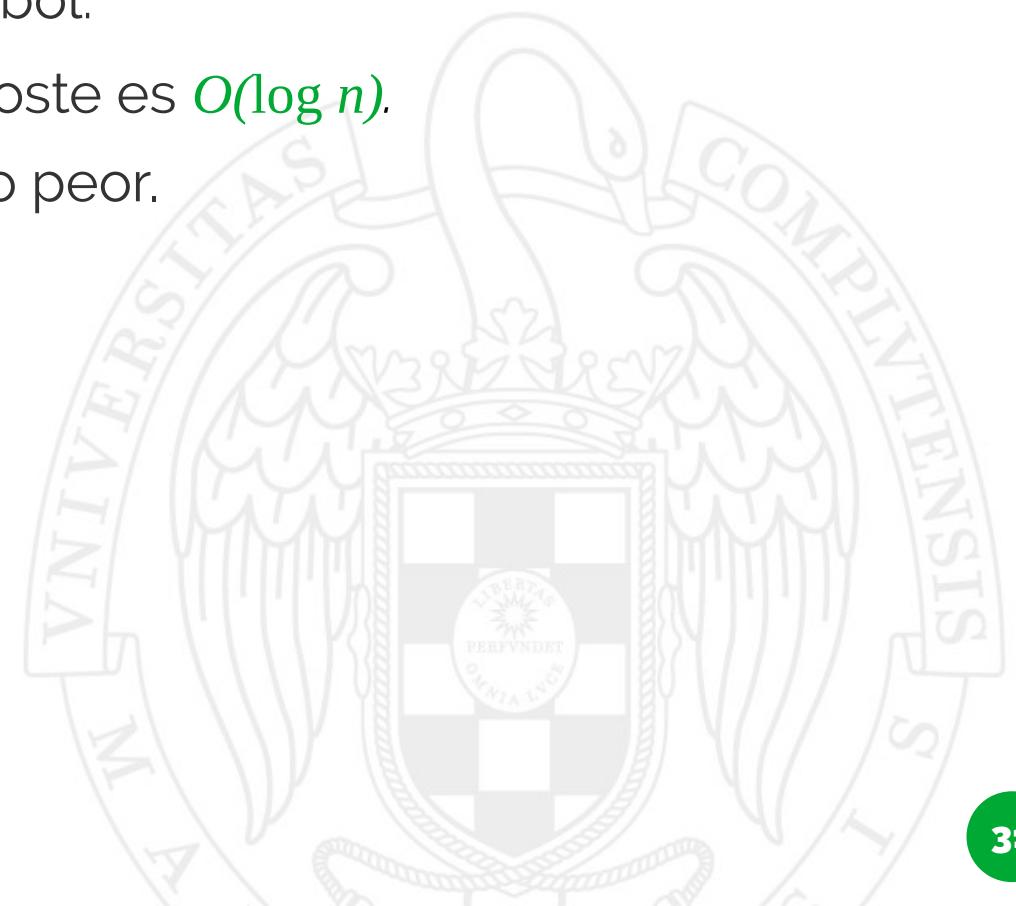
# Ejemplo

- Eliminar el valor 7



# Coste en tiempo

- Si  $h$  es la altura del árbol, el coste es  $O(h)$ .
- Y si  $n$  es el número de nodos del árbol:
  - Si el árbol está equilibrado, el coste es  $O(\log n)$ .
  - Si no, el coste es  $O(n)$  en el caso peor.



ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

# Implementación del TAD Conjunto mediante ABBs

Manuel Montenegro Montes

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

# Operaciones en el TAD Conjunto

- Constructoras:
  - Crear un conjunto vacío: ***create\_empty***
- Mutadoras:
  - Añadir un elemento al conjunto: ***insert***
  - Eliminar un elemento del conjunto: ***erase***
- Observadoras:
  - Averiguar si un elemento está en el conjunto: ***contains***
  - Saber si el conjunto está vacío: ***empty***
  - Saber el tamaño del conjunto: ***size***

# Dos implementaciones

- Mediante **listas**.
- Mediante **árboles binarios de búsqueda**.



Este vídeo



# Interfaz de SetTree

```
template <typename T>
class SetTree {
public:
    SetTree();
    SetTree(const SetTree &other);
    ~SetTree();

    void insert(const T &elem);
    void erase(const T &elem);

    bool contains(const T &elem) const;
    int size() const;
    bool empty() const;

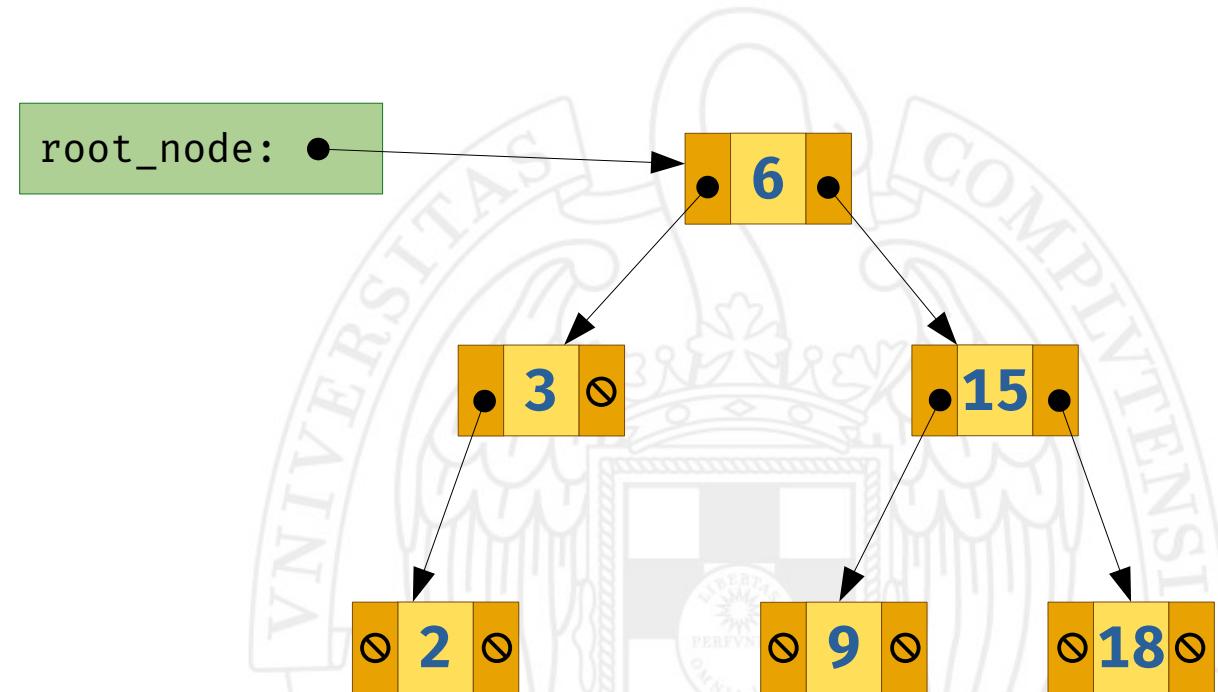
private:
    ...
};
```



# Implementación de SetTree

```
template <typename T>
class SetTree {
public:
    ...
private:
    struct Node {
        T elem;
        Node *left, *right;
    };
    Node *root_node;
};
```

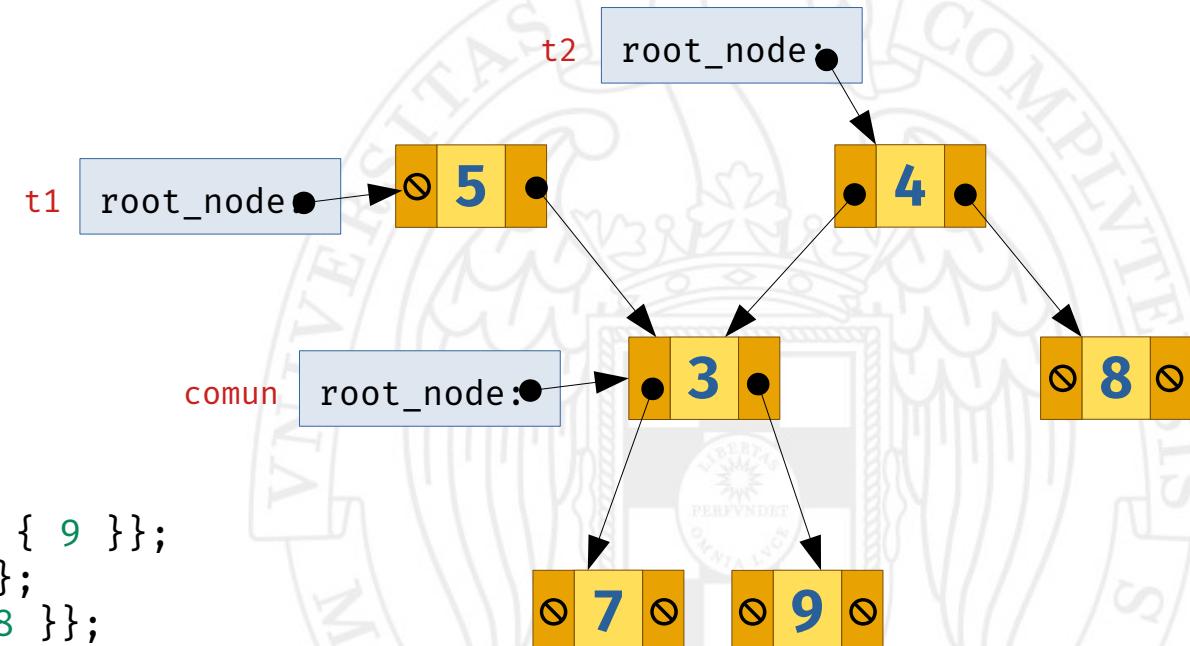
{2, 3, 6, 9, 15, 18}



# Sobre la competición

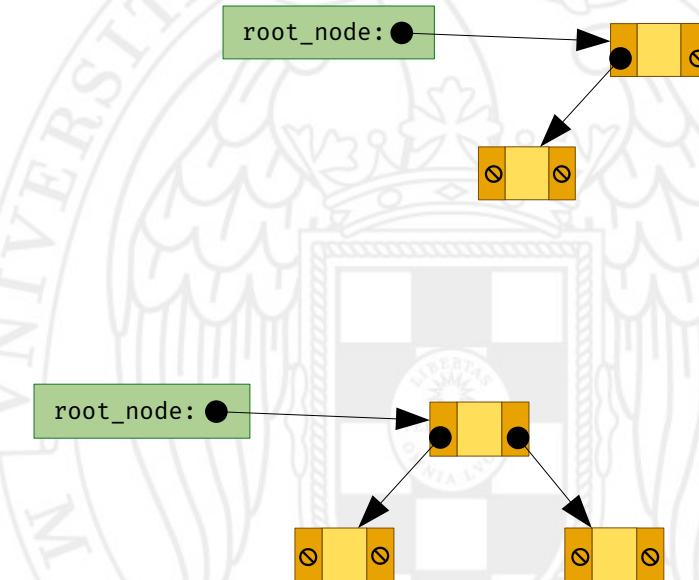
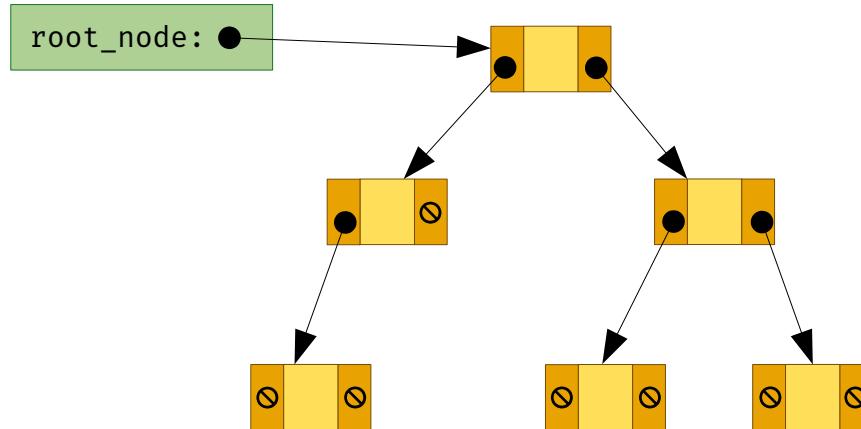
# Anteriormente...

- Implementamos un TAD para árboles binarios.
- Utilizábamos *smart pointers* para enlazar los nodos, porque árboles binarios distintos podían compartir nodos:



# Pero aquí...

- Implementamos un TAD para conjuntos.
- Cada objeto de la clase SetTree apunta a la raíz de su propio árbol de nodos.
- **No hay compartición** entre los nodos de dos SetTree distintos.



# Pero aquí...

- Implementamos un TAD para conjuntos.
- Cada objeto de la clase SetTree apunta a la raíz de su propio árbol de nodos.
- **No hay compartición** entre los nodos de dos SetTree distintos.
- Consecuencias:
  - No necesitamos punteros inteligentes.
  - Cada SetTree es responsable de liberar sus nodos.
  - El constructor de copia de SetTree debe copiar los nodos del conjunto origen al conjunto destino.

# Constructores y destructor de SetTree

```
template <typename T>
class SetTree {
public:
    SetTree(): root_node(nullptr) { }

    SetTree(const SetTree &other): root_node(copy_nodes(other.root_node)) { }

    ~SetTree() {
        delete_nodes(root_node);
    }

private:
    ...
    Node *root_node;
};
```

# Operaciones consultoras y mutadoras

# Métodos auxiliares

```
template <typename T>
class SetTree {
public:
private:
    Node *root_node;
    ...
    static Node * insert(Node *root, const T &elem);
    static bool search(const Node *root, const T &elem);
    static Node * erase(Node *root, const T &elem);
    ...
};
```

# Métodos de la interfaz

```
template <typename T>
class SetTree {
public:

    void insert(const T &elem) { root_node = insert(root_node, elem); }
    void erase(const T &elem) { root_node = erase(root_node, elem); }
    bool contains(const T &elem) const { return search(root_node, elem); }
    bool empty() const { return root_node == nullptr; }
    int size() const { return num_nodes(root_node); }

private:
    Node *root_node;
    ...
    static Node * insert(Node *root, const T &elem);
    static bool search(const Node *root, const T &elem);
    static Node * erase(Node *root, const T &elem);
    ...
};

};
```

# Coste de las operaciones

Operación	Árbol equilibrado	Árbol no equilibrado
<i>constructor</i>	$O(1)$	$O(1)$
<i>empty</i>	$O(1)$	$O(1)$
<i>size</i>	$O(n)$	$O(n)$
<i>contains</i>	$O(\log n)$	$O(n)$
<i>insert</i>	$O(\log n)$	$O(n)$
<i>erase</i>	$O(\log n)$	$O(n)$

$n$  = número de elementos del conjunto

# Mejorando la operación size()

# Mejorando el coste de size()

- Contar los nodos de un árbol binario de búsqueda tiene coste lineal con respecto al número de nodos.
- Es posible mejorar ese coste incluyendo un atributo `num_elems` en la clase `SetTree` y actualizándolo cada vez que haya una inserción o eliminación.

# Mejorando el coste de size()

```
template <typename T>
class SetTree {
public:

    int size() const { return num_elems; }

    void insert(const T &elem) {
        root_node = insert(root_node, elem);
        num_elems++;
    }

    void erase(const T &elem) {
        root_node = erase(root_node, elem);
        num_elems--;
    }

    ...

private:
    Node *root_node;
    int num_elems;

    ...
};
```

Incorrecto!

Incorrecto!

# ¿Por qué no es correcto insert?

- Porque si el elemento a insertar ya se encuentra en el conjunto, no aumenta el número de elementos del conjunto.
- En este caso, no tenemos que incrementar num\_elems.
- Cambiamos la función auxiliar insert:

```
Node * insert(Node *node, const T &elem)
```

por:

```
pair<Node *, bool> insert(Node *node, const T &elem)
```

- La función devuelve true si el elem se ha insertado realmente, o false si no se ha insertado porque ya existía en el conjunto.

# ¿Por qué no es correcto erase?

- Porque si el elemento a eliminar no se encuentra en el conjunto, la función `erase` no elimina nada.
- En este caso, no tenemos que decrementar `num_elems`.
- Cambiamos la función auxiliar `erase`:

```
Node * erase(Node *node, const T &elem)
```

por:

```
pair<Node *, bool> erase(Node *node, const T &elem)
```

# Cambios en insert

```
static std::pair<Node *, bool> insert(Node *root, const T &elem) {
    if (root == nullptr) {
        return {new Node(nullptr, elem, nullptr), true};
    } else if (elem < root->elem) {
        auto [new_root_left, inserted] = insert(root->left, elem);
        root->left = new_root_left;
        return {root, inserted};
    } else if (root->elem < elem) {
        auto [new_root_right, inserted] = insert(root->right, elem);
        root->right = new_root_right;
        return {root, inserted};
    } else {
        return {root, false};
    }
}
```

# Cambios en la clase SetTree

```
template <typename T>
class SetTree {
public:
    ...
    void insert(const T &elem) {
        auto [new_root, inserted] = insert(root_node, elem);
        root_node = new_root;
        if (inserted) { num_elems++; }
    }

    void erase(const T &elem) {
        auto [new_root, removed] = erase(root_node, elem);
        root_node = new_root;
        if (removed) { num_elems--; }
    }

    ...
private:
    Node *root_node;
    int num_elems;
    ...
};
```

# Coste de las operaciones

Operación	Árbol equilibrado	Árbol no equilibrado
<i>constructor</i>	$O(1)$	$O(1)$
<i>empty</i>	$O(1)$	$O(1)$
<i>size</i>	$O(1)$	$O(1)$
<i>contains</i>	$O(\log n)$	$O(n)$
<i>insert</i>	$O(\log n)$	$O(n)$
<i>erase</i>	$O(\log n)$	$O(n)$

$n$  = número de elementos del conjunto

ESTRUCTURAS DE DATOS

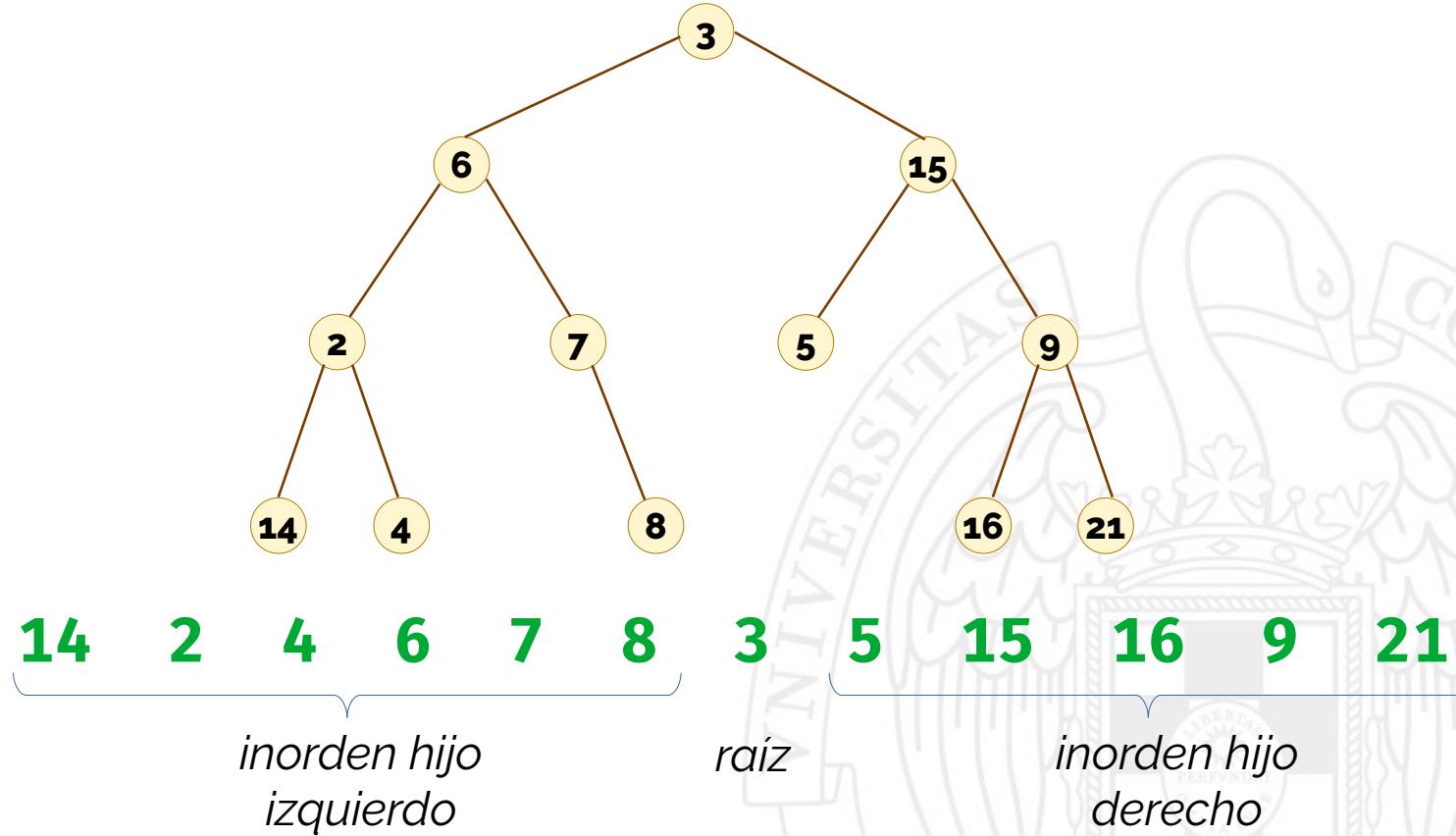
TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

# Recorrido en inorden iterativo (1)

Manuel Montenegro Montes

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

# Recordatorio: recorrido en inorden

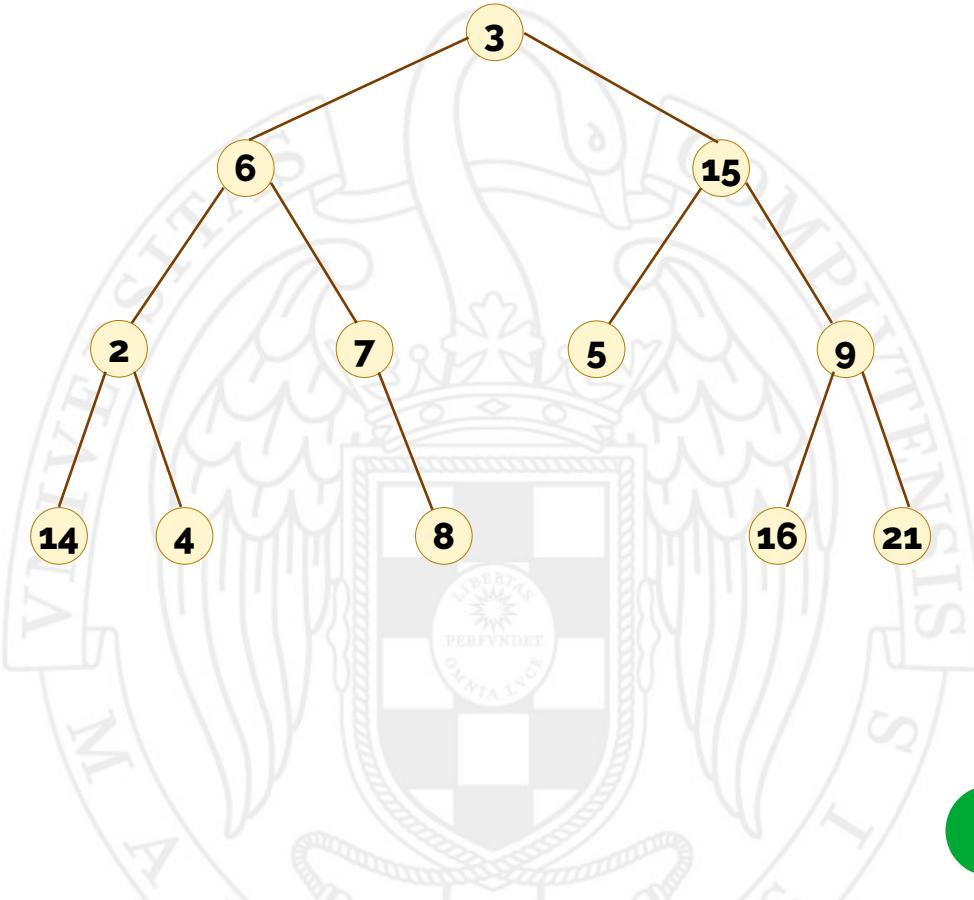


# Observaciones previas

# Consideraciones previas

- ¿Cuál es el primer nodo que se visita?

El que se alcanza tras descender por los hijos izquierdos hasta que no se pueda más.

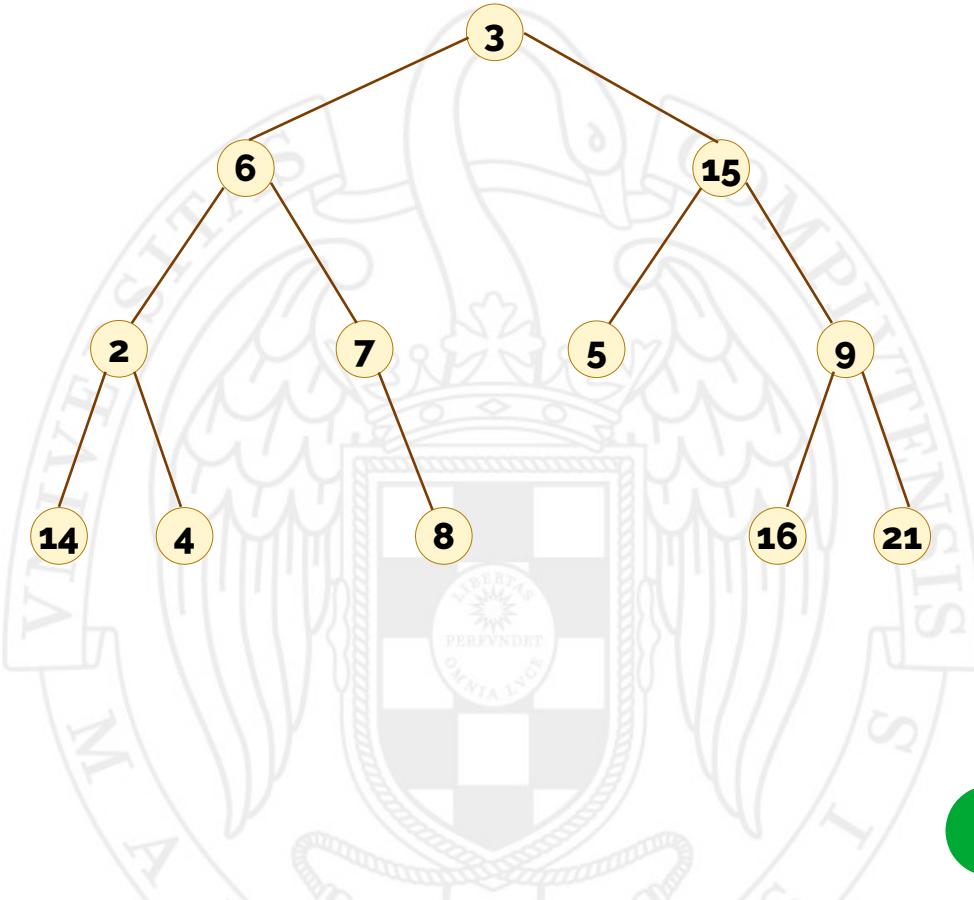


# Consideraciones previas

- Acabo de visitar un nodo. ¿Cuál es el siguiente?

Baja al hijo derecho, y busca allí el primer nodo que habría que visitar:

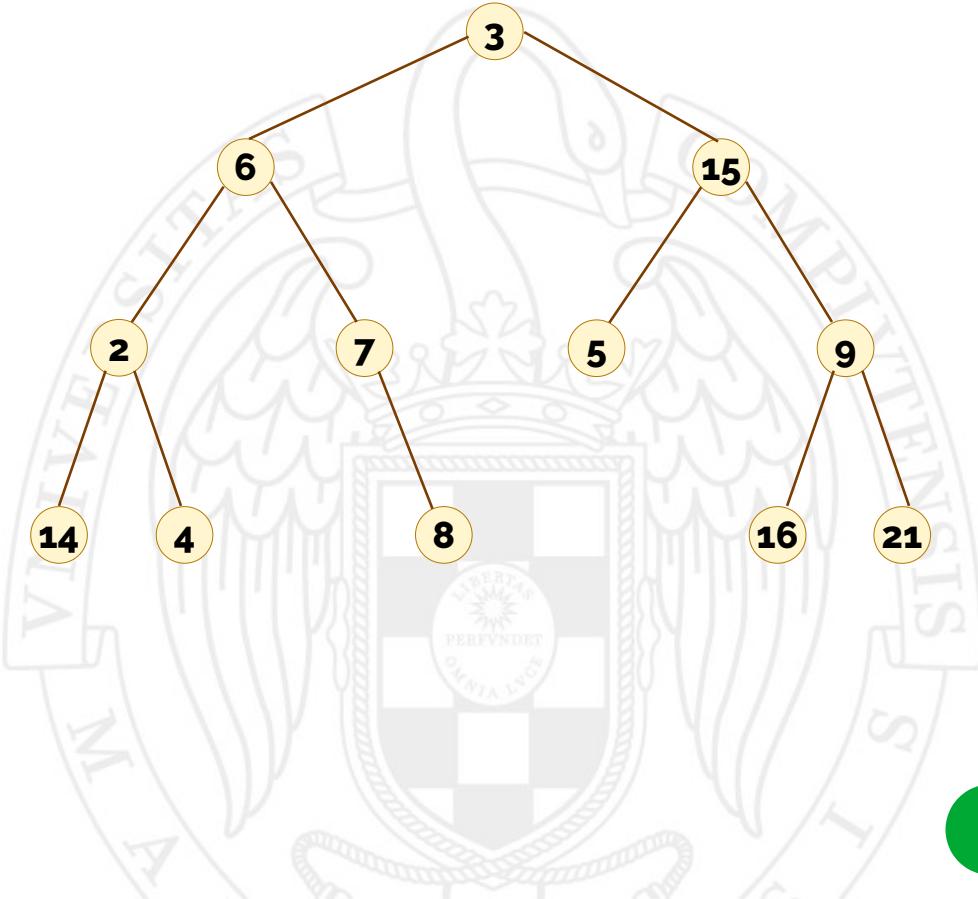
*El que se alcanza tras descender por los hijos izquierdos hasta que no se pueda más.*



# Consideraciones previas

- ¿Y si no hay hijo derecho?

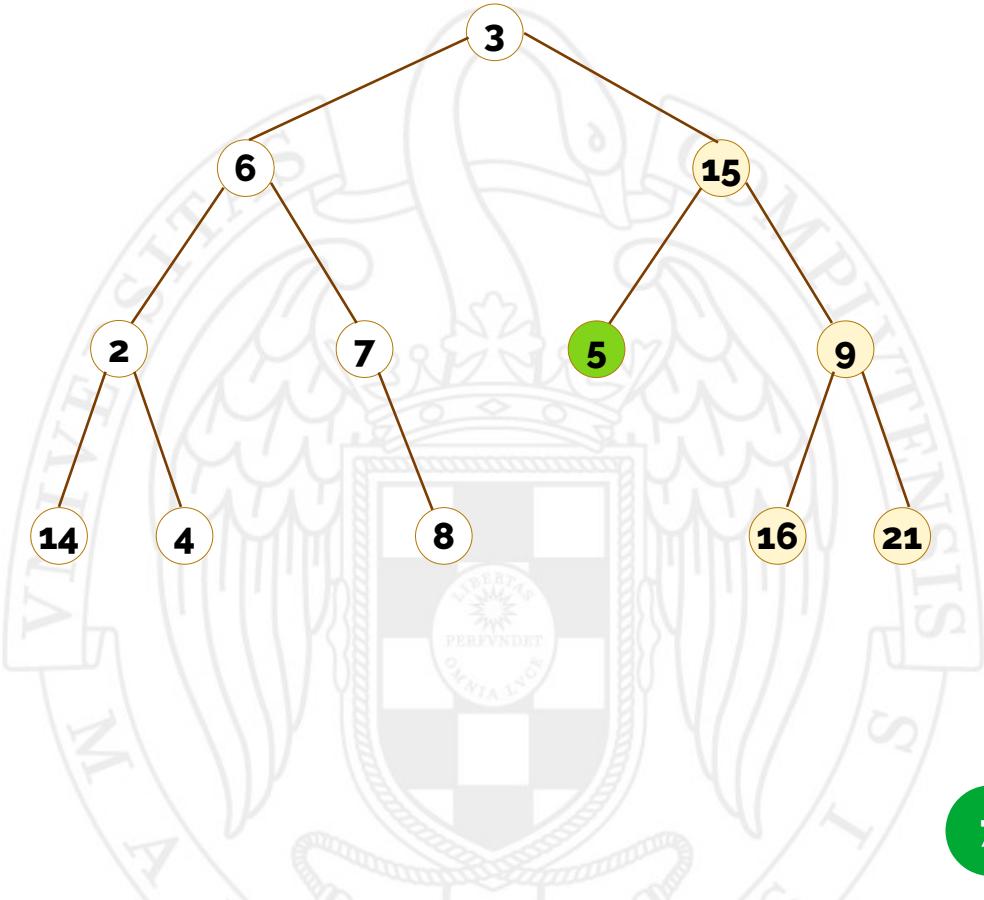
Hay que subir por el árbol hasta el primer antecesor no visitado.



# Consideraciones previas

- ¿Y si no hay hijo derecho?

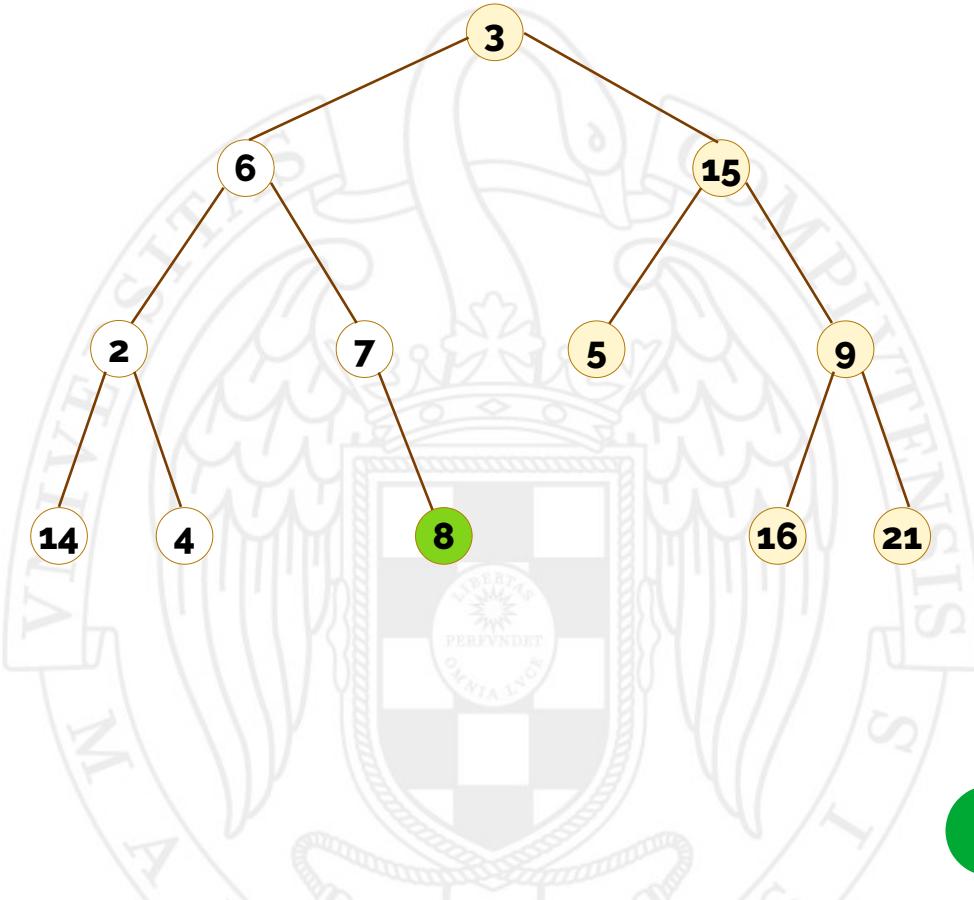
Hay que subir por el árbol hasta el primer antecesor no visitado.



# Consideraciones previas

- ¿Y si no hay hijo derecho?

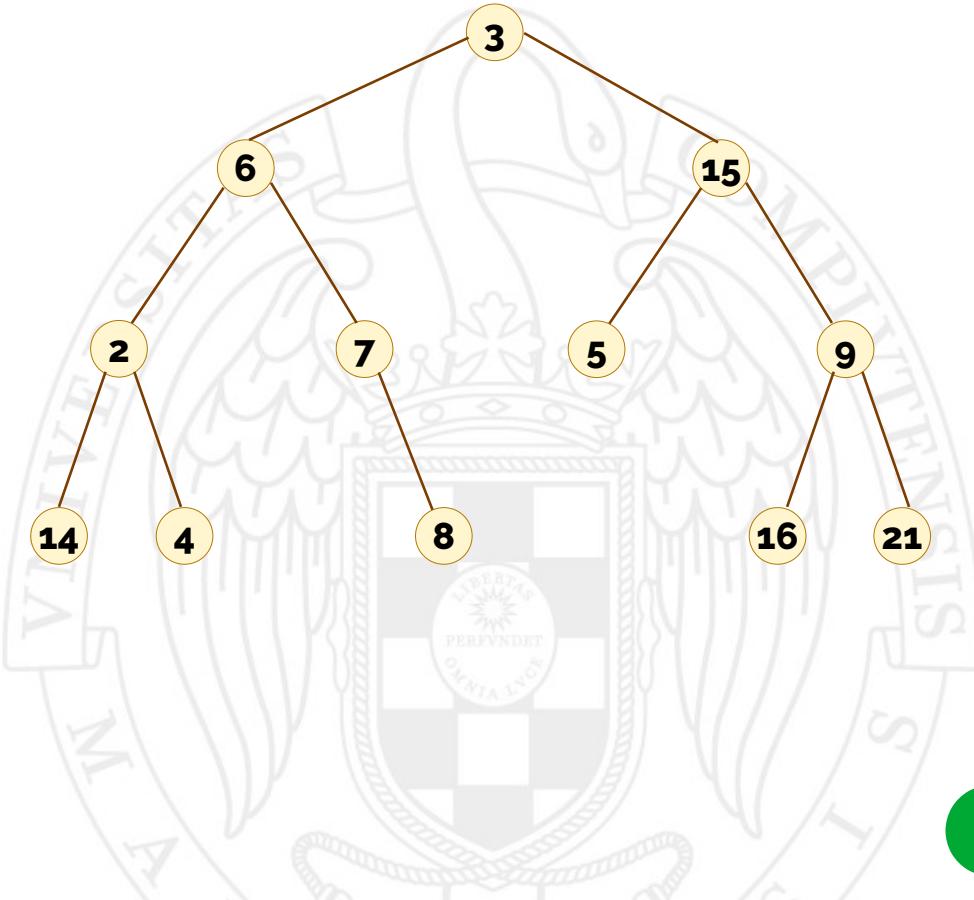
Hay que subir por el árbol hasta el primer antecesor no visitado.



# Consideraciones previas

- *¿Cómo subo hasta el antecesor no visitado? Solamente tengo punteros a los hijos, pero no al nodo padre.*

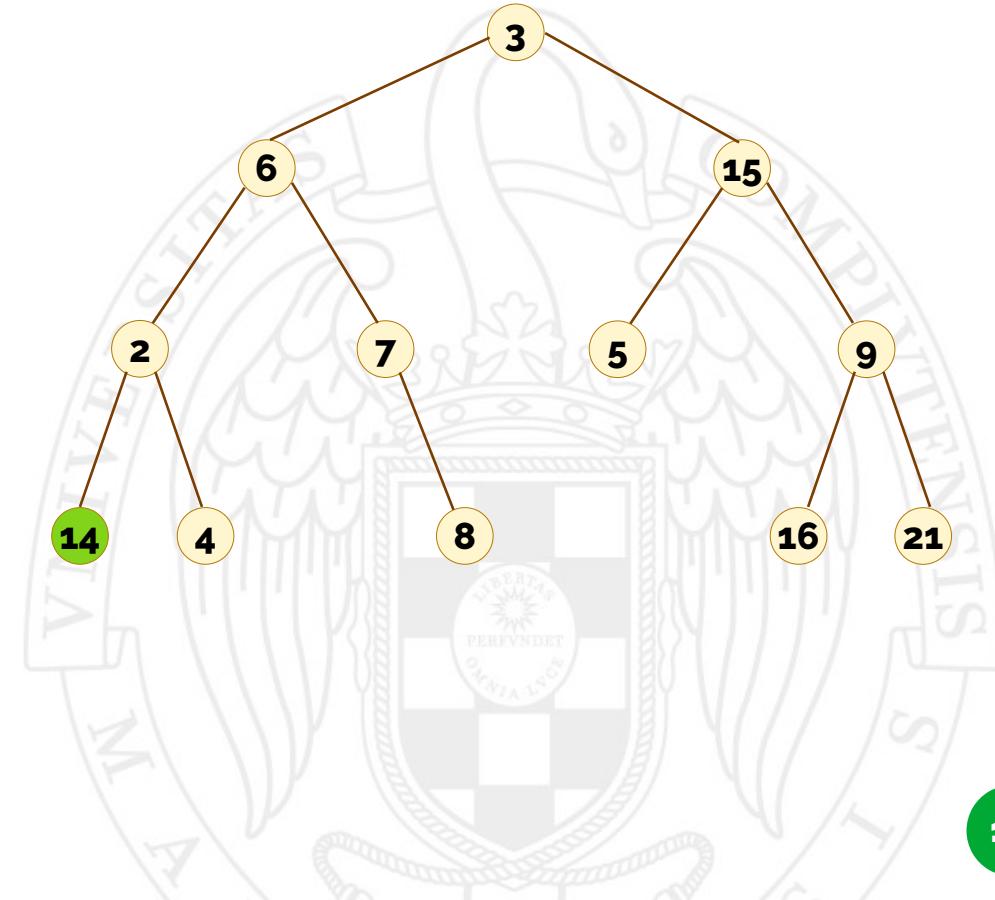
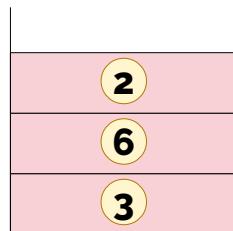
Es necesaria una **pila** que almacene los antecesores no visitados hasta uno dado.



# Consideraciones previas

- ¿Cómo subo hasta el antecesor no visitado? Solamente tengo punteros a los hijos, pero no al nodo padre.

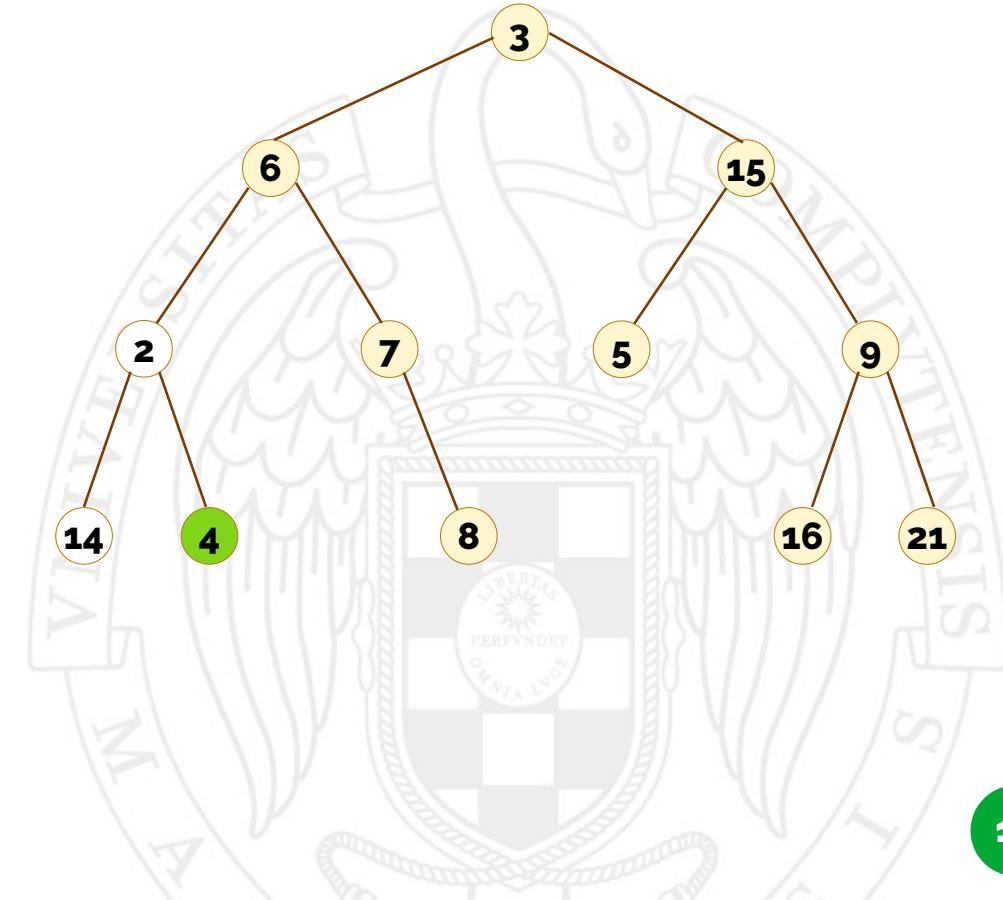
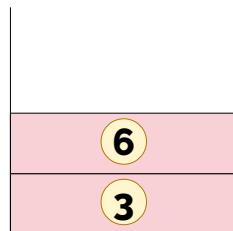
Es necesaria una **pila** que almacene los antecesores no visitados hasta uno dado.



# Consideraciones previas

- ¿Cómo subo hasta el antecesor no visitado? Solamente tengo punteros a los hijos, pero no al nodo padre.

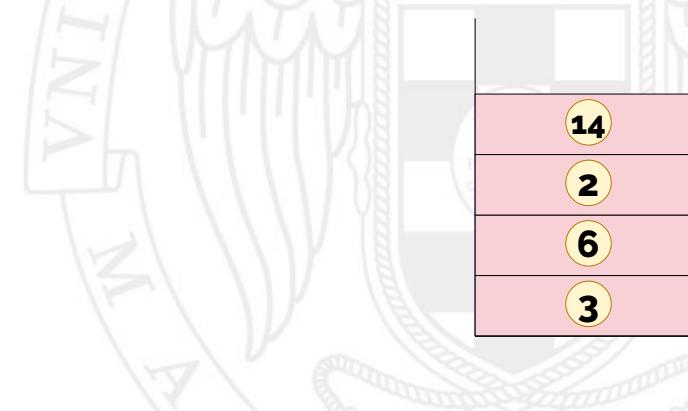
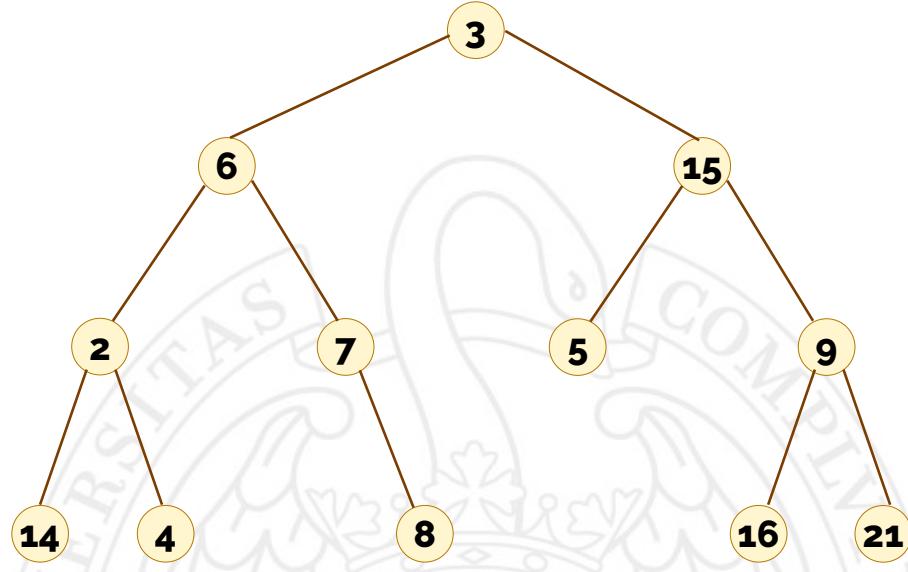
Es necesaria una **pila** que almacene los antecesores no visitados hasta uno dado.



# Algoritmo de recorrido

# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

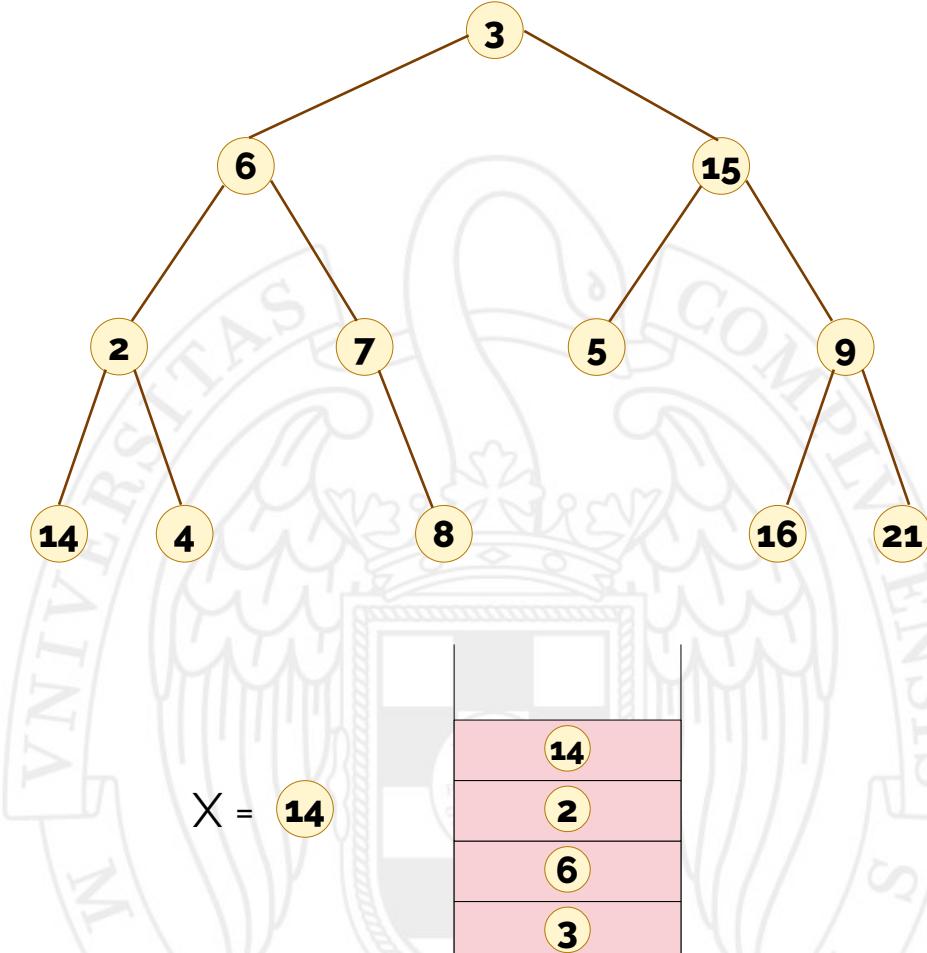


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.

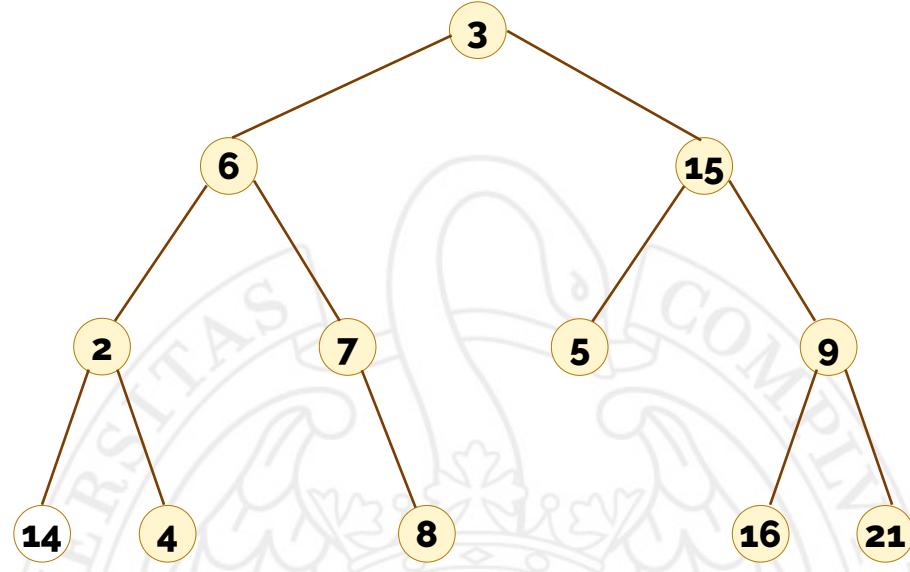


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.



$$X = 14$$

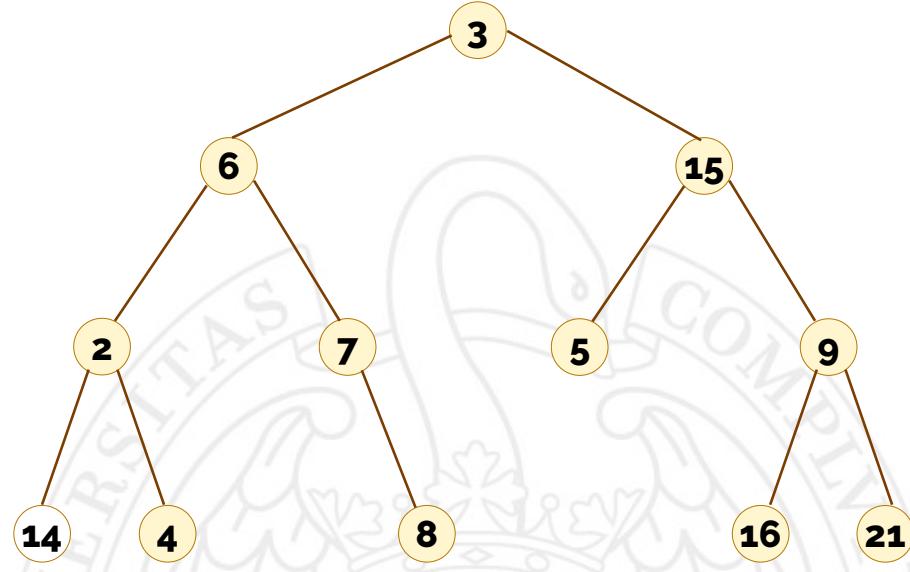
2
6
3

# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - ...
  - ...



$$X = 14$$

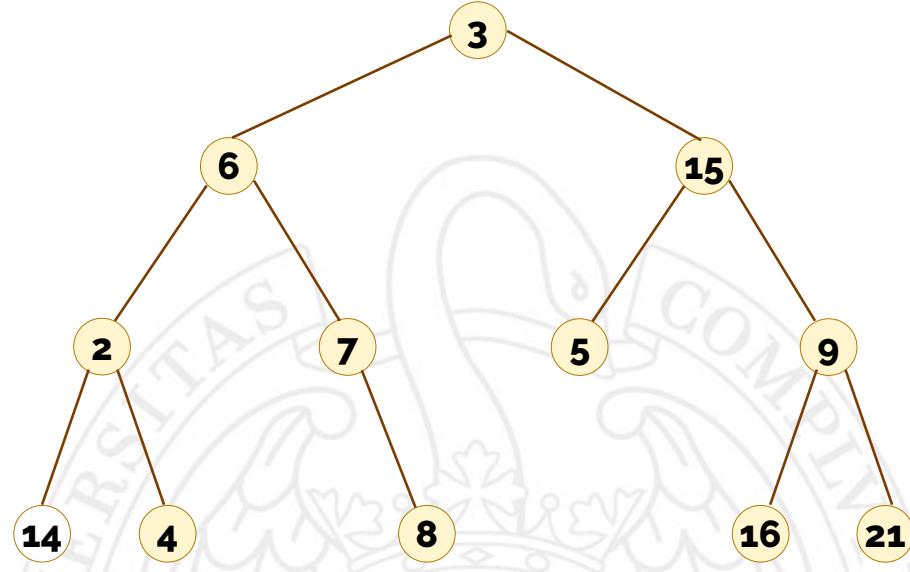
2
6
3

# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - ...
  - ...



$$X = 2$$

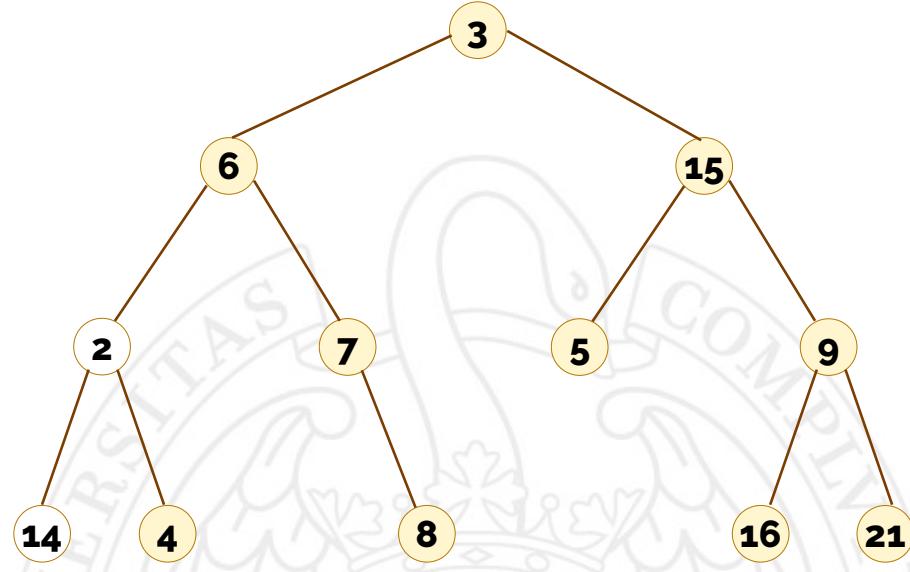
2
6
3

# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - ...
  - ...



$$X = 2$$

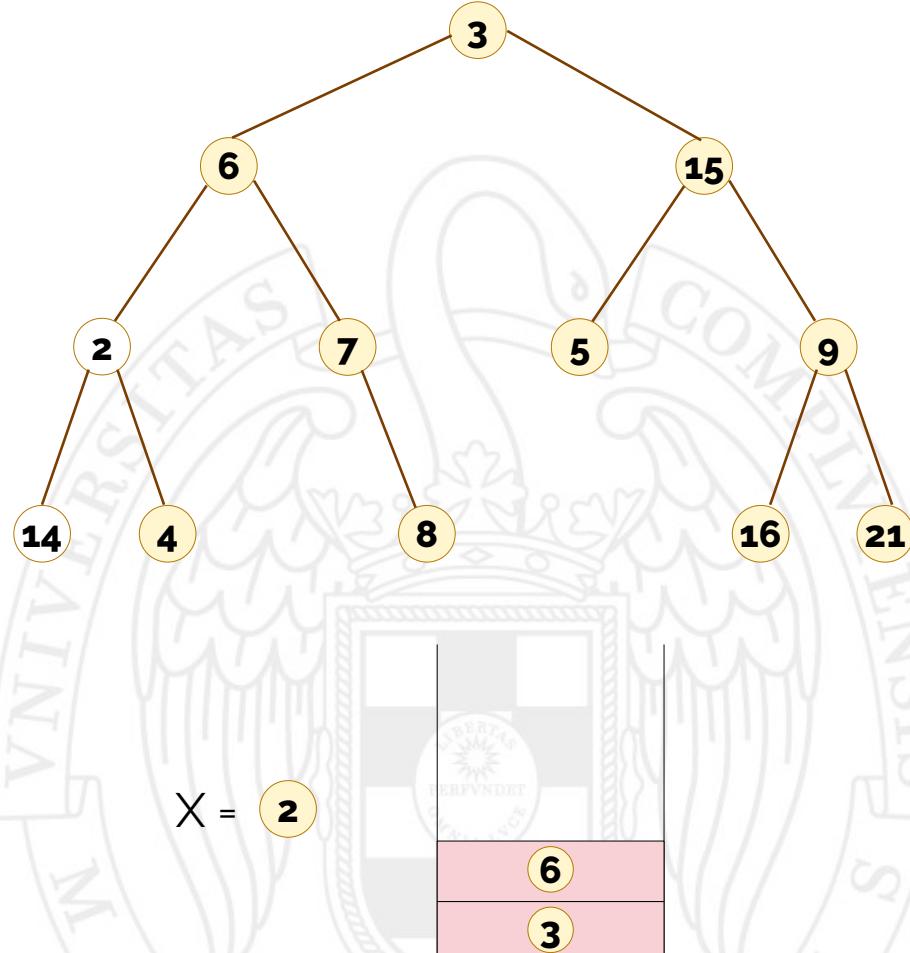
6
3

# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

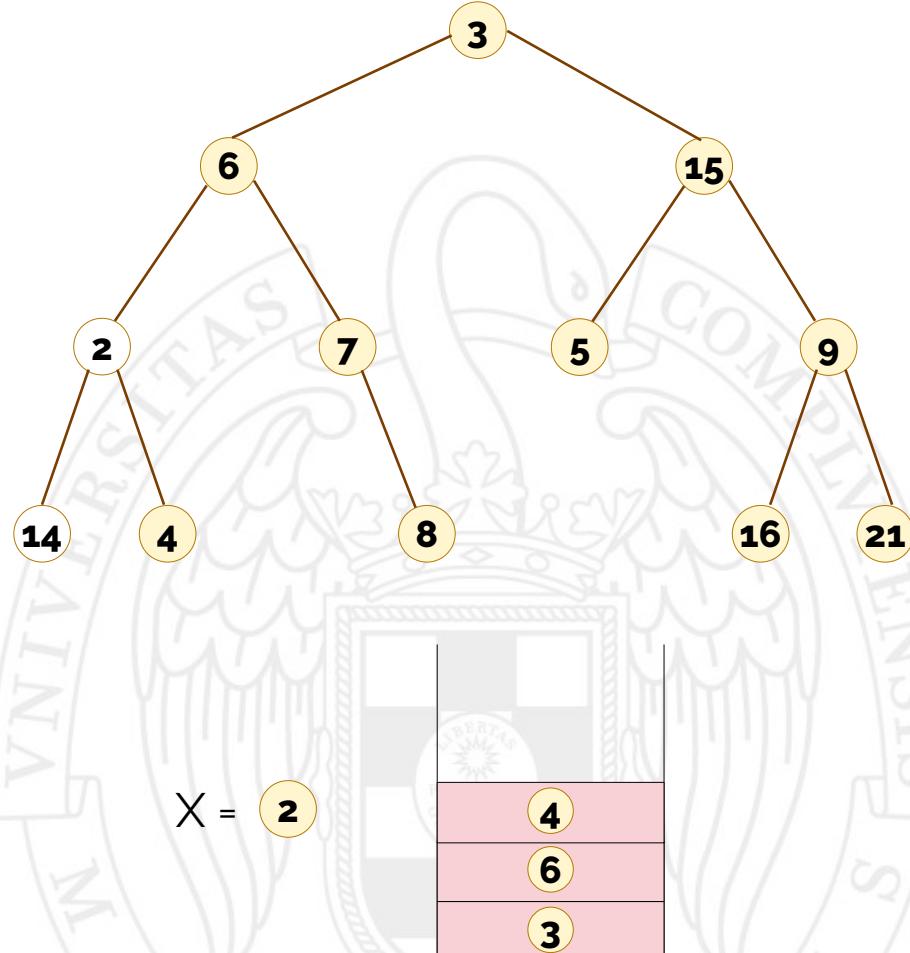


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

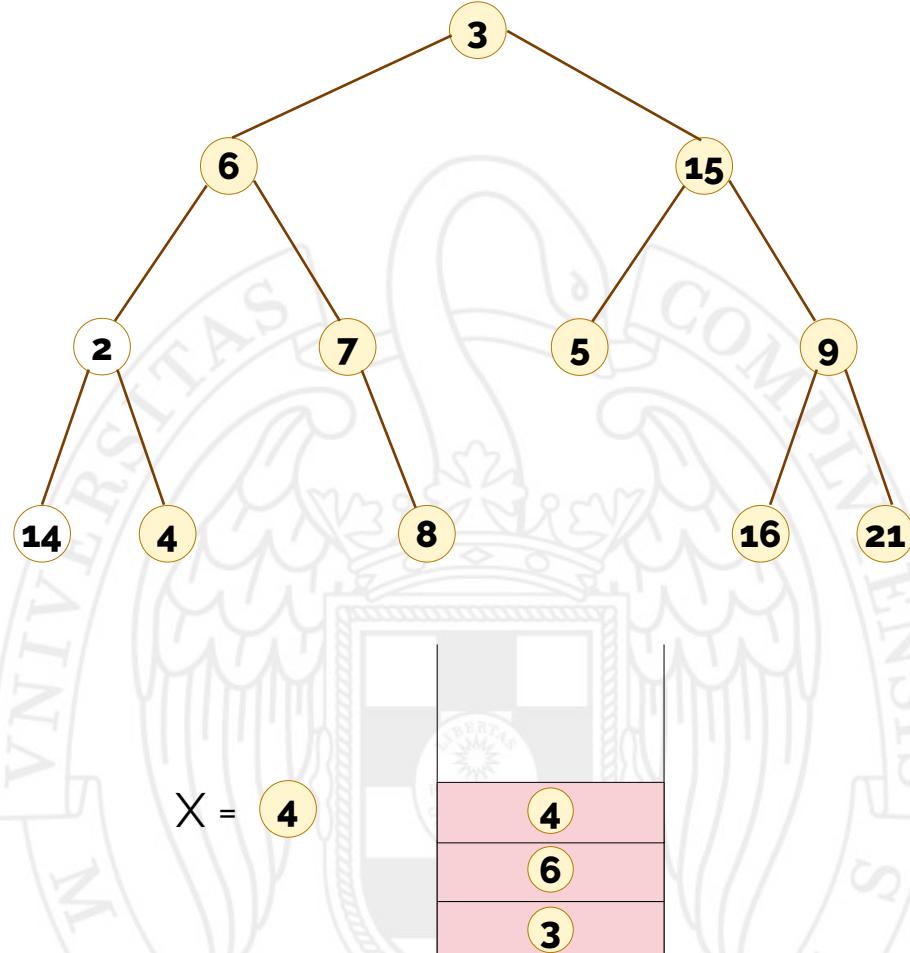


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

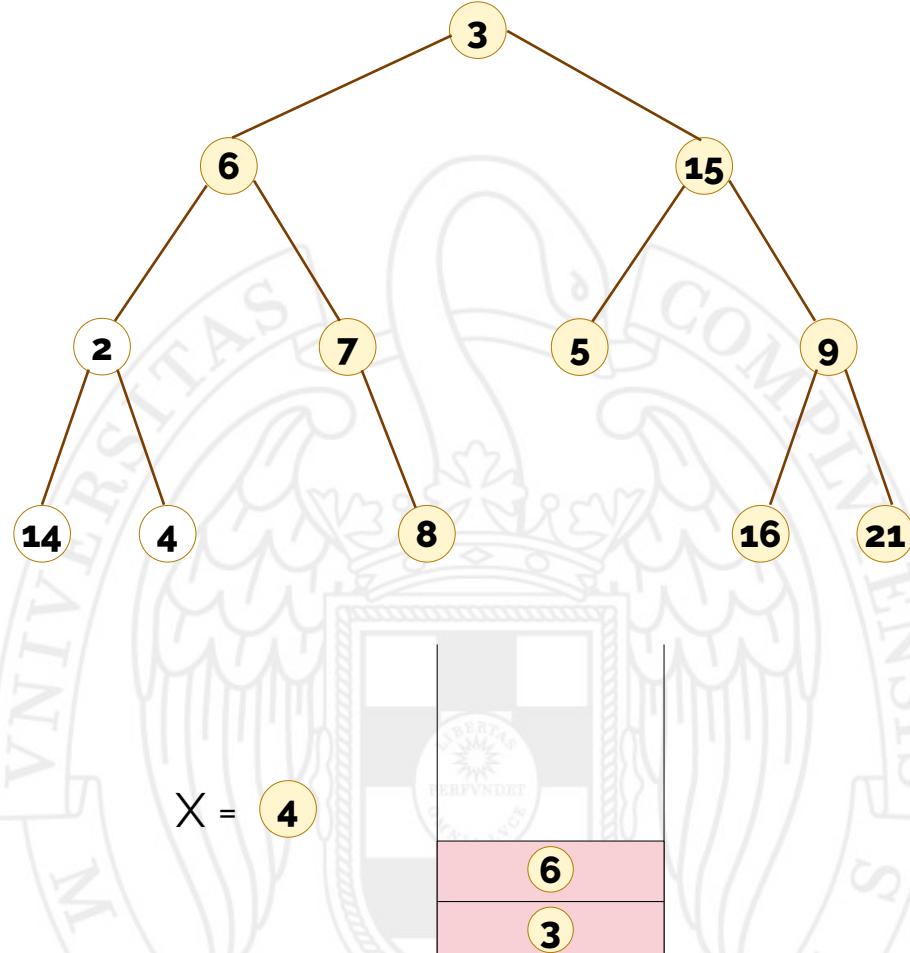


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

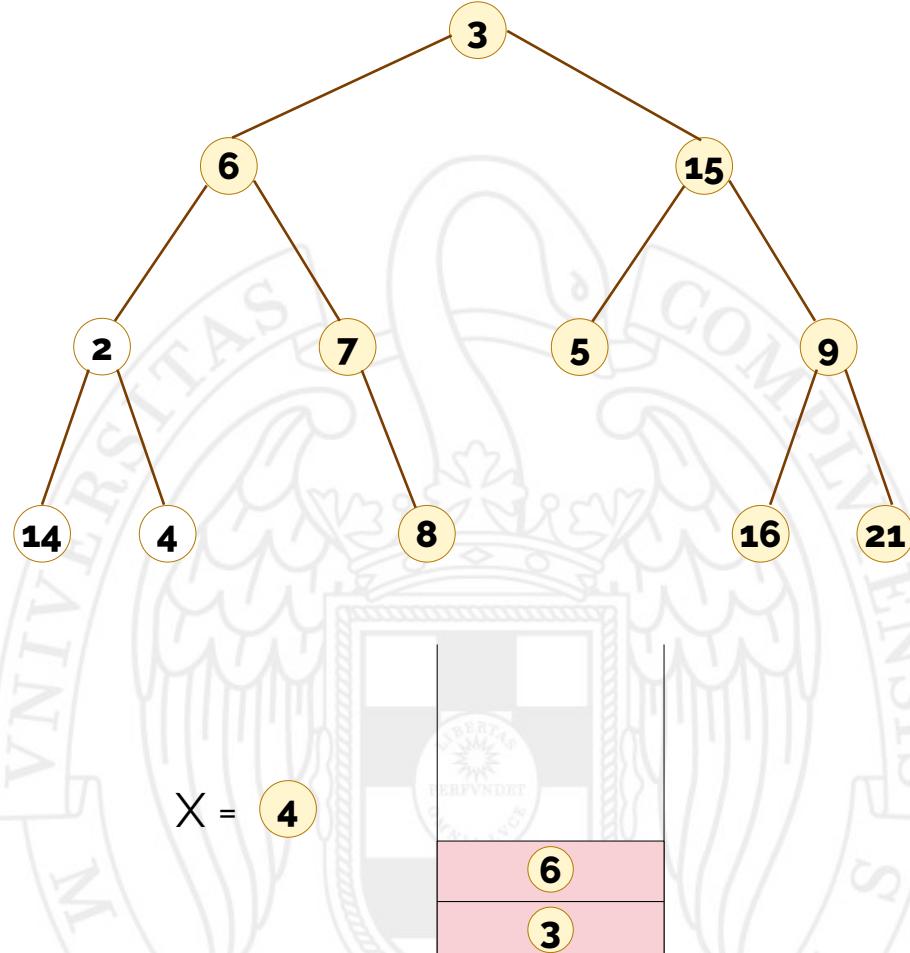


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

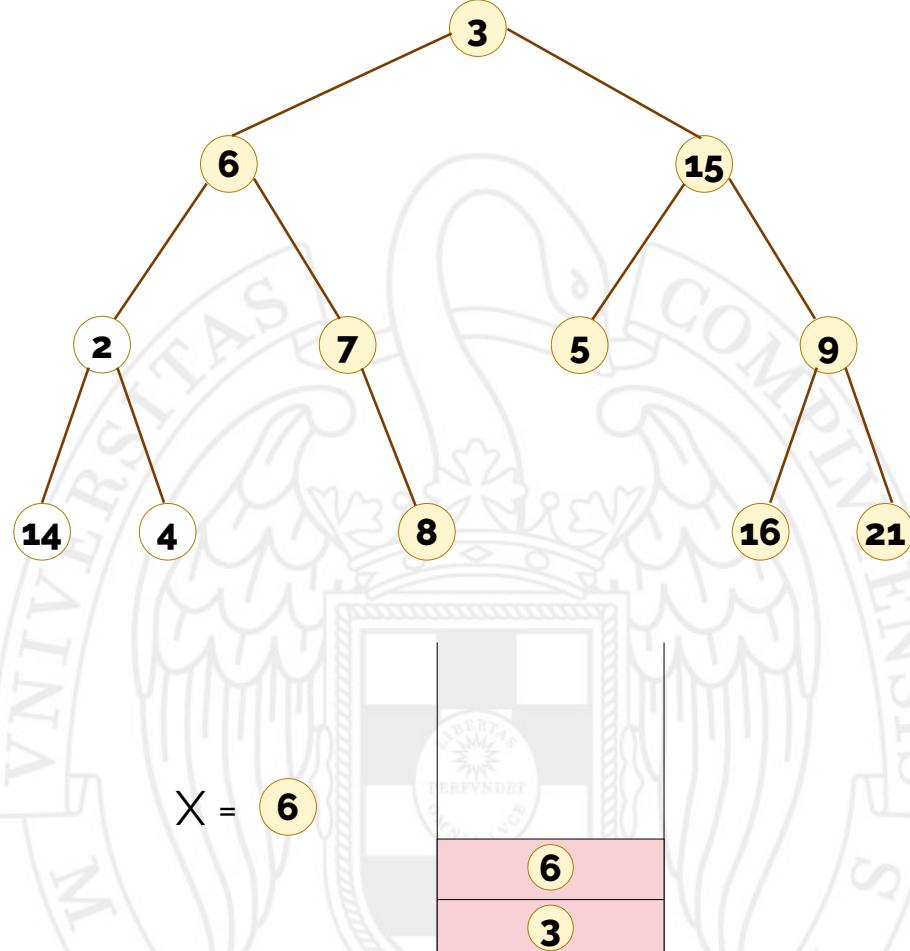


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

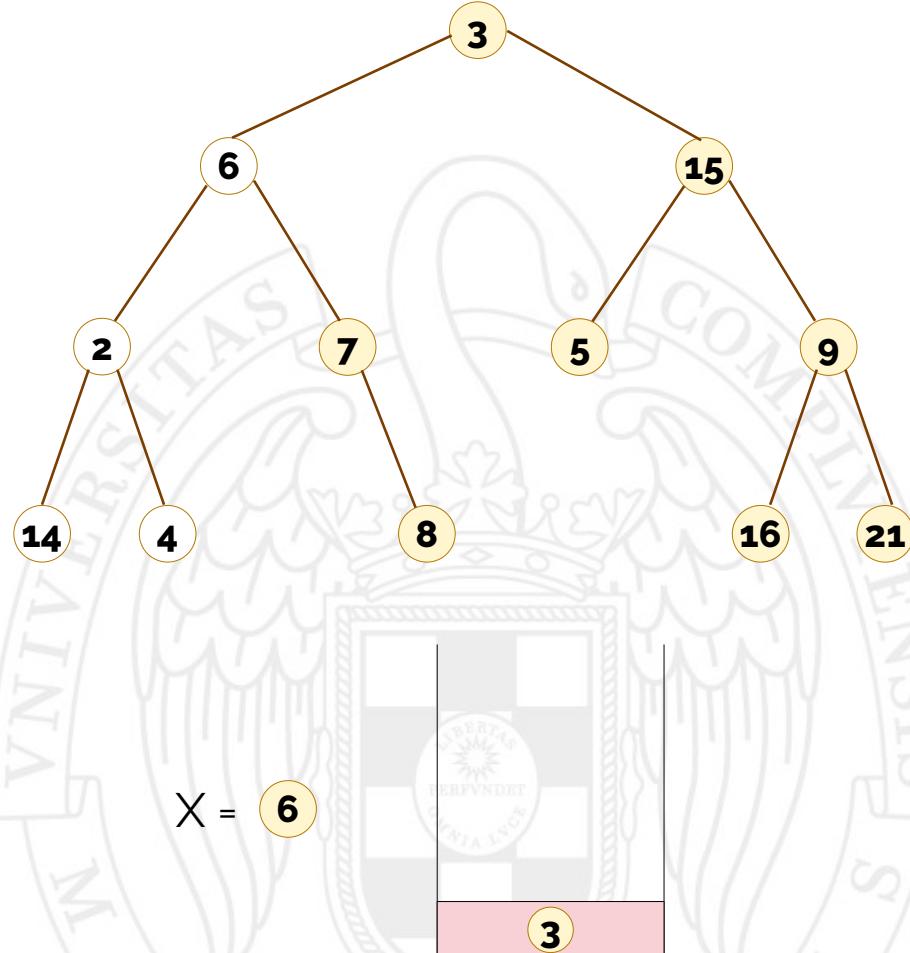


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

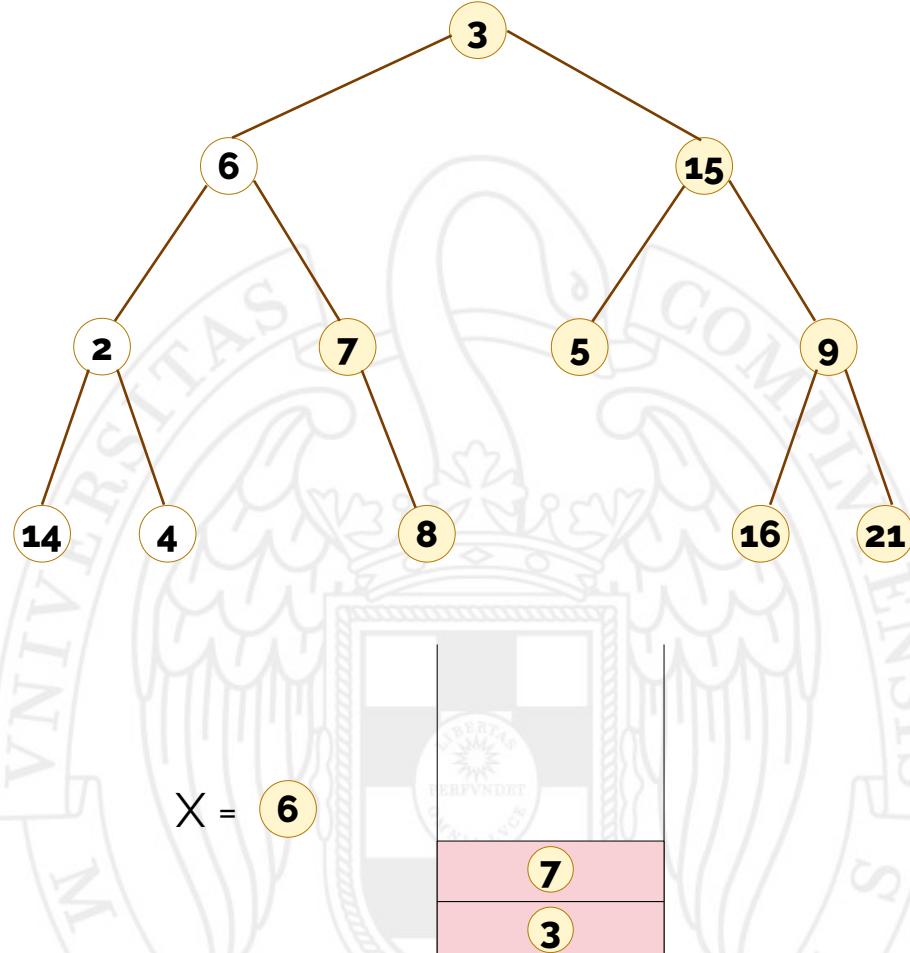


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

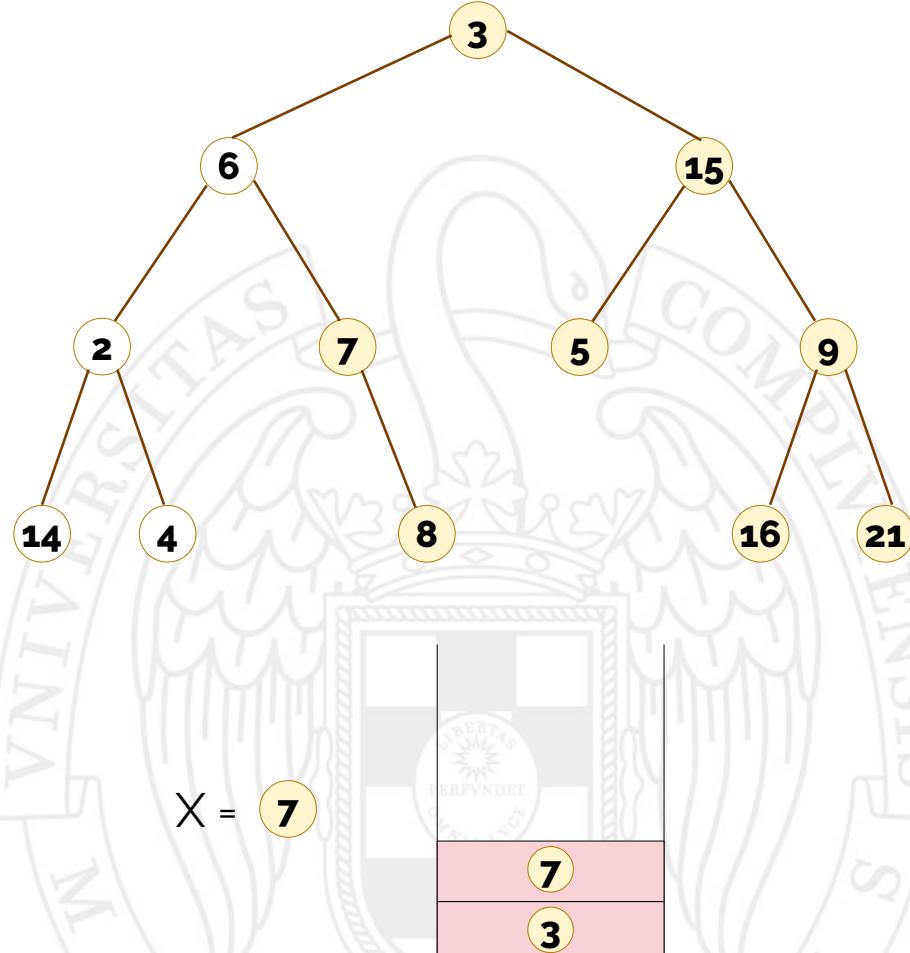


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

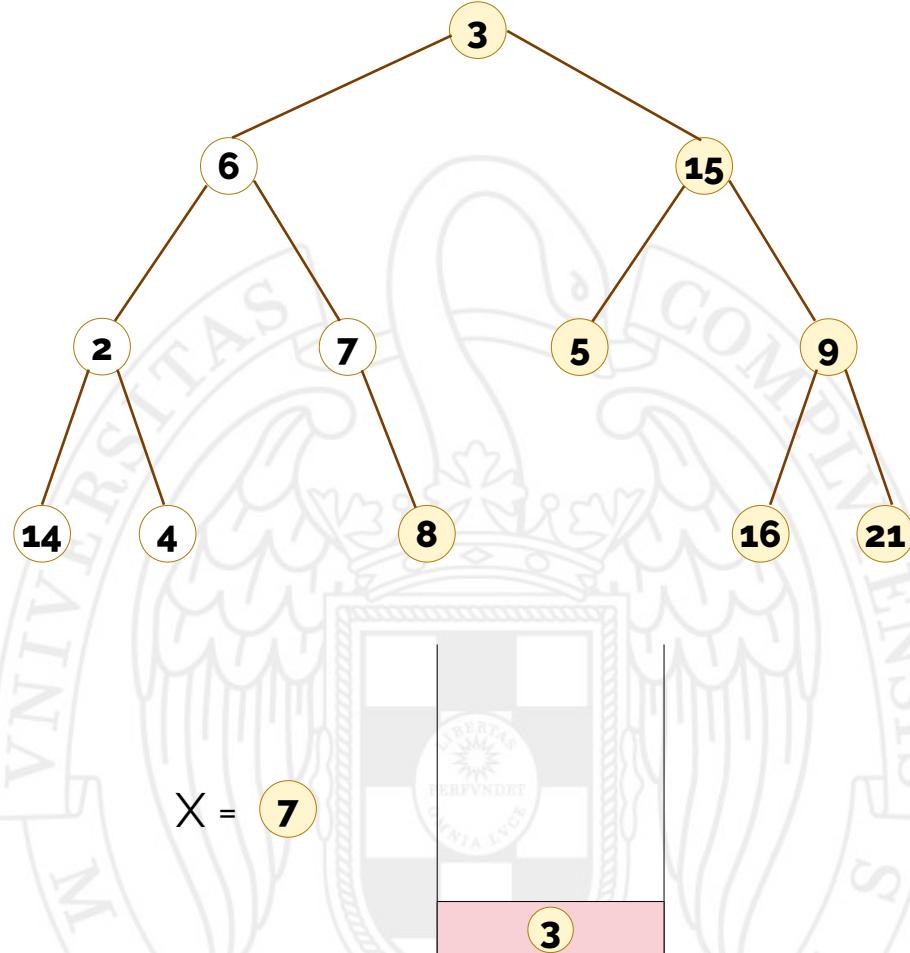


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

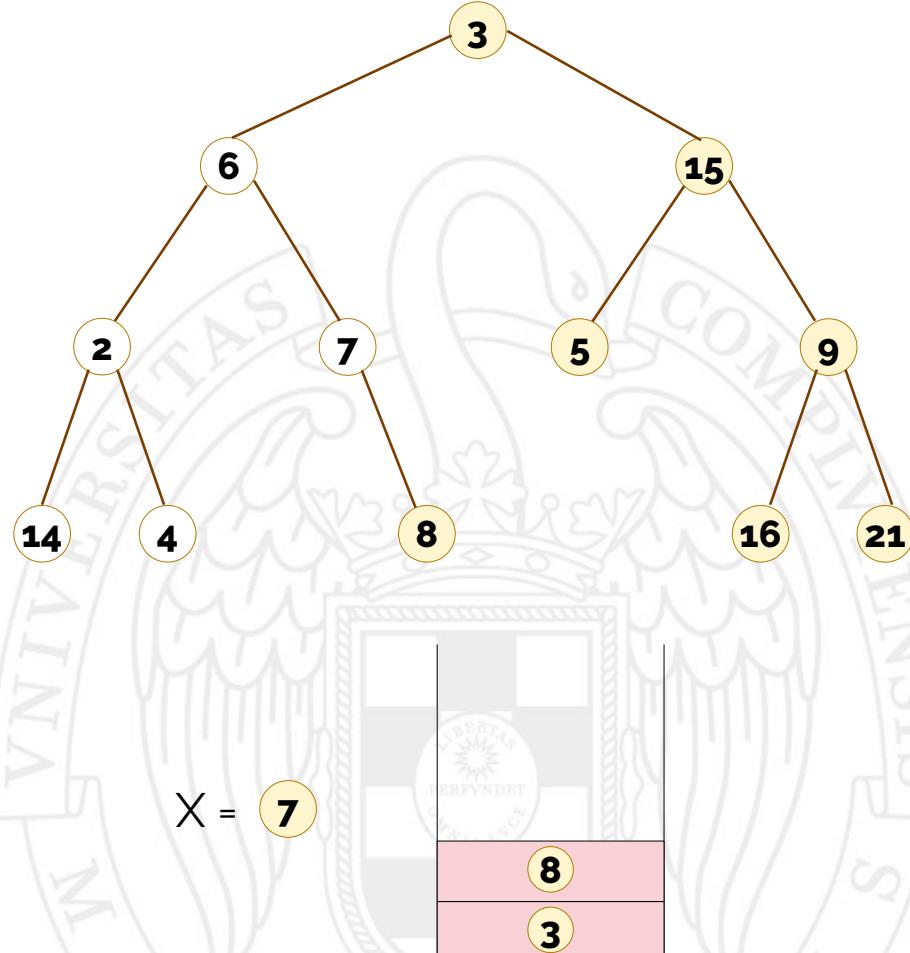


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

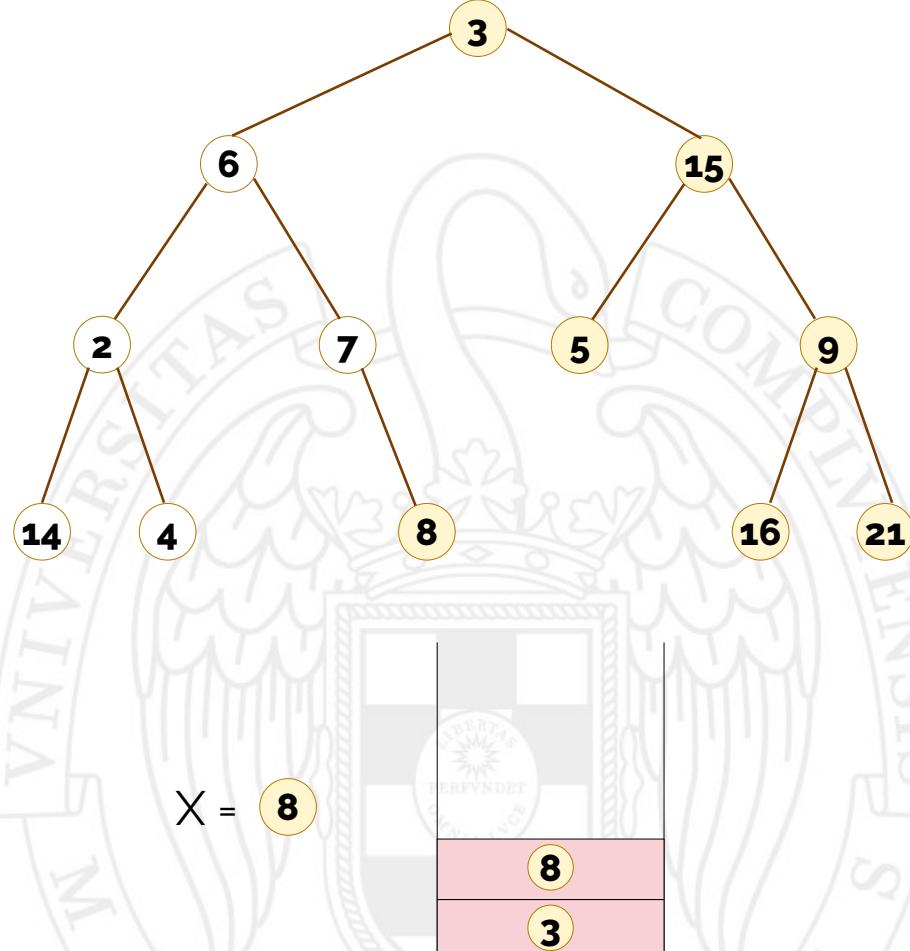


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

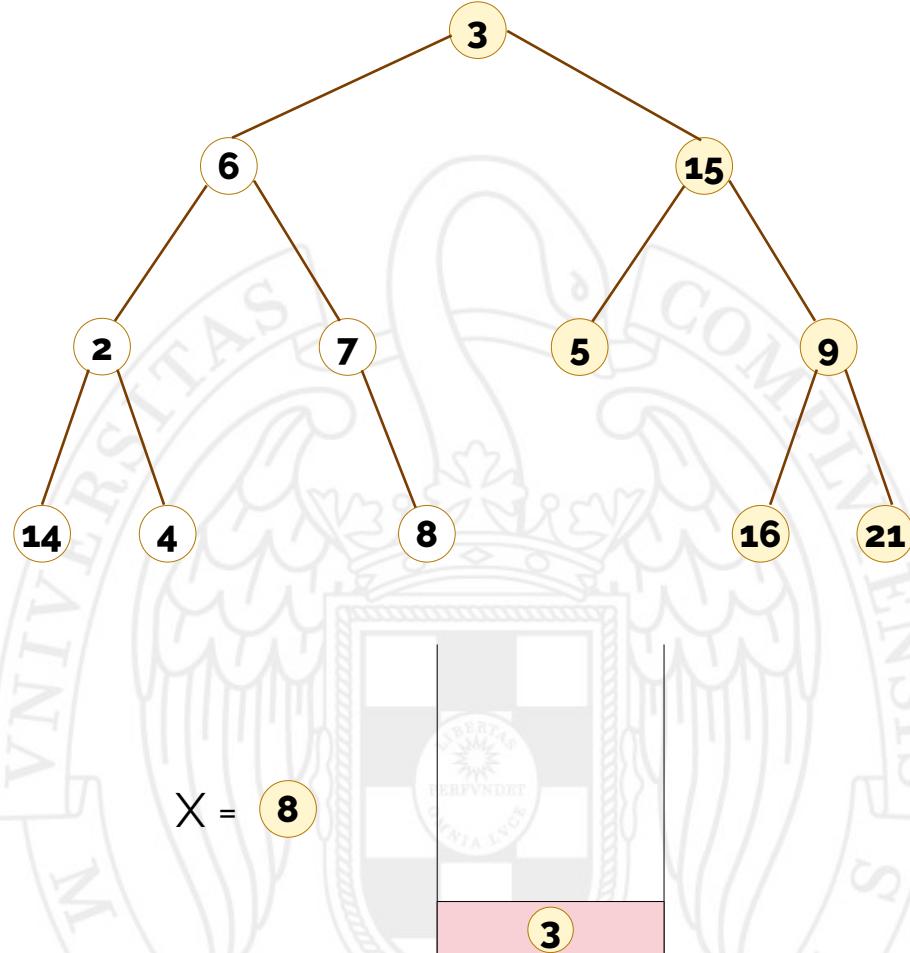


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

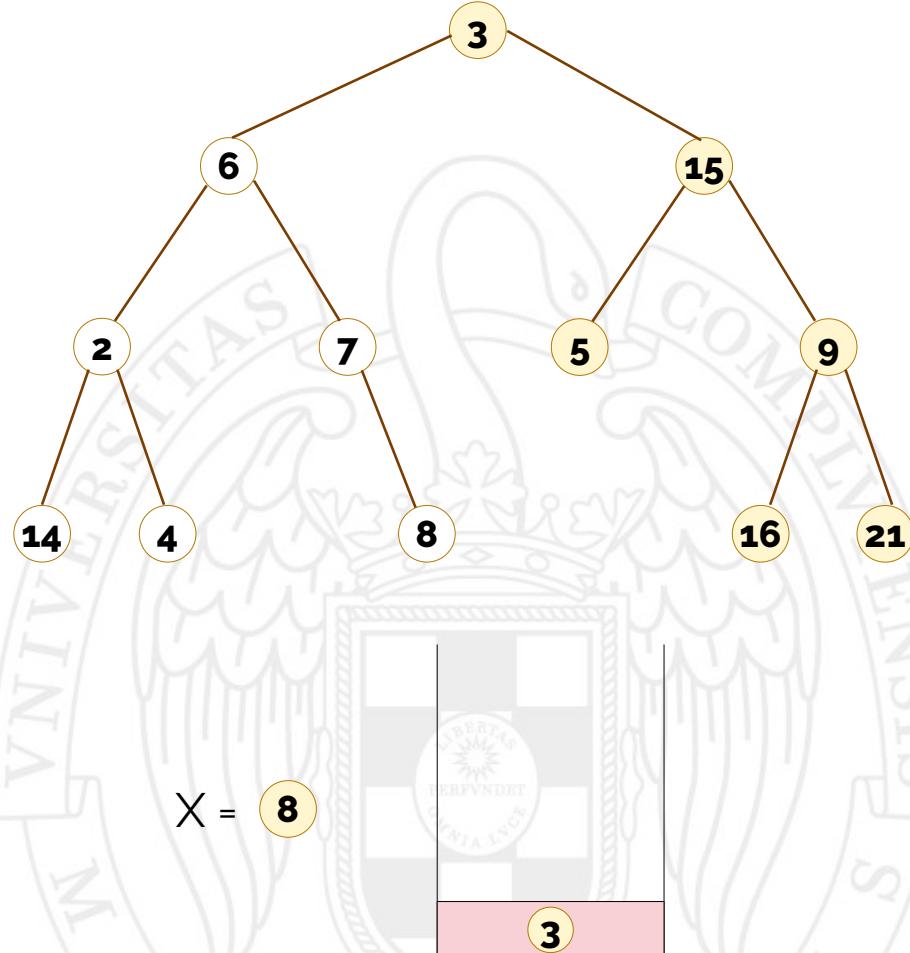


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

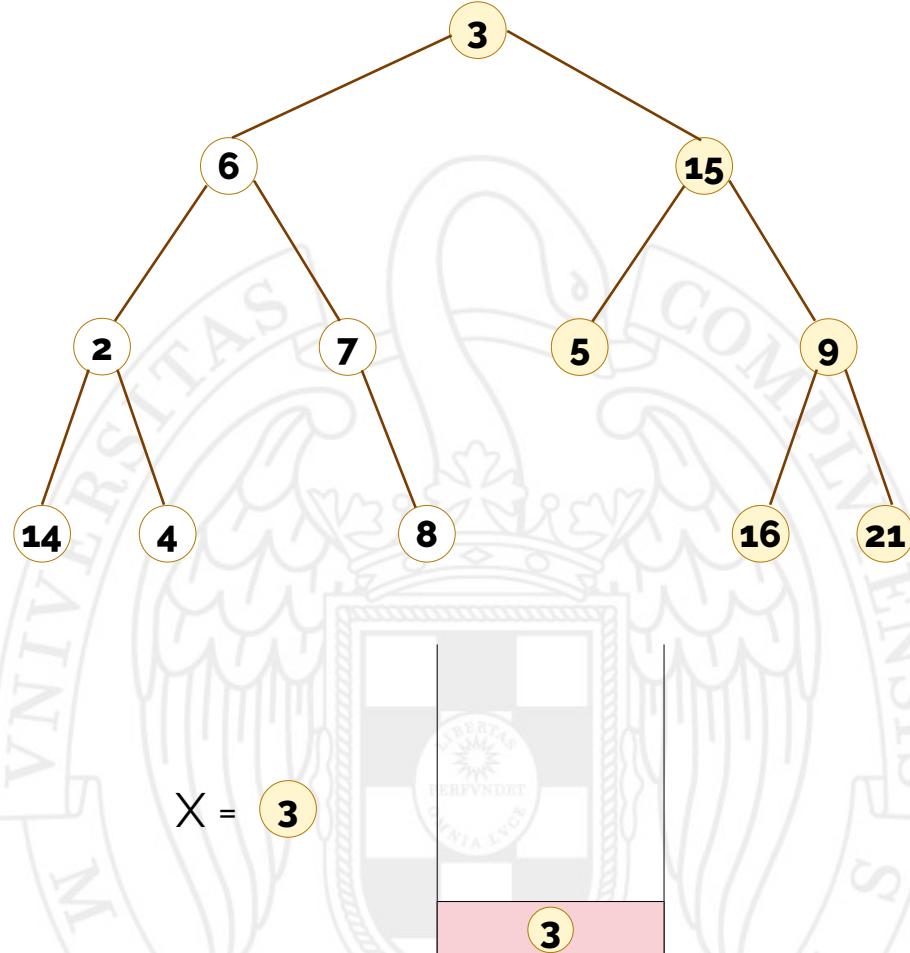


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

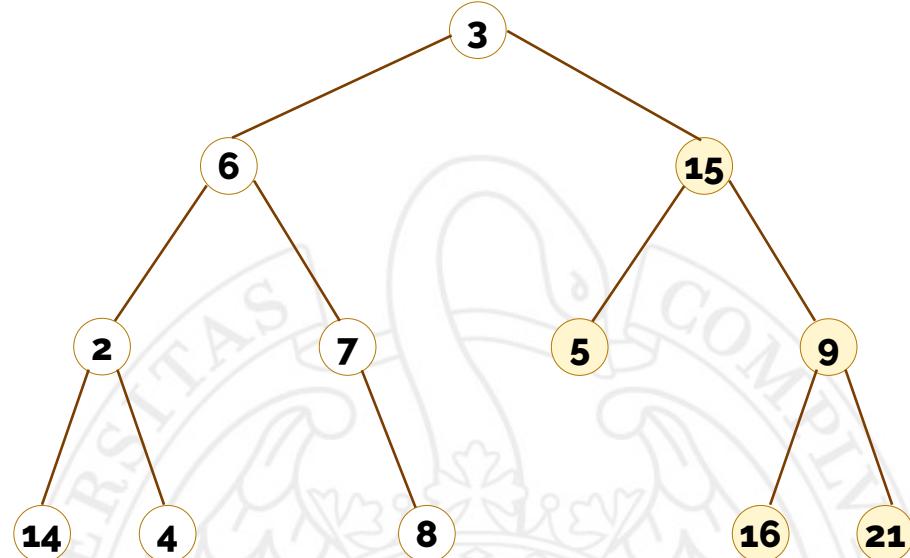


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.



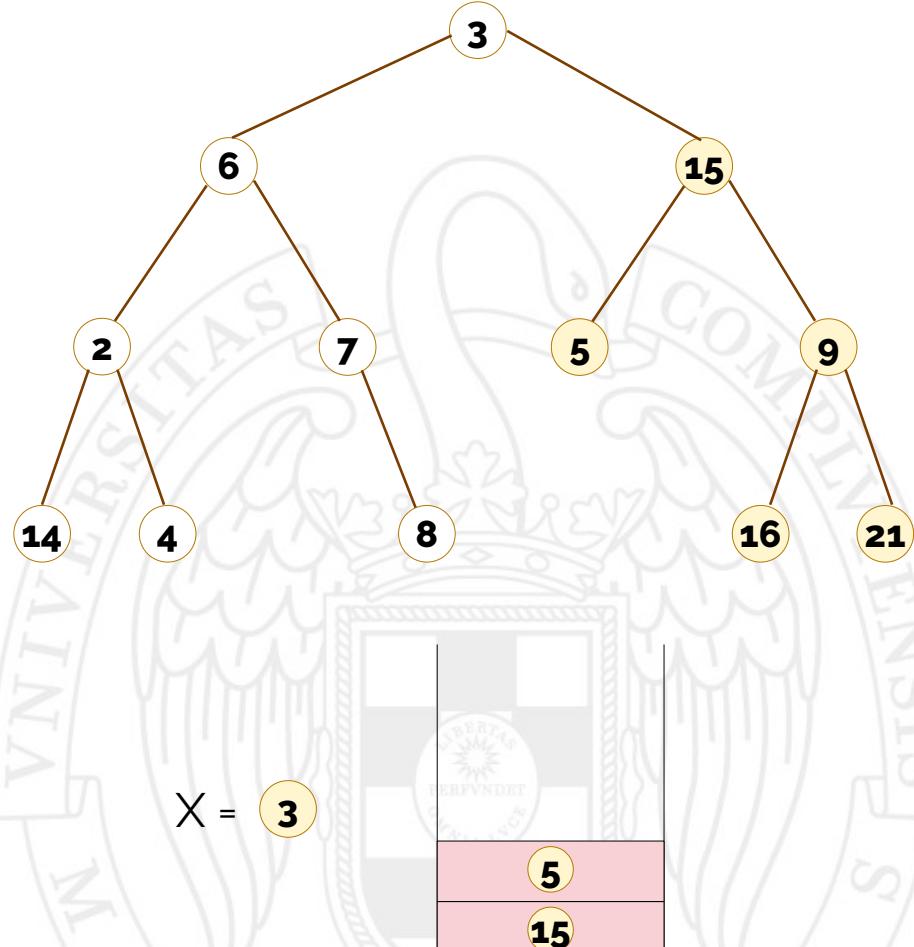
X = 3

# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

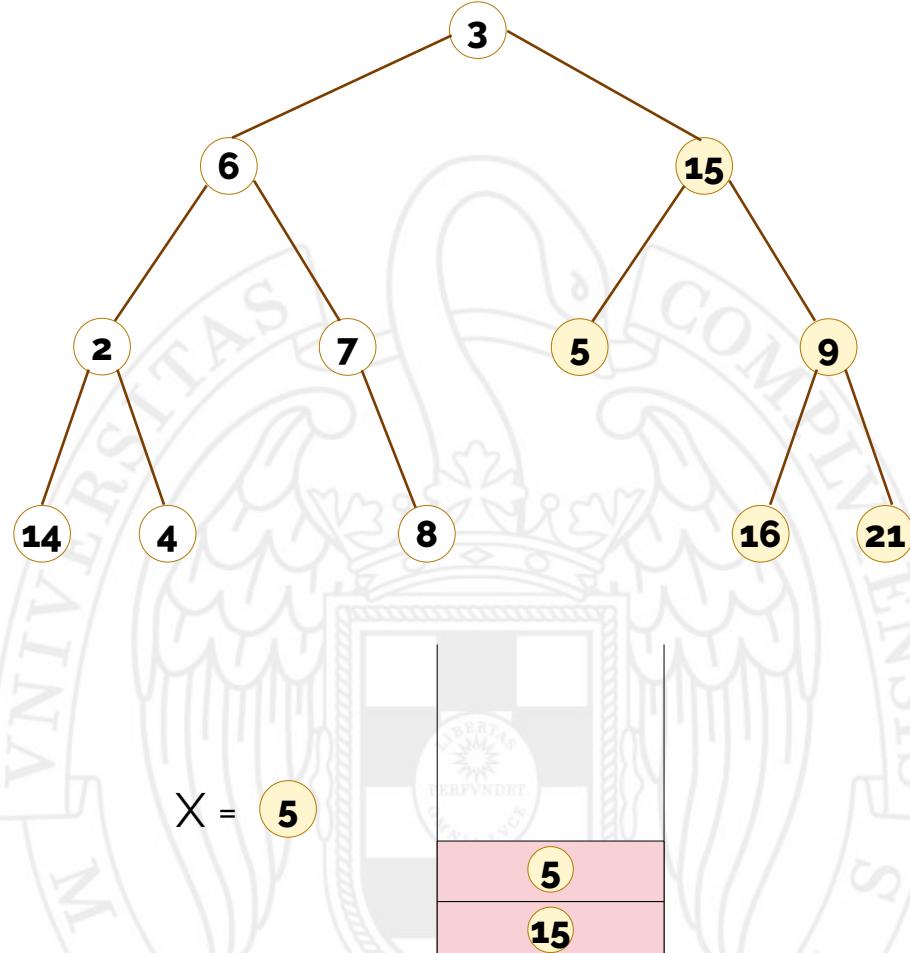


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

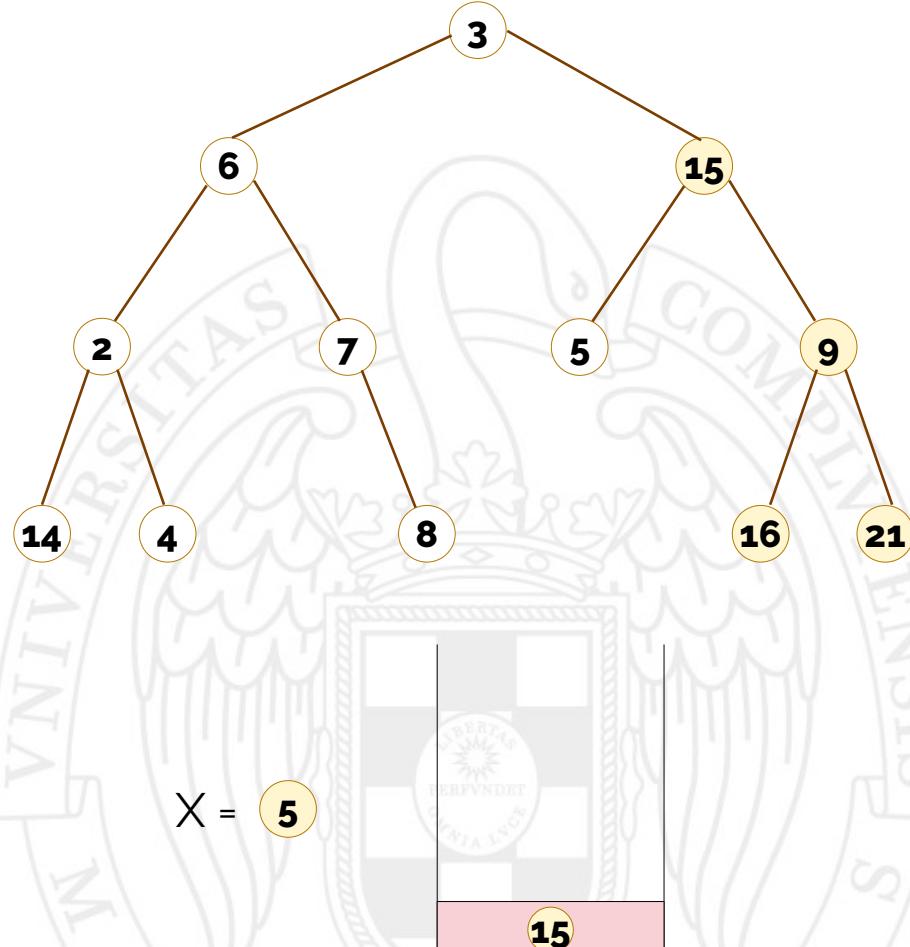


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

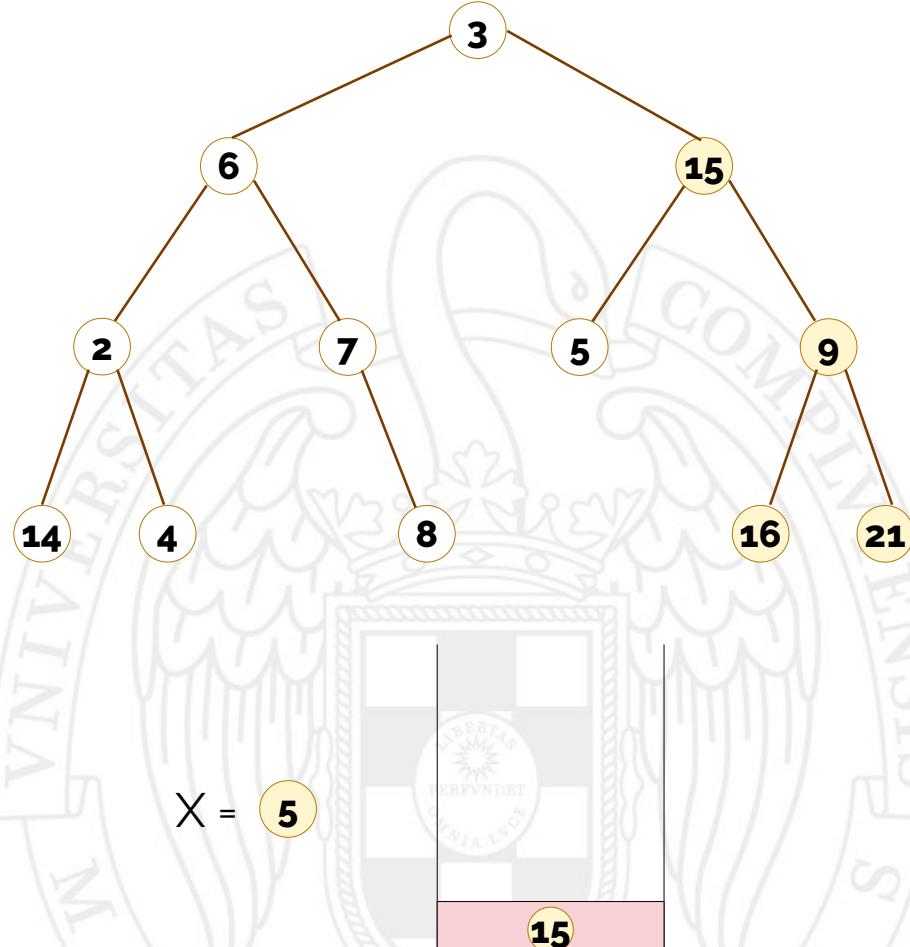


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

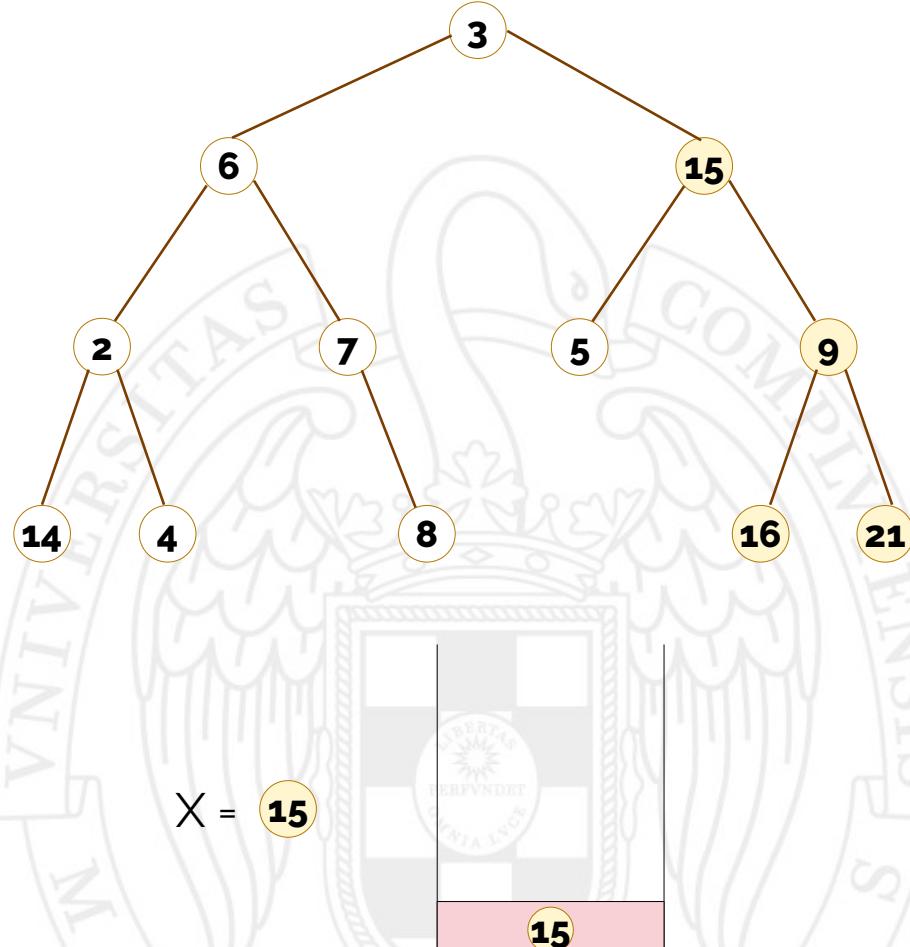


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

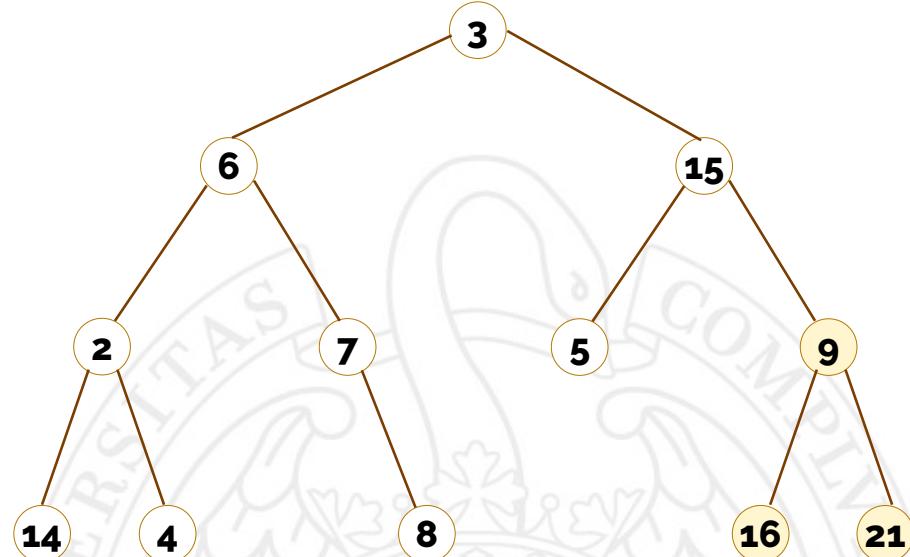


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.



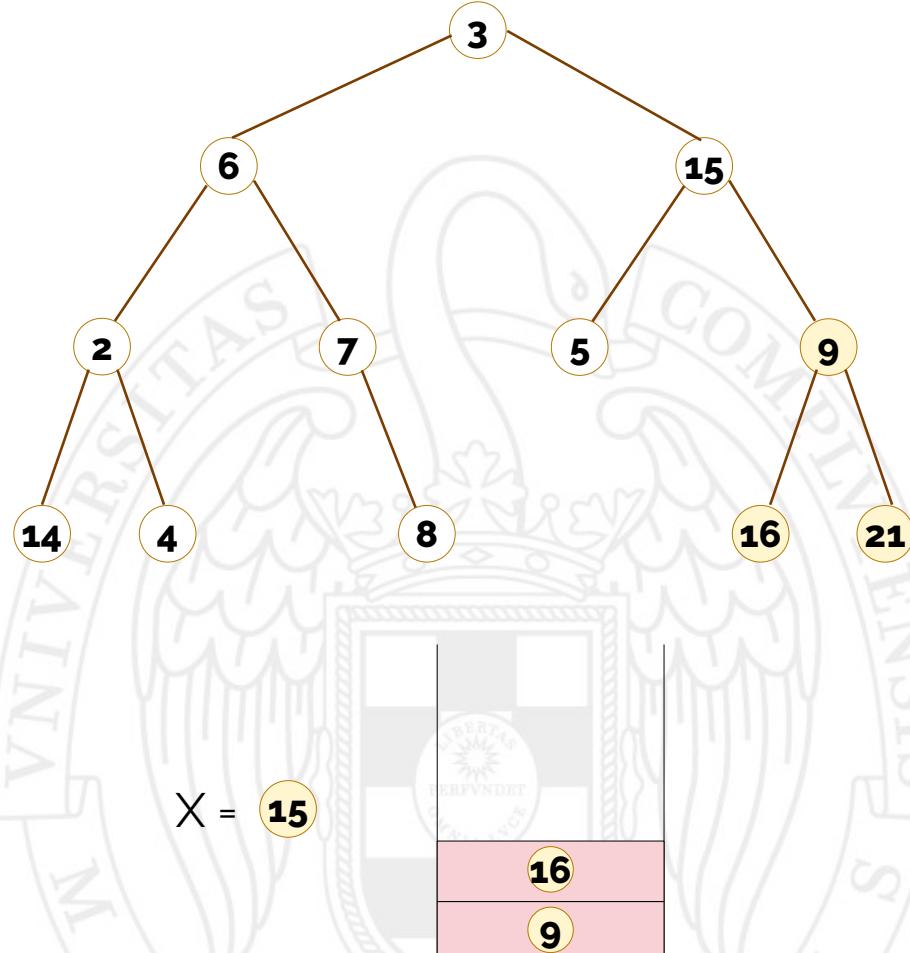
X = 15

# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

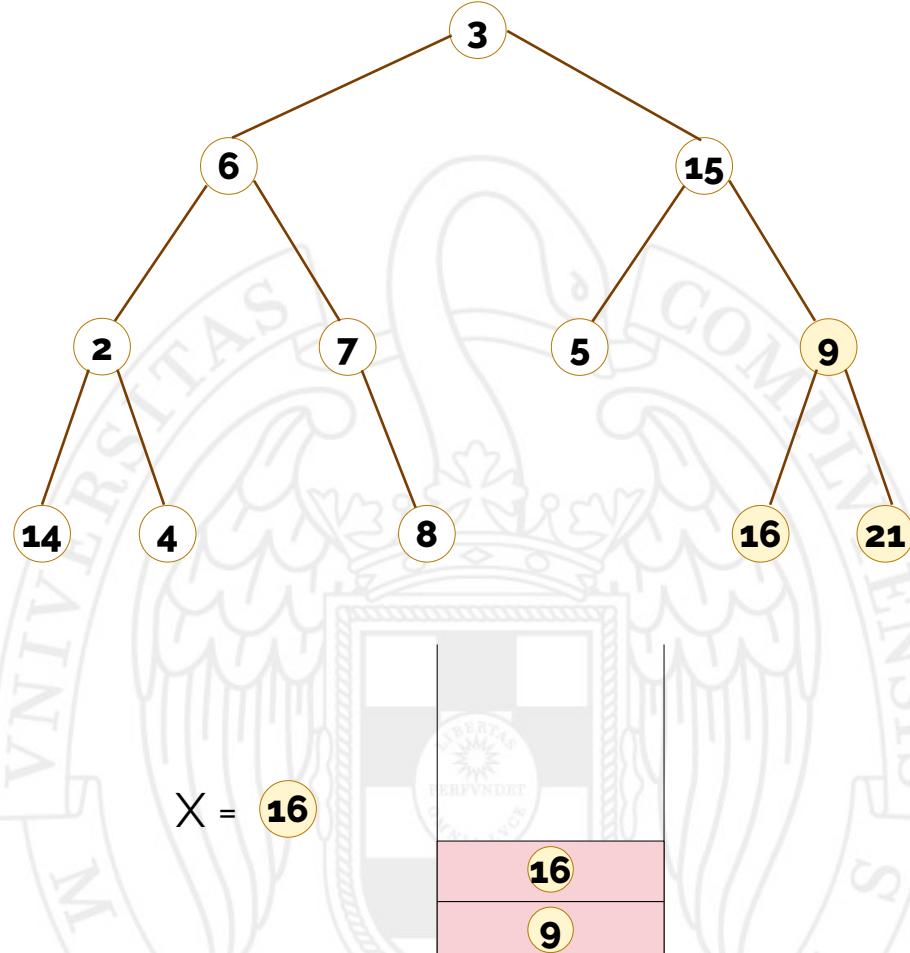


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

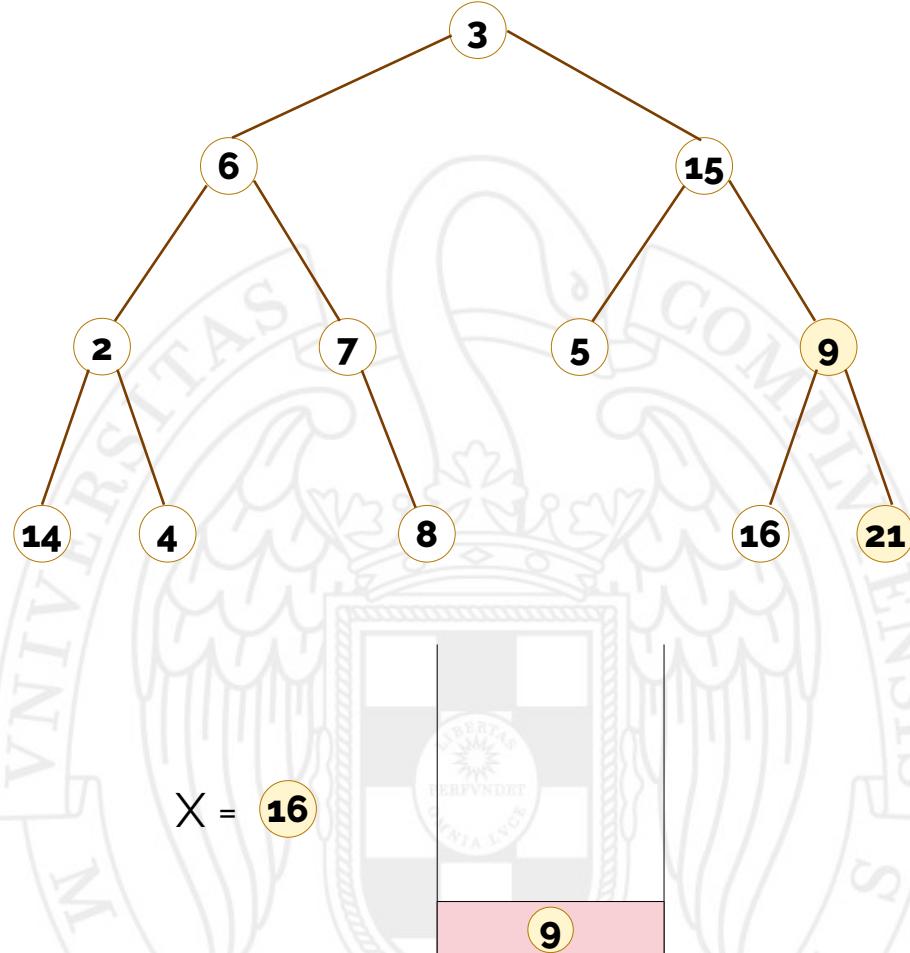


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

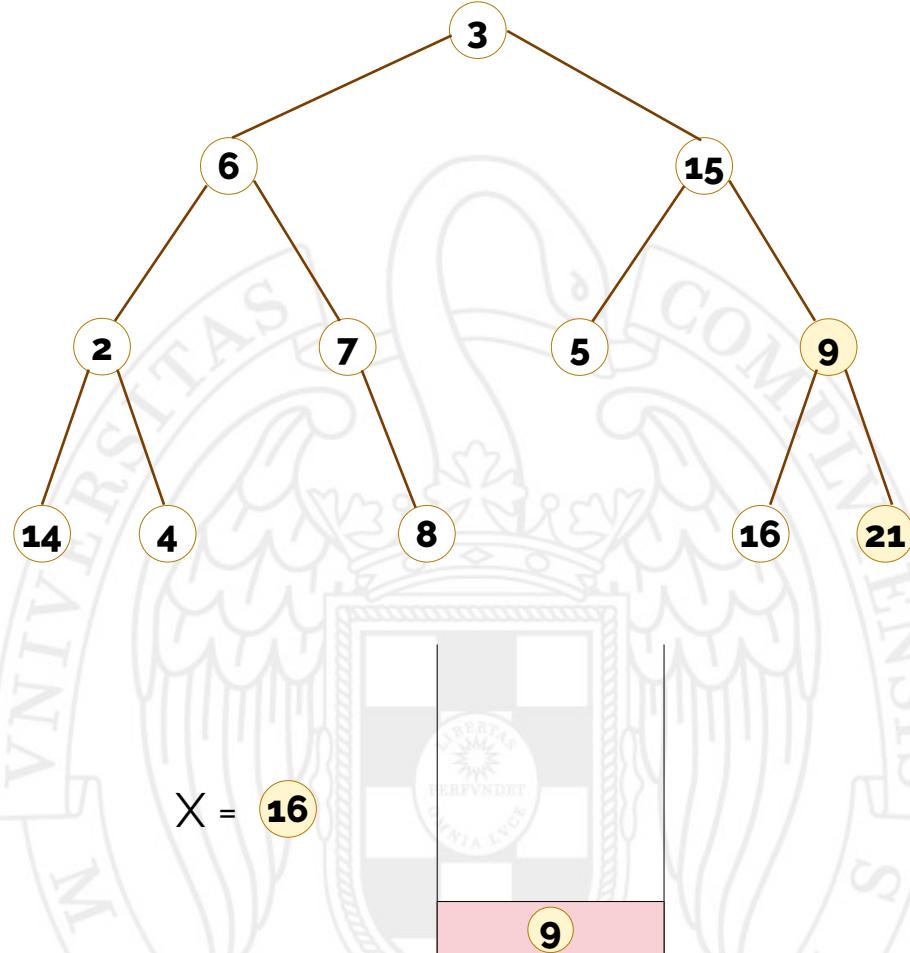


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

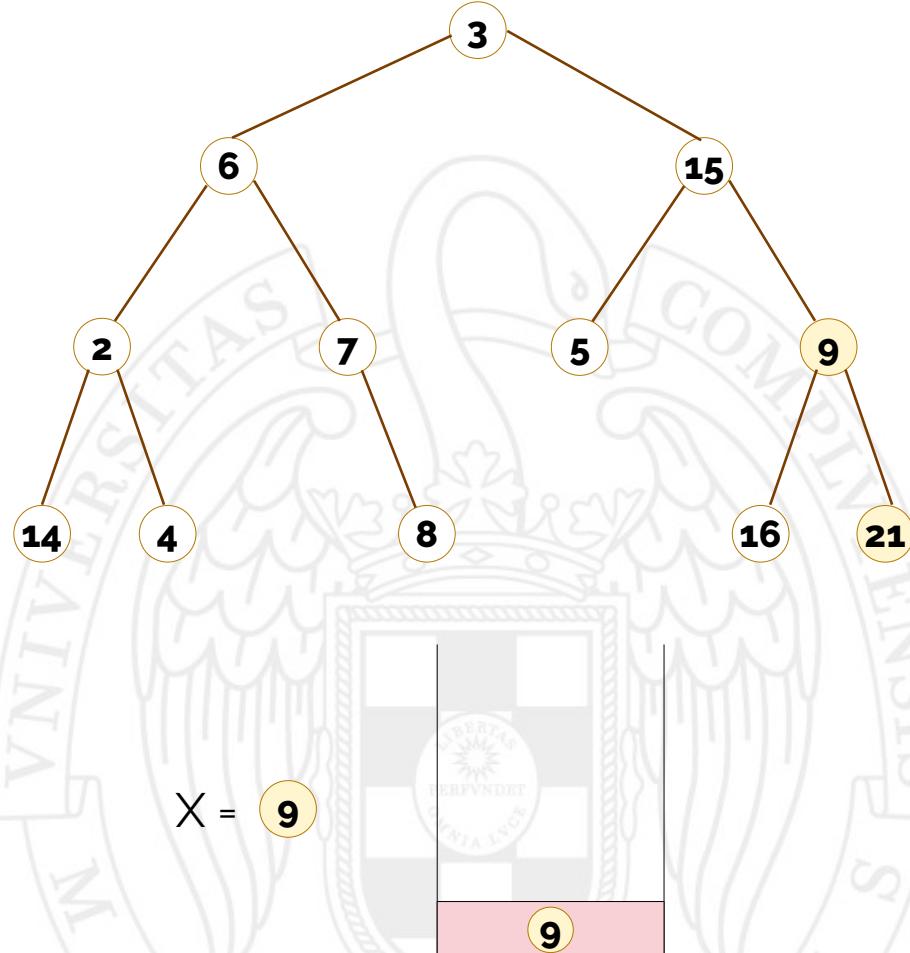


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

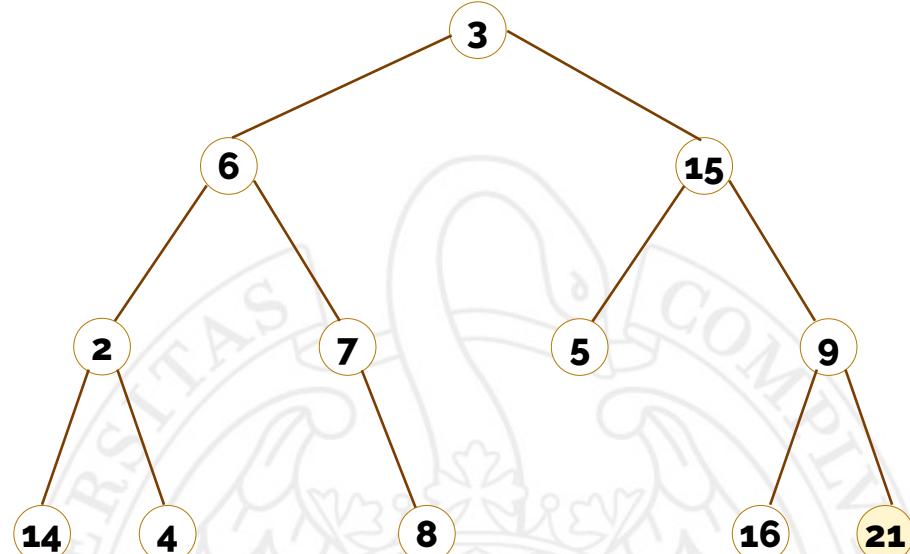


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.



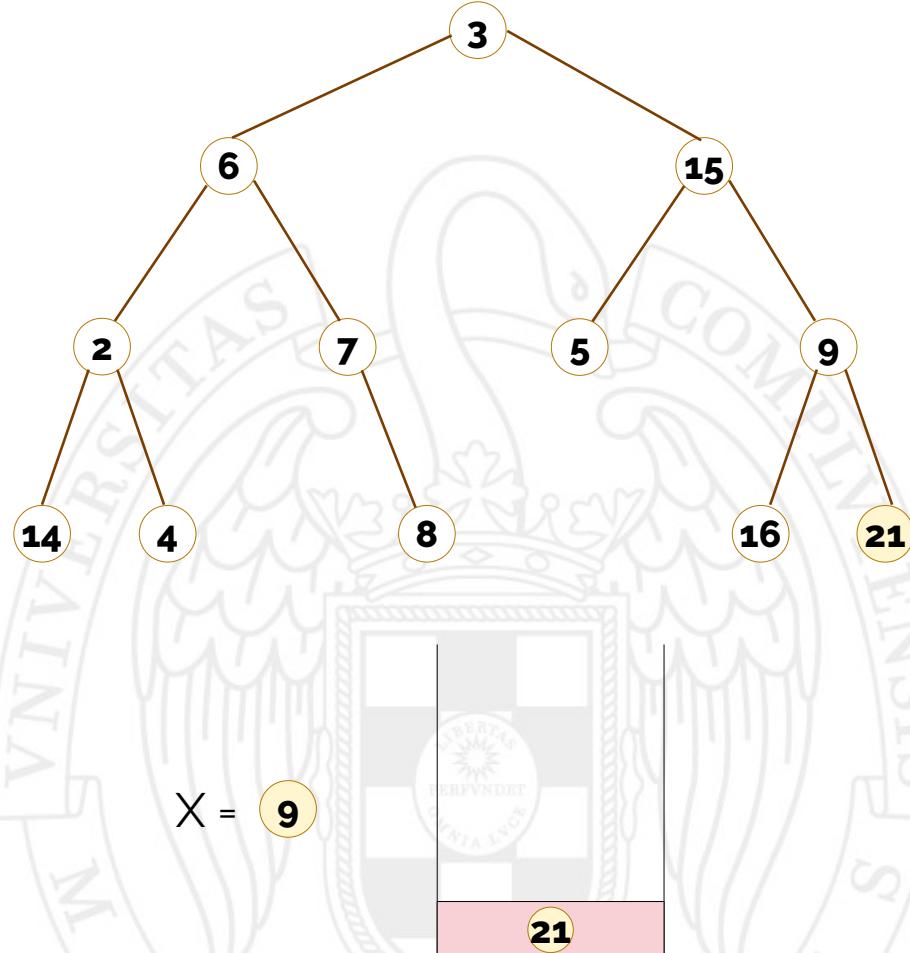
X = 9

# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

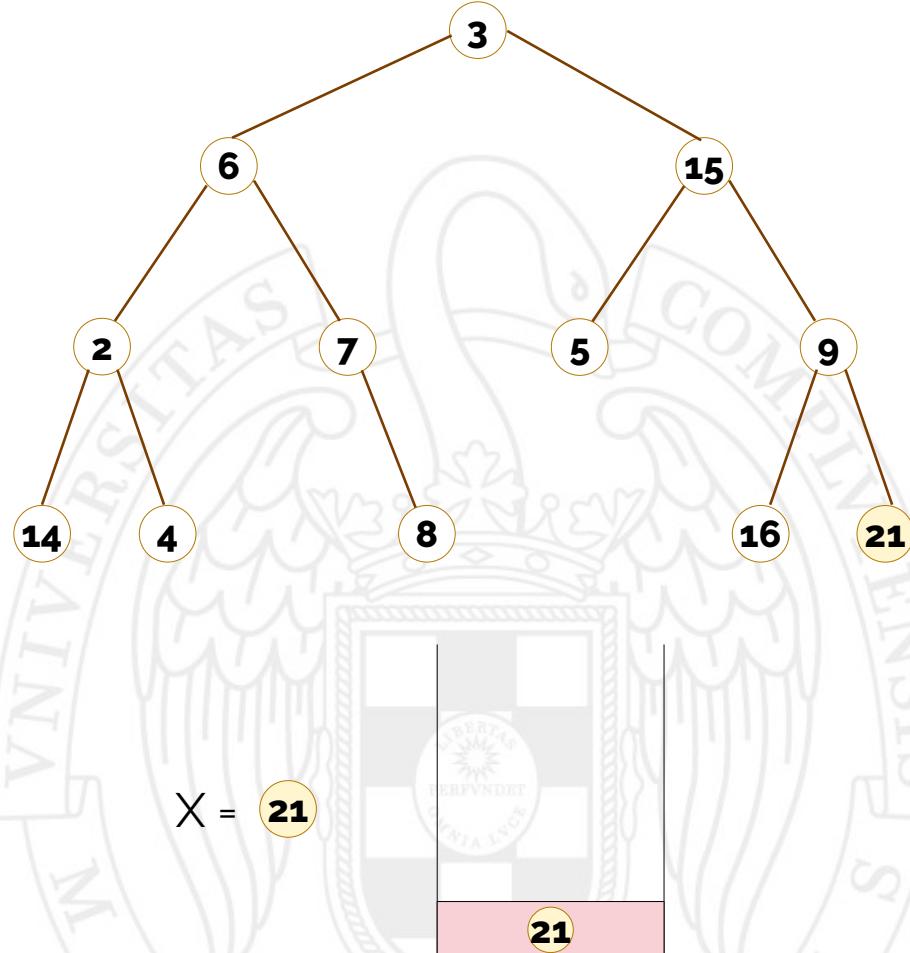


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

## Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
  - Visitar X.
  - Si X tiene hijo derecho:
    - Bajar al hijo derecho de X.
    - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

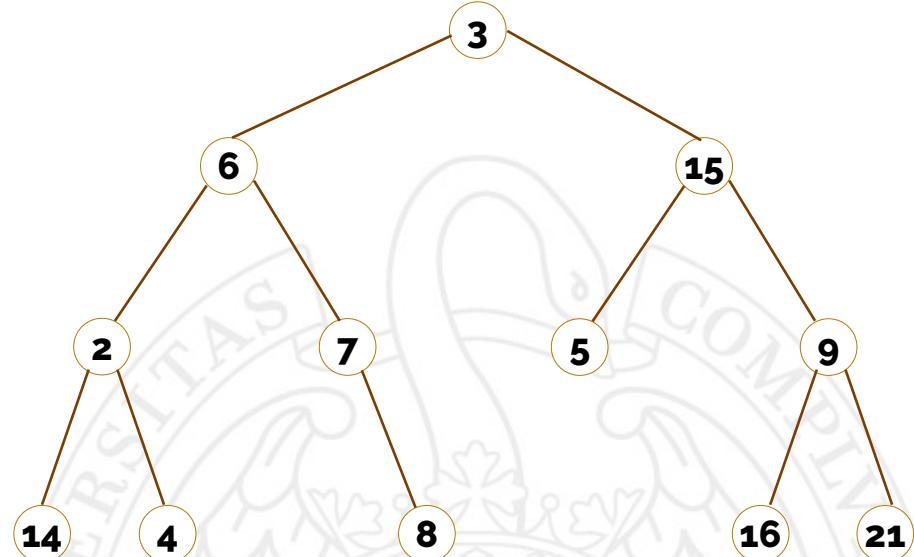


# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.



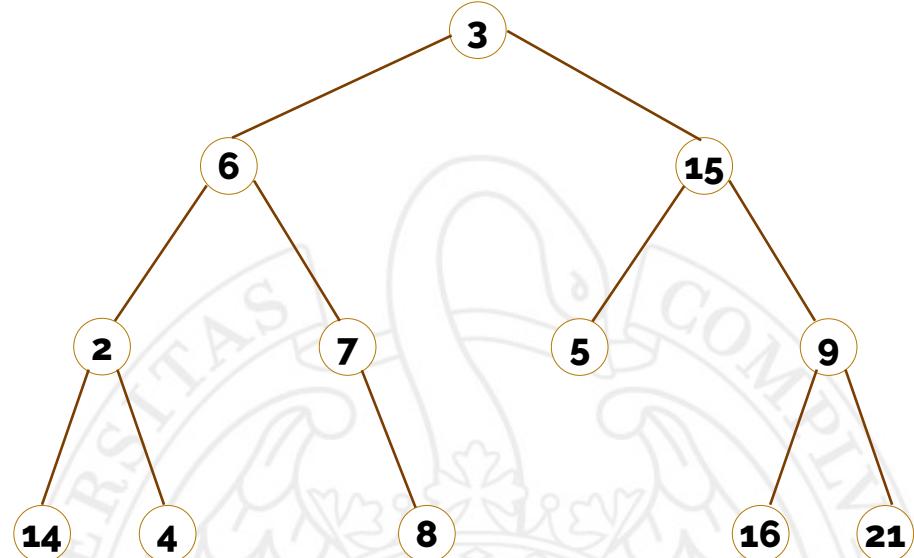
X = 21

# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.



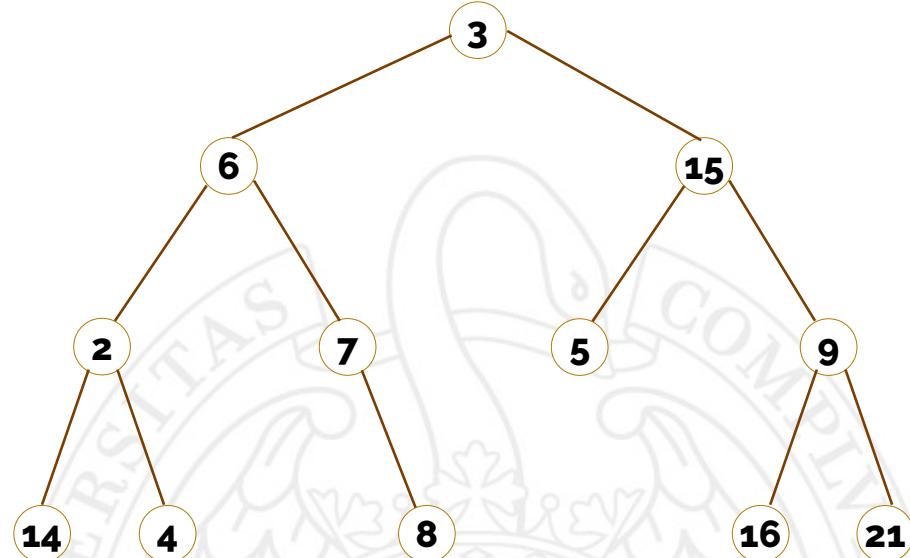
X = 21

# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir **mientras la pila no esté vacía**:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.



X = 21

# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir mientras la pila no esté vacía:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

```
void descend_and_push(const NodePointer &node,  
                      std::stack<NodePointer> &st) {  
    NodePointer current = node;  
    while (current != nullptr) {  
        st.push(current);  
        current = current->left;  
    }  
}
```

# Algoritmo de recorrido

Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

Repetir mientras la pila no esté vacía:

- Sacar el nodo de la cima de la pila. Lo llamamos X.
- Visitar X.
- Si X tiene hijo derecho:
  - Bajar al hijo derecho de X.
  - Descender por los hijos izquierdos mientras sea posible. Apilar los nodos encontrados en el camino.

```
template <typename U>
void inorder(U func) const {

    std::stack<NodePointer> st;
    descend_and_push(root_node, st);

    while (!st.empty()) {
        NodePointer x = st.top();
        st.pop();
        func(x->elem);
        descend_and_push(x->right, st);
    }
}
```

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

# Recorrido en inorden iterativo (2)

Manuel Montenegro Montes

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

# Objetivo

- Aplicar técnicas de transformación de programas:

```
void inorder(NodePointer &node) {  
    if (node != nullptr) {  
        inorder(node→left);  
        visit(node→elem);  
        inorder(node→right);  
    }  
}
```



```
void inorder(NodePointer &node) {  
  
    std::stack<NodePointer> st;  
    descend_and_push(node, st);  
  
    while (!st.empty()) {  
        NodePointer x = st.top();  
        st.pop();  
        visit(x→elem);  
        descend_and_push(x→right, st);  
    }  
}
```

# Transformación de funciones recursivas finales

# Recordatorio: funciones recursivas

- Esquema general de una función recursiva simple:

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursivo();  
        f(x);  
        post_recursivo();  
    }  
}
```

- Una función es **recursiva final** (o recursiva de cola) si finaliza justo después de la llamada recursiva.
- Es decir, si no se realiza ninguna acción en *post\_recursivo()*.

# Transformación a iterativo

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursivo();  
        f(x);  
    }  
}
```

# Transformación a iterativo

*previo();*

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursivo();  
        f(x);  
    }  
}
```

# Transformación a iterativo

```
previo();  
{suponemos !es_caso_base(x)}
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursivo();  
        f(x);  
    }  
}
```

# Transformación a iterativo

```
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursivo();  
        f(x);  
    }  
}
```

# Transformación a iterativo

```
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursivo();  
        f(x);  
    }  
}
```

# Transformación a iterativo

```
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();  
previo();
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursivo();  
        f(x);  
    }  
}
```

# Transformación a iterativo

```
previo();  
{suponemos ~es_caso_base(x)}  
pre_recursivo();  
previo();  
{suponemos ~es_caso_base(x)}
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursivo();  
        f(x);  
    }  
}
```

# Transformación a iterativo

```
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursivo();  
        f(x);  
    }  
}
```

# Transformación a iterativo

```
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursivo();  
        f(x);  
    }  
}
```

# Transformación a iterativo

```
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();  
previo();
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursivo();  
        f(x);  
    }  
}
```

# Transformación a iterativo

```
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();  
previo();  
{suponemos ¬es_caso_base(x)}
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursivo();  
        f(x);  
    }  
}
```

# Transformación a iterativo

```
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
previo();
{suponemos ¬es_caso_base(x)}
pre_recursivo();
```

```
void f(x) {
    previo();
    if (es_caso_base(x)) {
        caso_base();
    } else {
        pre_recursivo();
        f(x);
    }
}
```

# Transformación a iterativo

```
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursivo();  
        f(x);  
    }  
}
```

# Transformación a iterativo

```
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();  
previo();
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursivo();  
        f(x);  
    }  
}
```

# Transformación a iterativo

```
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();  
previo();  
{suponemos es_caso_base(x)}
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursivo();  
        f(x);  
    }  
}
```

# Transformación a iterativo

```
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();  
previo();  
{suponemos es_caso_base(x)}  
caso_base();
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursivo();  
        f(x);  
    }  
}
```

# Transformación a iterativo

```
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();  
previo();  
{suponemos ¬es_caso_base(x)}  
pre_recursivo();  
previo();  
{suponemos es_caso_base(x)}  
caso_base();
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursivo();  
        f(x);  
    }  
}
```

# Transformación a iterativo

```
previo();  
{suponemos !es_caso_base(x)}  
pre_recursivo();  
previo();  
{suponemos !es_caso_base(x)}  
pre_recursivo();  
previo();  
{suponemos !es_caso_base(x)}  
pre_recursivo();  
previo();  
{suponemos es_caso_base(x)}  
caso_base();
```

```
void f(x) {  
    previo();  
    while (!es_caso_base(x)) {  
        pre_recursivo();  
        previo();  
    }  
    caso_base();  
}
```

# Transformación a iterativo

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursivo();  
        f(x);  
    }  
}
```



```
void f(x) {  
    previo();  
    while (!es_caso_base(x)) {  
        pre_recursivo();  
        previo();  
    }  
    caso_base();  
}
```

# Transformación de inorder

# Recorrido en inorder

```
void inorder(NodePointer node) {  
    if (node != nullptr) {  
        inorder(node→left);  
        visit(node);  
        inorder(node→right);  
    }  
}
```



# Dos funciones auxiliares

```
inorder_stack(stack<NodePointer> &st)
```

Desapila todos los elementos de `st`, y para cada uno de ellos:

- Visita su raíz.
- Realiza un recorrido en inorder de su hijo derecho

```
void inorder_stack(stack<NodePointer> &st) {
    if (!st.empty()) {
        NodePointer current = st.top();
        st.pop();
        visit(current);
        inorder(current->right);
        inorder_stack(st);
    }
}
```

# Dos funciones auxiliares

```
inorder_gen(NodePointer node, stack<NodePointer> &st)
```

- Realiza un recorrido en inorder de node.
- Llama a `inorder_stack` pasándole `st` como parámetro.

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    inorder(node);  
    inorder_stack(st);  
}
```

Si `st` es una pila vacía, entonces `inorder_gen()` hace lo mismo que `inorder()`

# Transformando inorder\_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    inorder(node);  
    inorder_stack(st);  
}
```



# Transformando inorder\_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    inorder(node);  
    inorder_stack(st);  
}
```

```
void inorder(NodePointer node) {  
    if (node != nullptr) {  
        inorder(node→left);  
        visit(node);  
        inorder(node→right);  
    }  
}
```

# Transformando inorder\_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    if (node != nullptr) {  
        inorder(node→left);  
        visit(node);  
        inorder(node→right);  
    }  
    inorder_stack(st);  
}
```

```
void inorder(NodePointer node) {  
    if (node != nullptr) {  
        inorder(node→left);  
        visit(node);  
        inorder(node→right);  
    }  
}
```

# Transformando inorder\_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
    if (node != nullptr) {
        inorder(node->left);
        visit(node);
        inorder(node->right);
    }
    inorder_stack(st);
}
```



# Transformando inorder\_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    if (node != nullptr) {  
        inorder(node->left);  
        visit(node);  
        inorder(node->right);  
    } else {  
    }  
    inorder_stack(st);  
}
```



# Transformando inorder\_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    if (node != nullptr) {  
        inorder(node->left);  
        visit(node);  
        inorder(node->right); } st.push(node);  
        inorder_stack(st);  
    } else {  
        inorder_stack(st);  
    }  
}
```

`inorder_stack(stack<NodePointer> st)`

Desapila todos los elementos de `st`, y para cada uno de ellos:

- Visita su raíz.
- Realiza un recorrido en inorder de su hijo derecho

# Transformando inorder\_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    if (node != nullptr) {  
        inorder(node->left);  
        st.push(node);  
        inorder_stack(st);  
    } else {  
        inorder_stack(st);  
    }  
}
```

# Transformando inorder\_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    if (node != nullptr) {  
        st.push(node);  
        inorder(node→left); } inorder_gen(node→left, st);  
        inorder_stack(st);  
    } else {  
        inorder_stack(st);  
    }  
}
```

inorder\_gen(NodePointer node, stack<NodePointer> st)

- Realiza un recorrido en inorder de node.
- Llama a `inorder_stack` pasándole `st` como parámetro.

# Transformando inorder\_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    if (node != nullptr) {  
        st.push(node);  
        inorder_gen(node->left, st);  
    } else {  
        inorder_stack(st);  
    }  
}
```

*iEs recursiva final!*

# Transformando inorder\_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
    if (node != nullptr) {
        st.push(node);
        node = node->left;
        inorder_gen(node, st);
    } else {
        inorder_stack(st);
    }
}
```

# Transformando inorder\_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    if (node == nullptr) {  
        inorder_stack(st);  
    } else {  
        st.push(node);  
        node = node->left;  
        inorder_gen(node, st);  
    }  
}
```

# Transformando inorder\_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    if (node == nullptr) {  
        inorder_stack(st);  
    } else {  
        st.push(node);  
        node = node->left;  
        inorder_gen(node, st);  
    }  
}
```

```
void f(x) {  
    previo();  
    if (es_caso_base(x)) {  
        caso_base();  
    } else {  
        pre_recursivo();  
        f(x);  
    }  
}
```



```
void f(x) {  
    previo();  
    while (!es_caso_base(x)) {  
        pre_recursivo();  
        previo();  
    }  
    caso_base();  
}
```

# Transformando inorder\_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
    while (node != nullptr) {
        st.push(node);
        node = node->left;
    }
    inorder_stack(st);
}
```

```
void f(x) {
    previo();
    if (es_caso_base(x)) {
        caso_base();
    } else {
        pre_recursivo();
        f(x);
    }
}
```



```
void f(x) {
    previo();
    while (!es_caso_base(x)) {
        pre_recursivo();
        previo();
    }
    caso_base();
}
```

# Transformando inorder\_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    while (node != nullptr) {  
        st.push(node);  
        node = node->left;  
    }  
    inorder_stack(st);  
}
```

# Transformando inorder\_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    while (node != nullptr) {  
        st.push(node);  
        node = node->left;  
    }  
    inorder_stack(st);  
}
```

```
void inorder_stack(stack<NodePointer> &st) {  
    if (!st.empty()) {  
        NodePointer current = st.top();  
        st.pop();  
        visit(current);  
        inorder(current->right);  
        inorder_stack(st);  
    }  
}
```

# Transformando inorder\_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
    while (node != nullptr) {
        st.push(node);
        node = node->left;
    }
    if (!st.empty()) {
        NodePointer current = st.top();
        st.pop();
        visit(current);
        inorder(current->right);
        inorder_stack(st);
    }
}
```

```
void inorder_stack(stack<NodePointer> &st) {
    if (!st.empty()) {
        NodePointer current = st.top();
        st.pop();
        visit(current);
        inorder(current->right);
        inorder_stack(st);
    }
}
```

# Transformando inorder\_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    while (node != nullptr) {  
        st.push(node);  
        node = node->left;  
    }  
    if (!st.empty()) {  
        NodePointer current = st.top();  
        st.pop();  
        visit(current);  
        inorder(current->right); } inorder_gen(current->right, st);  
        inorder_stack(st);  
    }  
}
```

inorder\_gen(NodePointer node, stack<NodePointer> st)

- Realiza un recorrido en inorder de node.
- Llama a `inorder_stack` pasándole `st` como parámetro.

# Transformando inorder\_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    while (node != nullptr) {  
        st.push(node);  
        node = node->left;  
    }  
    if (!st.empty()) {  
        NodePointer current = st.top();  
        st.pop();  
        visit(current);  
        inorder_gen(current->right, st);  
    }  
}
```

*iEs recursiva final!*

# Transformando inorder\_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    while (node != nullptr) {  
        st.push(node);  
        node = node->left;  
    }  
    if (!st.empty()) {  
        NodePointer current = st.top();  
        st.pop();  
        visit(current);  
        node = current->right;  
        inorder_gen(node, st);  
    }  
}
```

*iEs recursiva final!*

# Transformando inorder\_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {
    while (node != nullptr) {
        st.push(node);
        node = node->left;
    }
    if (st.empty()) {

    } else {
        NodePointer current = st.top();
        st.pop();
        visit(current);
        node = current->right;
        inorder_gen(node, st);
    }
}
```

```
void f(x) {
    previo();
    if (es_caso_base(x)) {
        caso_base();
    } else {
        pre_recursivo();
        f(x);
    }
}
```

# Transformando inorder\_gen

```
void inorder_gen(NodePointer node, stack<NodePointer> &st) {  
    while (node != nullptr) {  
        st.push(node);  
        node = node->left;  
    }  
    while (!st.empty()) {  
        NodePointer current = st.top();  
        st.pop();  
        visit(current);  
        node = current->right;  
        while (node != nullptr) {  
            st.push(node);  
            node = node->left;  
        }  
    }  
}
```

iEs iterativa!

# Versión iterativa

```
stack<NodePointer> st;
NodePointer node = root;

while (node != nullptr) {
    st.push(node);
    node = node->left;
}
while (!st.empty()) {
    NodePointer current = st.top();
    st.pop();
    visit(current);
    node = current->right;
    while (node != nullptr) {
        st.push(node);
        node = node->left;
    }
}
```



# Comparación

```
stack<NodePointer> st;
NodePointer node = root;

while (node != nullptr) {
    st.push(node);
    node = node->left;
}
while (!st.empty()) {
    NodePointer current = st.top();
    st.pop();
    visit(current);
    node = current->right;
    while (node != nullptr) {
        st.push(node);
        node = node->left;
    }
}
```

```
stack<NodePointer> st;
descend_and_push(root, st);

while (!st.empty()) {
    NodePointer x = st.top();
    st.pop();
    visit(x);
    descend_and_push(x->right, st);
}
```

ESTRUCTURAS DE DATOS

TIPOS ABSTRACTOS DE DATOS ARBORESCENTES

# Iteradores en árboles

Manuel Montenegro Montes

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

# Recordatorio

```
void inorder(NodePointer &node) {  
  
    std::stack<NodePointer> st;  
    descend_and_push(node, st);  
  
    while (!st.empty()) {  
        NodePointer x = st.top();  
        visit(x->elem);  
        st.pop();  
        descend_and_push(x->right, st);  
    }  
}
```



# ¿Cómo implementar un iterador?

- Un iterador debe simular este recorrido, pero «por partes».

```
void inorder(NodePointer &node) {  
  
    std::stack<NodePointer> st;  
    descend_and_push(node, st);  
  
    while (!st.empty()) {  
        NodePointer x = st.top();  
        visit(x->elem);  
        st.pop();  
        descend_and_push(x->right, st);  
    }  
  
    for (auto it = tree.begin(); it != tree.end(); ++it) {  
        visit(*it);  
    }  
}  
  
auto it = tree.begin();  
visit(*it);  
++it;
```

# Interfaz de iteradores

```
template<class T>
class BinTree {
public:
    iterator begin();
    iterator end();

    class iterator {
public:
    T & operator*() const;
    iterator & operator++();
    bool operator==(const iterator &other);
    bool operator!=(const iterator &other);

    ...
};

};
```

# Implementación privada de iteradores

```
class iterator {  
public:  
    T & operator*() const;  
    iterator & operator++();  
    bool operator==(const iterator &other);  
    bool operator!=(const iterator &other);  
  
private:  
    iterator();  
    iterator(const NodePointer &root);  
  
    std::stack<NodePointer> st;  
};
```

```
void inorder(NodePointer &node) {  
  
    std::stack<NodePointer> st;  
    descend_and_push(node, st);  
  
    while (!st.empty()) {  
        NodePointer x = st.top();  
        visit(x->elem);  
        st.pop();  
        descend_and_push(x->right, st);  
    }  
}
```

# Implementación privada de iteradores

```
class iterator {  
public:  
    T & operator*() const;  
    iterator & operator++();  
    bool operator==(const iterator &other);  
    bool operator!=(const iterator &other);  
  
private:  
    iterator() { }  
    iterator(const NodePointer &root) {  
        BinTree::descend_and_push(root, st);  
    }  
  
    std::stack<NodePointer> st;  
};
```

```
void inorder(NodePointer &node) {  
  
    std::stack<NodePointer> st;  
    descend_and_push(node, st);  
  
    while (!st.empty()) {  
        NodePointer x = st.top();  
        visit(x->elem);  
        st.pop();  
        descend_and_push(x->right, st);  
    }  
}
```

# Implementación privada de iteradores

```
class iterator {  
public:  
    T & operator*() const {  
        assert(!st.empty());  
        return st.top()→elem;  
    }  
    iterator & operator++();  
    bool operator==(const iterator &other);  
    bool operator!=(const iterator &other);  
  
private:  
    iterator();  
    iterator(const NodePointer &root);  
  
    std::stack<NodePointer> st;  
};
```

```
void inorder(NodePointer &node) {  
  
    std::stack<NodePointer> st;  
    descend_and_push(node, st);  
  
    while (!st.empty()) {  
        NodePointer x = st.top();  
        visit(x→elem);  
        st.pop();  
        descend_and_push(x→right, st);  
    }  
}
```

# Implementación privada de iteradores

```
class iterator {  
public:  
    T & operator*() const;  
  
    iterator & operator++() {  
        assert(!st.empty());  
        NodePointer top = st.top();  
        st.pop();  
        BinTree::descend_and_push(top->right, st);  
        return *this;  
    }  
  
    bool operator==(const iterator &other);  
    bool operator!=(const iterator &other);  
  
private:  
    iterator();  
    iterator(const NodePointer &root);  
  
    std::stack<NodePointer> st;  
};
```

```
void inorder(NodePointer &node) {  
  
    std::stack<NodePointer> st;  
    descend_and_push(node, st);  
  
    while (!st.empty()) {  
        NodePointer x = st.top();  
        visit(x->elem);  
        st.pop();  
        descend_and_push(x->right, st);  
    }  
}
```

# Creación de iteradores

```
template<class T>
class BinTree {
public:

    iterator begin() {
        return iterator(root_node);
    }

    iterator end() {
        return iterator();
    }

    ...

};
```



# Ejemplo

```
int main() {
    BinTree<int> tree {{{ 9 }, 4, { 5 }}, 7, {{ 10 }, 4, { 6 }}};

    for (auto it = tree.begin(); it != tree.end(); ++it) {
        cout << *it << " ";
    }

    return 0;
}
```

9 4 5 7 10 4 6

# Ejemplo

```
int main() {
    BinTree<int> tree {{{ 9 }, 4, { 5 }}, 7, {{ 10 }, 4, { 6 }}};

    for (int x: tree) {
        cout << x << " ";
    }

    return 0;
}
```

9 4 5 7 10 4 6

# Posibles extensiones

- Iteradores constantes: `cbegin()`, `cend()`, etc.
- Diferencia entre postincremento (`it++`) y preincremento (`++it`).
- Aplicación a SetTree y MapTree.



ESTRUCTURAS DE DATOS

DICCIONARIOS

# EL TAD Diccionario

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

# Motivación

- Leer un texto de entrada e imprimir el número de veces que aparece cada palabra.

*David tomó la llave para entregársela a Laura. Esta última, no obstante, declinó hacer uso de la llave mientras que no fuera absolutamente necesario.*

David 1  
tomó 1  
la 2  
llave 2  
para 1  
...

- ¿Cómo almacenamos las palabras que nos encontramos?

# Motivación

- Tabla en la que almacenamos el número de veces que aparece cada palabra encontrada hasta el momento.
- Con cada palabra recibida:
  - Si existe una entrada en la tabla con esa palabra, incrementamos su contador.
  - Si no, insertamos una nueva entrada con esa palabra y con su contador a 1.

Palabra	Contador
“David”	1
“tomó”	1
“la”	2
“llave”	2
...	...

# ¿Qué es un diccionario?

- Un tipo abstracto de datos que almacena un conjunto de pares.
- A cada par se le llama **entrada**.
- A la primera componente de cada par se le denomina **clave**.
- A la segunda componente se le denomina **valor asociado** a esa clave.
- No existen dos pares con la misma clave.

Palabra	Contador
“David”	1
“tomó”	1
“la”	2
“llave”	2
...	...

**Claves**      **Valores**

# Terminología

*diccionarios*

*maps*

*tablas*

*associative arrays*

*arrays asociativos*

*dictionaries*

*symbol tables*

Palabra	Contador
“David”	1
“tomó”	1
“la”	2
“llave”	2
...	...

# Modelo conceptual de diccionarios

- Sean:
  - $K$  – conjunto de claves
  - $V$  – conjunto de valores
- Un diccionario  $M$  es un conjunto de pares  $(k, v)$ , donde  $k \in K$ ,  $v \in V$ .
- No existen pares  $(k, v), (k, v') \in M$  tales que  $v \neq v'$ .

Palabra	Contador
“David”	1
“tomó”	1
“la”	2
“llave”	2
...	...

$$M = \{("David", 1), ("tomó", 1), ("la", 2), \dots\}$$

# Operaciones en el TAD Diccionario

- Constructoras:
  - Crear un diccionario vacío: ***create\_empty***
- Mutadoras:
  - Añadir una entrada al diccionario: ***insert***
  - Eliminar una entrada del diccionario: ***erase***
- Observadoras:
  - Saber si existe una entrada con una clave determinada: ***contains***
  - Saber el valor asociado con una clave: ***at***
  - Saber si el diccionario está vacío: ***empty***
  - Saber el número de entradas del diccionario: ***size***

# Operaciones constructoras y mutadoras

{ true }

**create\_empty()** → (M: Map)

{ M = Ø }

{ true }

**insert**(k: Key, v: Value, M: Map)

$$\left\{ M = \begin{cases} old(M) & \text{si } \exists v' : (k, v') \in old(M) \\ old(M) \cup \{(k, v)\} & \text{en otro caso} \end{cases} \right\}$$

{ true }

**erase**(k: Key, M: Map)

$$\left\{ M = \{ (k', v') \in old(M) \mid k' \neq k \} \right\}$$

# Operaciones observadoras

$\{ \text{ true } \}$

**contains**( $k: \text{Key}$ ,  $M: \text{Map}$ )  $\rightarrow (b: \text{bool})$

$\{ b \Leftrightarrow \exists v. (k, v) \in M \ }$

$\{ \exists v'. (k, v') \in M \ }$

**at**( $k: \text{Key}$ ,  $M: \text{Map}$ )  $\rightarrow (v: \text{value})$

$\{ (k, v) \in M \ }$

$\{ \text{ true } \}$

**empty**( $M: \text{Map}$ )  $\rightarrow (b: \text{bool})$

$\{ b \Leftrightarrow M = \emptyset \ }$

$\{ \text{ true } \}$

**size**( $M: \text{Map}$ )  $\rightarrow (n: \text{int})$

$\{ n = |M| \ }$

# Interfaz en C++

```
template <typename K, typename V>
class map {
public:
    map();
    map(const map &other);
    ~map();

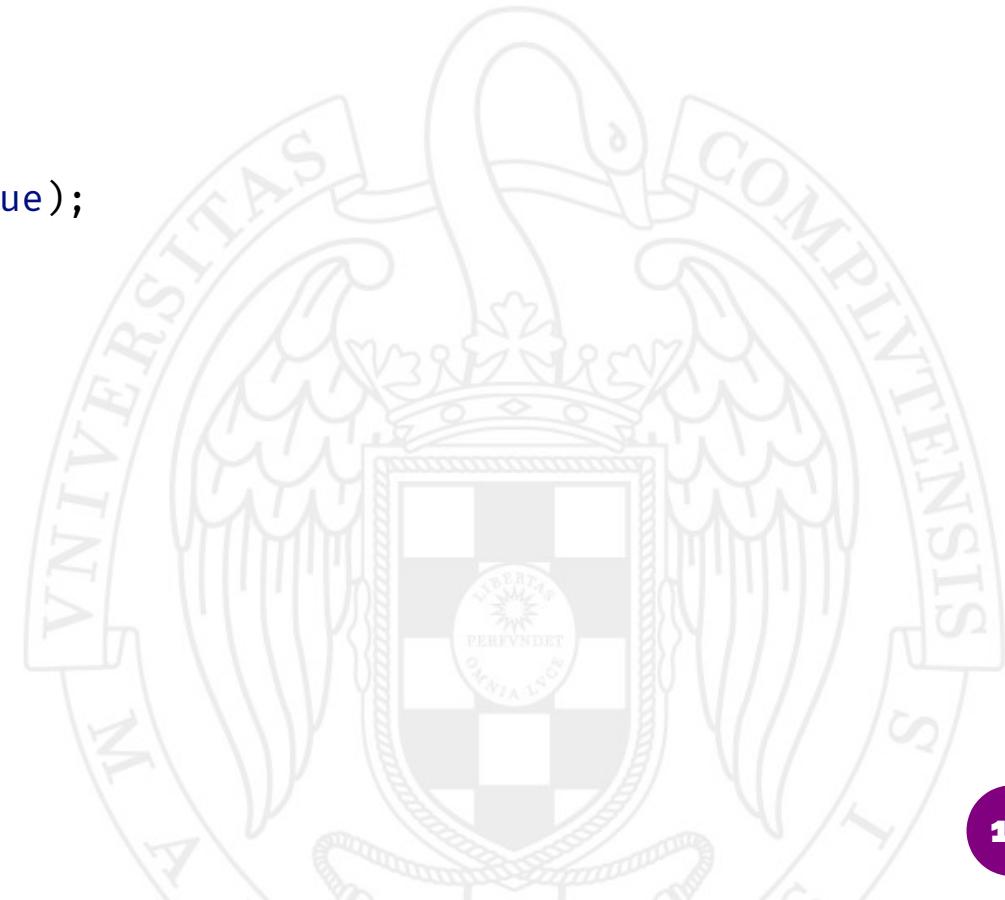
    void insert(const K &key, const V &value);
    void erase(const K &key);

    bool contains(const K &key) const;

    const V & at(const K &key) const;
    V & at(const K &key);

    int size() const;
    bool empty() const;

private:
    // ...
};
```



# Interfaz en C++

```
template <typename K, typename V>
class map {
public:
    map();
    map(const map &other);
    ~map();

    void insert(const map_entry &entry);
    void erase(const K &key);

    bool contains(const K &key) const;

    const V & at(const K &key) const;
    V & at(const K &key);

    int size() const;
    bool empty() const;

private:
    // ...
};
```

```
struct map_entry {
    K key;
    V value;
};
```



# Ejemplo

```
map<string, int> personas;
```

personas =  $\emptyset$

```
personas.insert({ "Aarón", 42 });
personas.insert({ "Estela", 41 });
```

personas = {("Aarón", 42), ("Estela", 41) }

```
cout << personas.contains("Aarón") << endl;
cout << personas.at("Aarón") << endl;
```

true  
42

```
personas.insert({ "Carlos", 31 });
```

personas = {("Aarón", 42), ("Carlos", 31), ("Estela", 41) }

```
personas.erase("Estela");
```

personas = {("Aarón", 42), ("Carlos", 31) }

```
personas.at("Aarón") = 43;
```

personas = {("Aarón", 43), ("Carlos", 31) }

# Ejemplo

```
string palabra;
map<string, int> dicc;

cin >> palabra;

while (!cin.eof()) {
    if (dicc.contains(palabra)) {
        dicc.at(palabra)++;
    } else {
        dicc.insert({palabra, 1});
    }
    cin >> palabra;
}
```



# Dos implementaciones

- Mediante **árboles binarios de búsqueda** (MapTree)
- Mediante **tablas hash** (MapHash)



ESTRUCTURAS DE DATOS

DICCIONARIOS

# **Diccionarios mediante árboles binarios de búsqueda**

Manuel Montenegro Montes

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

# Operaciones en el TAD Diccionario

- Constructoras:
  - Crear un diccionario vacío: ***create\_empty***
- Mutadoras:
  - Añadir una entrada al diccionario: ***insert***
  - Eliminar una entrada del diccionario: ***erase***
- Observadoras:
  - Saber si existe una entrada con una clave determinada: ***contains***
  - Saber el valor asociado con una clave: ***at***
  - Saber si el diccionario está vacío: ***empty***
  - Saber el número de entradas del diccionario: ***size***

# Dos implementaciones

- Mediante **árboles binarios de búsqueda** (MapTree) ← **Este video**
- Mediante **tablas hash** (MapTable)



# Interfaz de MapTree

```
template <typename K, typename V>
class MapTree {
public:
    MapTree();
    MapTree(const MapTree &other);
    ~MapTree();

    void insert(const MapEntry &entry);
    void erase(const K &key);

    bool contains(const K &key) const;
    const V & at(const K &key) const;
    V & at(const K &key);

    int size() const;
    bool empty() const;

private:
    // ...
};
```

```
struct MapEntry {
    K key;
    V value;

    MapEntry(K key, V value);
    MapEntry(K key);
};
```

# Representación privada de MapTree

```
template <typename K, typename V>
class MapTree {
...
private:
    struct Node {
        MapEntry entry;
        Node *left, *right;

        Node(Node *left, const MapEntry &entry, Node *right);
    };

    Node *root_node;
    int num_elems;

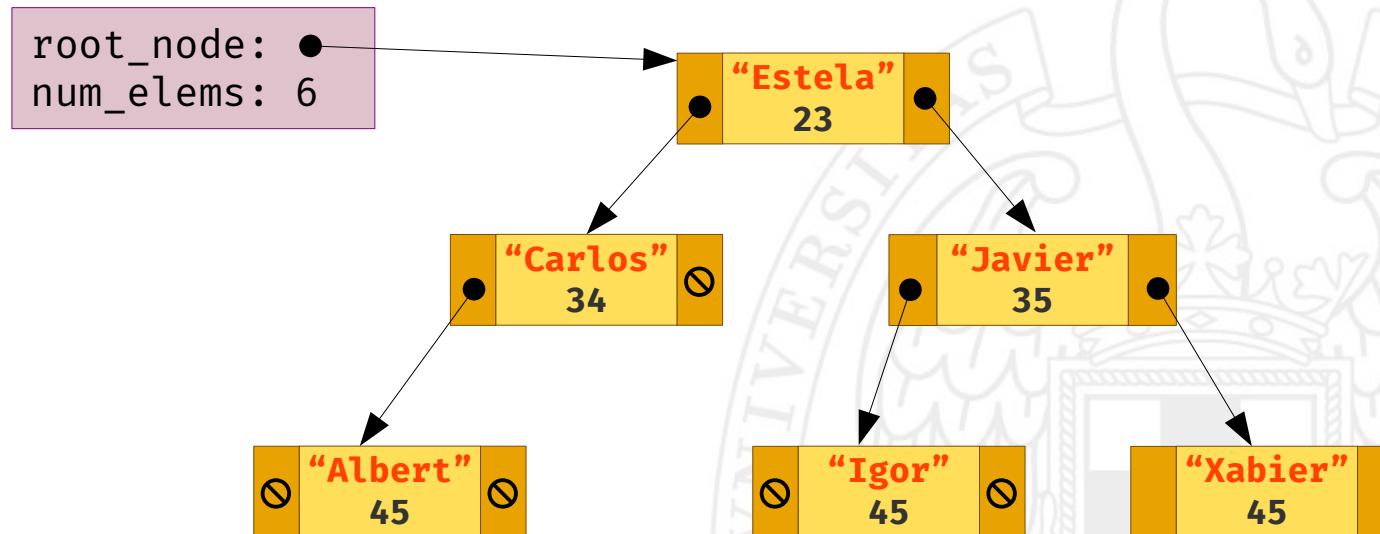
    // métodos auxiliares privados
    // ...
};
```

```
struct MapEntry {
    K key;
    V value;

    MapEntry(K key, V value);
    MapEntry(K key);
};
```

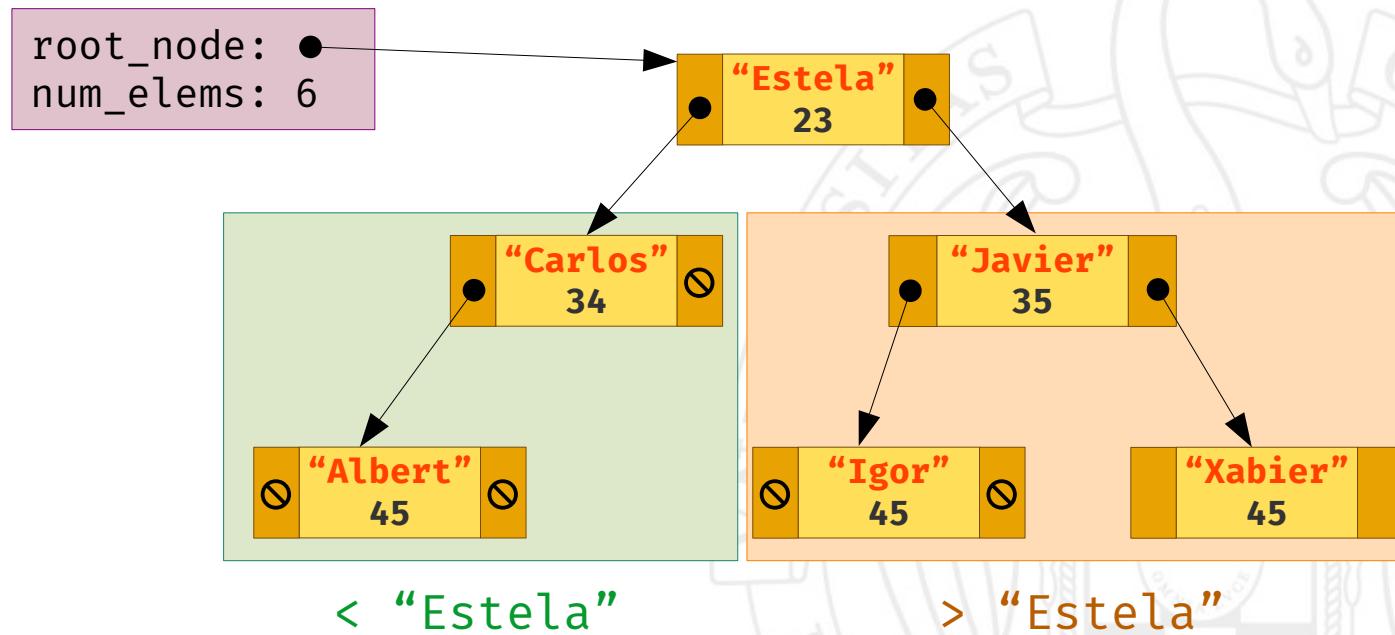
# Representación de un MapTree

{("Carlos", 34), ("Estela", 23), ("Xabier", 45), ("Igor", 45), ("Javier", 35), ("Albert", 45) }



# Representación de un MapTree

- El orden de los elementos en el árbol binario de búsqueda viene determinado por el orden de las claves.



# Métodos auxiliares

```
template <typename K, typename V>
class MapTree {
    ...
private:
    ...
    static std::pair<Node *, bool> insert(Node *root, const MapEntry &elem);
    static Node * search(Node *root, const K &key);
    static std::pair<Node *, bool> erase(Node *root, const K &key);
};
```

- Iguales que los utilizados en ABBs.
- Diferencia: se realizan comparaciones entre **las claves**.

# Métodos auxiliares

```
template <typename K, typename V>
class MapTree {
    ...
private:
    ...

    static Node * search(Node *root, const K &key) {
        if (root == nullptr) {
            return nullptr;
        } else if (key < root->entry.key) {
            return search(root->left, key);
        } else if (root->entry.key < key) {
            return search(root->right, key);
        } else {
            return root;
        }
    }
};
```



# Métodos contains() y at()

```
template <typename K, typename V>
class MapTree {
public:
    ...
    bool contains(const K &key) const {
        return search(root_node, key) != nullptr;
    }

    const V & at(const K &key) const {
        Node *result = search(root_node, key);
        assert (result != nullptr);
        return result->entry.value;
    }

    V & at(const K &key) {
        Node *result = search(root_node, key);
        assert (result != nullptr);
        return result->entry.value;
    }
};
```



# Búsqueda e inserción mediante [ ]

# Motivación

- Muchas veces encontramos código como este:

```
if (!dicc.contains(k)) {  
    words.insert({k, 1});  
} else {  
    words.at(k) = ...;  
}
```

- No es equivalente a:

```
if (!dicc.contains(k)) {  
    words.at(k) = 1; ←  
} else {  
    words.at(k) = ...;  
}
```

Error: at() exige que la clave se encuentre en el diccionario

# Motivación

Definimos una operación alternativa a `at()`, llamada `operator[]`.

`dicc.at(key)`

- Devuelve una referencia al valor asociado con la clave key.
- Si key no se encuentra, se produce un error.

`dicc[key]`

- Devuelve una referencia al valor asociado con la clave key.
- Si key no se encuentra, se añade una nueva entrada a dicc que asocia key con un valor por defecto.

# Nueva operación auxiliar

```
template <typename K, typename V>
class MapTree {
    ...
private:
    ...
    static std::tuple<bool, Node *, Node *> search_or_insert(Node *root,
                                                               const K &key) {
        if (root == nullptr) {
            Node *new_node = new Node(nullptr, {key}, nullptr);
            return {true, new_node, new_node};
        } else if (key < root->entry.key) {
            auto [inserted, new_root, found_node] = search_or_insert(root->left, key);
            root->left = new_root;
            return {inserted, root, found_node};
        } else if (root->entry.key < key) {
            auto [inserted, new_root, found_node] =
                search_or_insert(root->right, key);
            root->right = new_root;
            return {inserted, root, found_node};
        } else {
            return {false, root, root};
        }
    }
}
```

# Nueva operación auxiliar

```
template <typename K, typename V>
class MapTree {
    ...
private:
    ...
    static std::tuple<bool, Node *, Node *> search_or_insert(Node *root,
                                                               const K &key) {
        if (root == nullptr) {
            Node *new_node = new Node(nullptr, {key}, nullptr);
            return {true, new_node, new_node};
        } else if (key < root->entry.key) {
            auto [inserted, new_root, found_node] = search_or_insert(root->left, key);
            root->left = new_root;
            return {inserted, root, found_node};
        } else if (root->entry.key < key) {
            auto [inserted, new_root, found_node] =
                search_or_insert(root->right, key);
            root->right = new_root;
            return {inserted, root, found_node};
        } else {
            return {false, root, root};
        }
    }
}
```

```
struct MapEntry {
    K key;
    V value;

    MapEntry(K key, V value);
    MapEntry(K key);
};
```

# Nueva operación auxiliar

```
template <typename K, typename V>
class MapTree {
    ...
private:
    ...
    static std::tuple<bool, Node *, Node *> search_or_insert(Node *root,
                                                               const K &key) {
        if (root == nullptr) {
            Node *new_node = new Node(nullptr, {key}, nullptr);
            return {true, new_node, new_node};
        } else if (key < root->entry.key) {
            auto [inserted, new_root, found_node] = search_or_insert(root->left, key);
            root->left = new_root;
            return {inserted, root, found_node};
        } else if (root->entry.key < key) {
            auto [inserted, new_root, found_node] = search_or_insert(root->right, key);
            root->right = new_root;
            return {inserted, root, found_node};
        } else {
            return {false, root, root};
        }
    }
}
```

# Nueva operación auxiliar

```
template <typename K, typename V>
class MapTree {
    ...
private:
    ...
    static std::tuple<bool, Node *, Node *> search_or_insert(Node *root,
                                                               const K &key) {
        if (root == nullptr) {
            Node *new_node = new Node(nullptr, {key}, nullptr);
            return {true, new_node, new_node};
        } else if (key < root->entry.key) {
            auto [inserted, new_root, found_node] = search_or_insert(root->left, key);
            root->left = new_root;
            return {inserted, root, found_node};
        } else if (root->entry.key < key) {
            auto [inserted, new_root, found_node] = search_or_insert(root->right, key);
            root->right = new_root;
            return {inserted, root, found_node};
        } else {
            return {false, root, root};
        }
    }
}
```

# Implementación de operator[]

```
template <typename K, typename V>
class MapTree {
public:
    ...
    V &operator[](const K &key) {
        auto [inserted, new_root, found_node] = search_or_insert(root_node, key);
        this->root_node = new_root;
        if (inserted) { num_elems++; }
        return found_node->entry.value;
    }
};
```

# Resultado

```
if (!dicc.contains(k)) {  
    words.insert({k, 1});  
} else {  
    words.at(k) = ...;  
}
```



```
if (!dicc.contains(k)) {  
    words.at(k) = 1; X  
} else {  
    words.at(k) = ...;  
}
```

# Resultado

```
if (!dicc.contains(k)) {  
    words.insert({k, 1});  
} else {  
    words.at(k) = ...;  
}
```



```
if (!dicc.contains(k)) {  
    words[k] = 1;  
} else {  
    words[k] = ...;  
}
```



# Coste de las operaciones

Operación	Árbol equilibrado	Árbol no equilibrado
<i>constructor</i>	$O(1)$	$O(1)$
<i>empty</i>	$O(1)$	$O(1)$
<i>size</i>	$O(1)$	$O(1)$
<i>contains</i>	$O(\log n)$	$O(n)$
<i>at</i>	$O(\log n)$	$O(n)$
<i>operator[]</i>	$O(\log n)$	$O(n)$
<i>insert</i>	$O(\log n)$	$O(n)$
<i>erase</i>	$O(\log n)$	$O(n)$

$n$  = número de entradas en el diccionario

ESTRUCTURAS DE DATOS

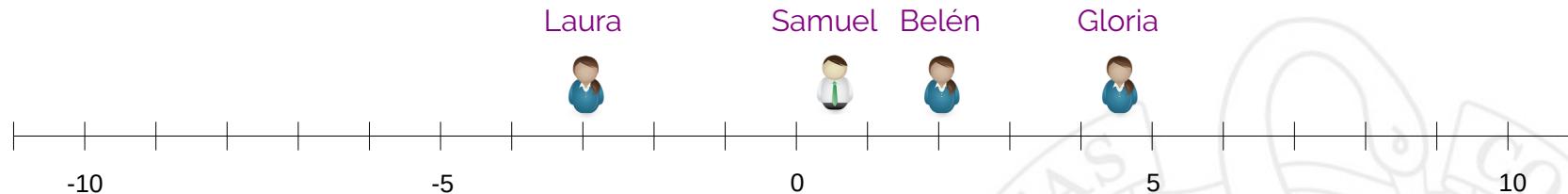
DICCIONARIOS

# Relaciones de orden en ABBs

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

# Motivación

- Queremos simular el movimiento de varias personas en una calle recta:

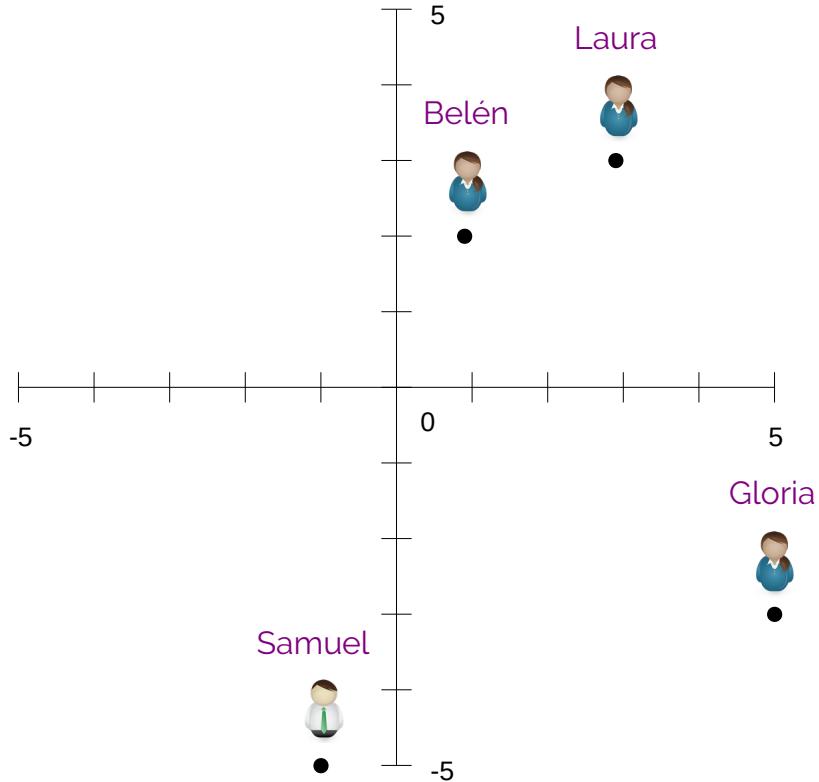


- ¿Cómo represento el estado actual?

```
MapTree<double, string> posiciones;  
posiciones.insert({-3, "Laura"});  
posiciones.insert({4.5, "Gloria"});  
posiciones.insert({2, "Belén"});  
posiciones.insert({0.5, "Samuel"});
```

# Motivación

- ¿Hacemos lo mismo, pero ahora en un plano?



```
struct Coords {  
    double x;  
    double y;  
};
```

```
MapTree<Coords, string> posiciones;  
posiciones.insert({{3, 3}, "Laura"});  
posiciones.insert({{5, -3}, "Gloria"});  
posiciones.insert({{1, 2}, "Belén"});  
posiciones.insert({{-1, -5}, "Samuel"});
```

# ¿Qué ha pasado?

- Obtenemos el siguiente error:

```
map_tree.h:127:30: error: no match for 'operator<' (operand types are 'const Coords'  
and 'Coords')
```

```
127 |         } else if (entry.key < root->entry.key) {  
|
```



# ¿Qué ha pasado?

- A la hora de insertar nodos de un árbol binario de búsqueda, comparamos la clave que queremos insertar con algunos de los nodos del árbol:
  - *Si clave < nodo.clave, insertar en el hijo izquierdo.*
  - *Si nodo.clave < clave, insertar en el hijo derecho.*
- Utilizamos el operador `<` para comparar los nodos.
- ¡Este operador no está definido para el tipo Coords!

# Solución 1: implementar <

# Solución 1

```
struct Coords {  
    double x;  
    double y;  
};  
  
bool operator<(const Coords &p1,  
                  const Coords &p2) {  
    return p1.x < p2.x  
        || p1.x = p2.x && p1.y < p2.y;  
}
```

- Orden **lexicográfico**:

Compara las coordenadas x.  
En caso de igualdad, compara  
las coordenadas y.

# ¿Sirve cualquier definición de `<`?

- Tiene que cumplir las siguientes propiedades:
  - **Antirreflexiva:** Nunca se cumple  $a < a$  para ningún  $a$ .
  - **Asimétrica:** Si  $a < b$ , entonces no se cumple  $b < a$ .
  - **Transitiva:** Si  $a < b$  y  $b < c$ , entonces  $a < c$ .

El compilador no comprueba que el operador `<` que definamos cumpla estas tres propiedades, pero si no las cumple, el ABB puede comportarse de manera inconsistente.

- La definición que escojamos determina el orden en el que iteremos sobre las entradas de un árbol.

# Problemas

- Si definimos el operador  $<$  para un tipo de datos, este se aplica a todos los MapTree que utilicen ese tipo como clave.
- ¿Y si quiero utilizar una relación de orden para un MapTree, y otra relación distinta para otro MapTree?



# Solución 2: parametrizar MapTree

# Parametrizar MapTree

- Indicamos cómo comparar las claves mediante un **objeto función**.
- Consiste en añadir un tercer parámetro de tipo a MapTree:

MapTree<**K**, **V**, **Comparator**>

Tipo de las claves

Tipo de los valores

Tipo del objeto función  
que compara las claves

# Parametrizar MapTree

- Indicamos cómo comparar las claves mediante un **objeto función**.

- Consiste en añadir un tercer parámetro de tipo a MapTree:

MapTree<K, V, Comparator>

- La clase **Comparator** debe sobrecargar el operador () .

- La sobrecarga recibe dos parámetros.

- Devuelve true si el primero es estrictamente menor que el segundo.

# Objetos función

```
struct OrdenLexicografico {  
    bool operator()(const Coords &p1, const Coords &p2) const {  
        return p1.x < p2.x || p1.x = p2.x && p1.y < p2.y;  
    }  
};
```

```
MapTree<Coords, string, OrdenLexicografico> dicc;  
dicc.insert({{3, 3}, "Laura"});  
dicc.insert({{5, -3}, "Gloria"});  
dicc.insert({{1, 2}, "Belén"});  
dicc.insert({{-1, -5}, "Samuel"});
```

# ¿Y si quiero utilizar <?

- Utilizar solamente dos parámetros: tipo de las claves y valores.

```
MapTree<Coords, string> dicc;  
dicc.insert({{3, 3}, "Laura"});  
dicc.insert({{5, -3}, "Gloria"});  
dicc.insert({{1, 2}, "Belén"});  
dicc.insert({{-1, -5}, "Samuel"});
```



# Implementación

```
template <typename K, typename V, typename ComparatorFunction = std::less<K>>
class MapTree {
    ...
private:
    struct Node { ... };

    Node *root_node;
    int num_elems;
    ComparatorFunction less_than;

    ...
}
```



# Implementación: antes

```
template <typename K, typename V, typename ComparatorFunction = std::less<K>>
class MapTree {
    ...
private:
    ...
    ComparatorFunction less_than;

    static Node * search(Node *root, const K &key) {
        if (root == nullptr) {
            return nullptr;
        } else if (key < root->entry.key) {
            return search(root->left, key);
        } else if (root->entry.key < key) {
            return search(root->right, key);
        } else {
            return root;
        }
    }
};
```



# Implementación: después

```
template <typename K, typename V, typename ComparatorFunction = std::less<K>>
class MapTree {
    ...
private:
    ...
    ComparatorFunction less_than;

    static Node * search(Node *root, const K &key) {
        if (root == nullptr) {
            return nullptr;
        } else if (less_than(key, root->entry.key)) {
            return search(root->left, key);
        } else if (less_than(root->entry.key, key)) {
            return search(root->right, key);
        } else {
            return root;
        }
    }
};
```

ESTRUCTURAS DE DATOS

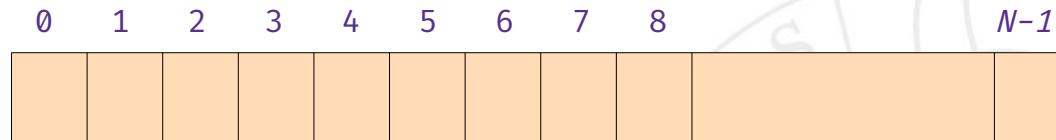
DICCIONARIOS

# Introducción a las tablas *hash*

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

# ¿Qué es una tabla hash?

- Es una estructura de datos que permite implementar colecciones de datos no secuenciales: diccionarios, conjuntos, etc.
- Utiliza un vector de tamaño  $N$  (número primo).



- Se basa en una **función hash**  $h$  que devuelve, para una clave, un número entero.

$$h: K \rightarrow \mathbb{Z}$$

- Este número determina la posición del vector en la que se almacena la clave.

# Ejemplos de funciones hash

- Para números enteros o naturales, la identidad es suficiente:

$$h(x) = x$$

- Para cadenas, suele utilizarse la siguiente fórmula:

$$h(s) = s[0] \cdot p^0 + s[1] \cdot p^1 + s[2] \cdot p^2 + \dots + s[n-1] \cdot p^{n-1}$$

donde:

- $s$  es una cadena de longitud  $n$ .
- $s[i]$  es el código asociado al carácter  $i$ -ésimo.
- $p$  es un número primo (normalmente  $p = 31 \circ p = 53 \circ p = 131$ )

# ¿Cómo se implementa esta función hash?

- Necesitamos una función hash que se comporte de forma diferente en función de si recibe un entero, una cadena, etc.
- También queremos poder extenderla para tratar nuevos tipos de claves.
- **Solución:** objetos función.

```
template<class K>
class std::hash {
public:
    int operator()(const K &key) const;
};
```



# ¿Cómo se implementa esta función *hash*?

- Las plantillas de C++ pueden particularizarse para tipos de datos concretos.
- Por ejemplo, implementación para el caso K = int.

```
template<typename T>
class std::hash<int> {
public:
    int operator()(const int &key) const {
        return key;
    }
};
```

# ¿Cómo se implementa esta función *hash*?

- Las plantillas de C++ pueden particularizarse para tipos de datos concretos.
- Por ejemplo, implementación para el caso K = string.

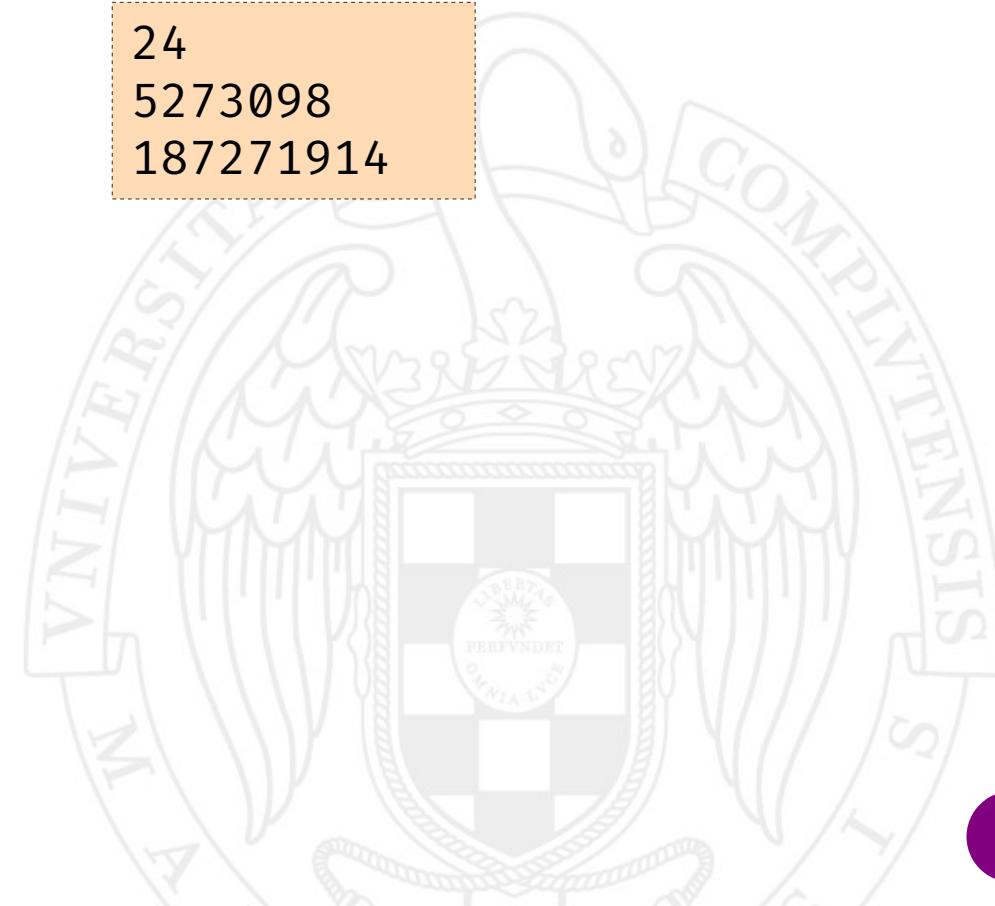
```
template<typename T>
class std::hash<std::string> {
public:
    int operator()(const std::string &key) const {
        const int POWER = 37;
        int result = 0;
        for (int i = key.length() - 1; i >= 0; i--) {
            result = result * POWER + key[i];
        }
        return result;
    }
};
```

# Ejemplo

```
hash<int> h_int;
hash<std::string> h_str;

std::cout << h_int(24) << std::endl;
std::cout << h_str("Pepe") << std::endl;
std::cout << h_str("Maria") << std::endl;
```

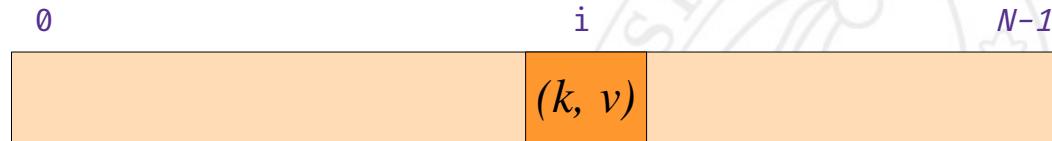
24  
5273098  
187271914



# ¿Cómo funcionan las tablas hash?

Supongamos que queremos **insertar** una entrada  $(k, v)$  en un diccionario implementado mediante una tabla *hash* con  $N$  posiciones.

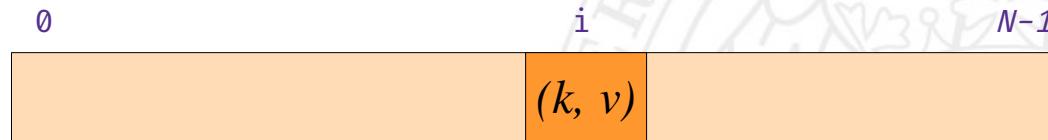
- 1) Calculamos  $i := h(k) \bmod N$ .
- 2) Insertamos el par  $(k, v)$  en la posición  $i$ -ésima del vector.



# ¿Cómo funcionan las tablas hash?

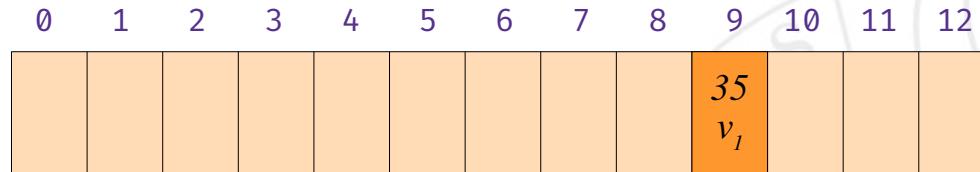
Supongamos que queremos **buscar** la entrada con clave  $k$  en un diccionario implementado mediante una tabla *hash* con  $N$  posiciones.

- 1) Calculamos  $i := h(k) \bmod N$ .
- 2) Obtenemos el par  $(k, v)$  de la posición  $i$ -ésima del vector.
- 3) Devolvemos  $v$ .



# Ejemplo

- Con  $N = 13$ , queremos insertar la entrada  $(35, v_1)$ .
- Hacemos  $h(35) \bmod 13 = 35 \bmod 13 = 9$ .



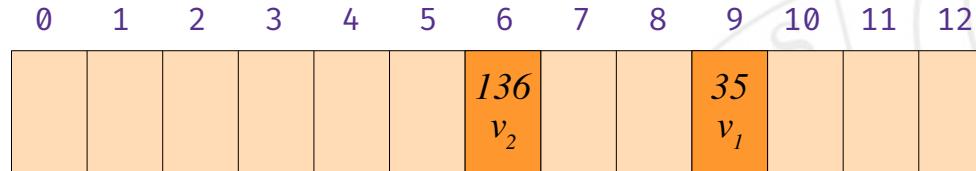
# Ejemplo

- Ahora insertamos la entrada  $(136, v_2)$
- Hacemos  $h(136) \bmod 13 = 136 \bmod 13 = 6$ .

0	1	2	3	4	5	6	7	8	9	10	11	12
						136 $v_2$			35 $v_1$			

# Ejemplo

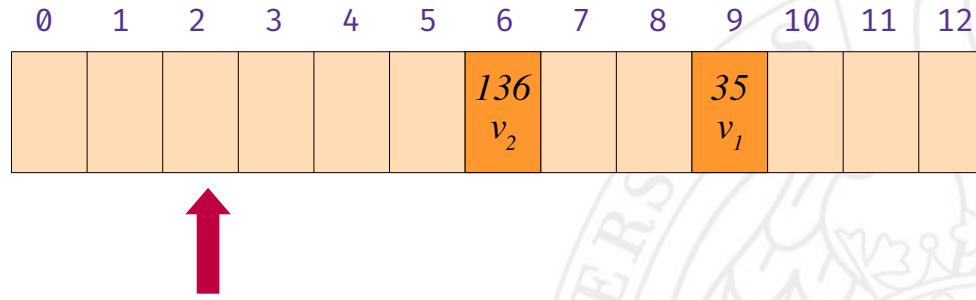
- Buscamos la entrada con clave 35.
- $h(35) \bmod 13 = 35 \bmod 13 = 9$ .



Clave  
encontrada

# Ejemplo

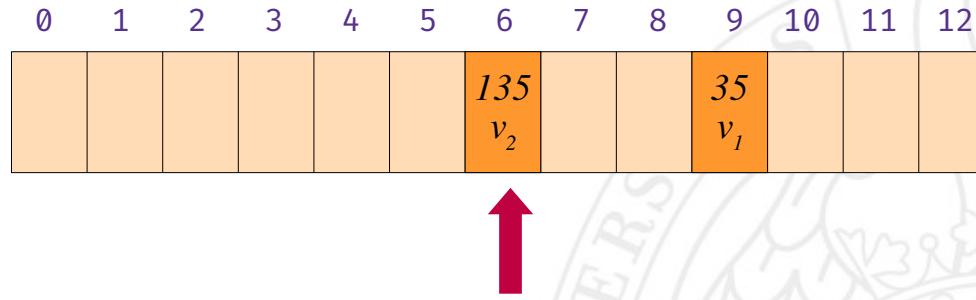
- Buscamos la entrada con clave 41.
- $h(41) \bmod 13 = 41 \bmod 13 = 2$ .



Clave **no**  
encontrada

# Ejemplo

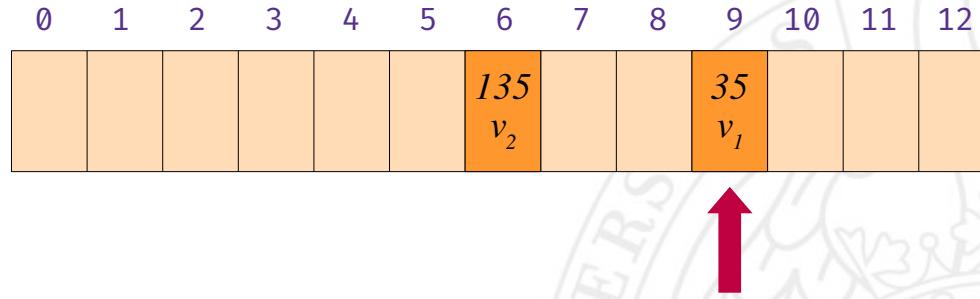
- Buscamos la entrada con clave 149.
- $h(149) \bmod 13 = 149 \bmod 13 = 6$ .



Clave **no**  
encontrada

# Ejemplo

- Insertamos la entrada  $(61, v_3)$ .
- $h(61) \bmod 13 = 61 \bmod 13 = 9.$

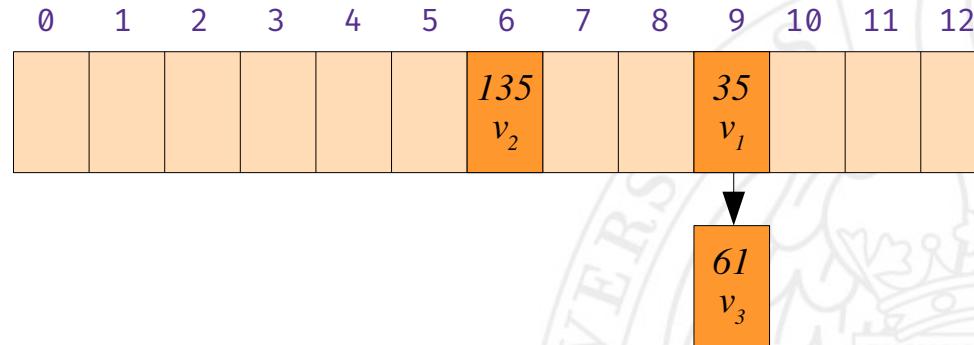


# Colisiones

- Cuando la función *hash* envía dos claves  $k_1$  y  $k_2$  a la misma posición del vector se produce una **colisión**.
- Esto sucede cuando  $h(k_1) \bmod N = h(k_2) \bmod N$ .
- Una buena función *hash* debe distribuir de la manera más uniformemente posible las claves entre las distintas posiciones del vector, para que la probabilidad de colisiones sea baja.
- Pero, tarde o temprano, tendremos colisiones.

# ¿Cómo solucionamos las colisiones?

- **Tablas hash abiertas:** Cada posición del vector contiene una **lista** de todas las claves destinadas ahí.



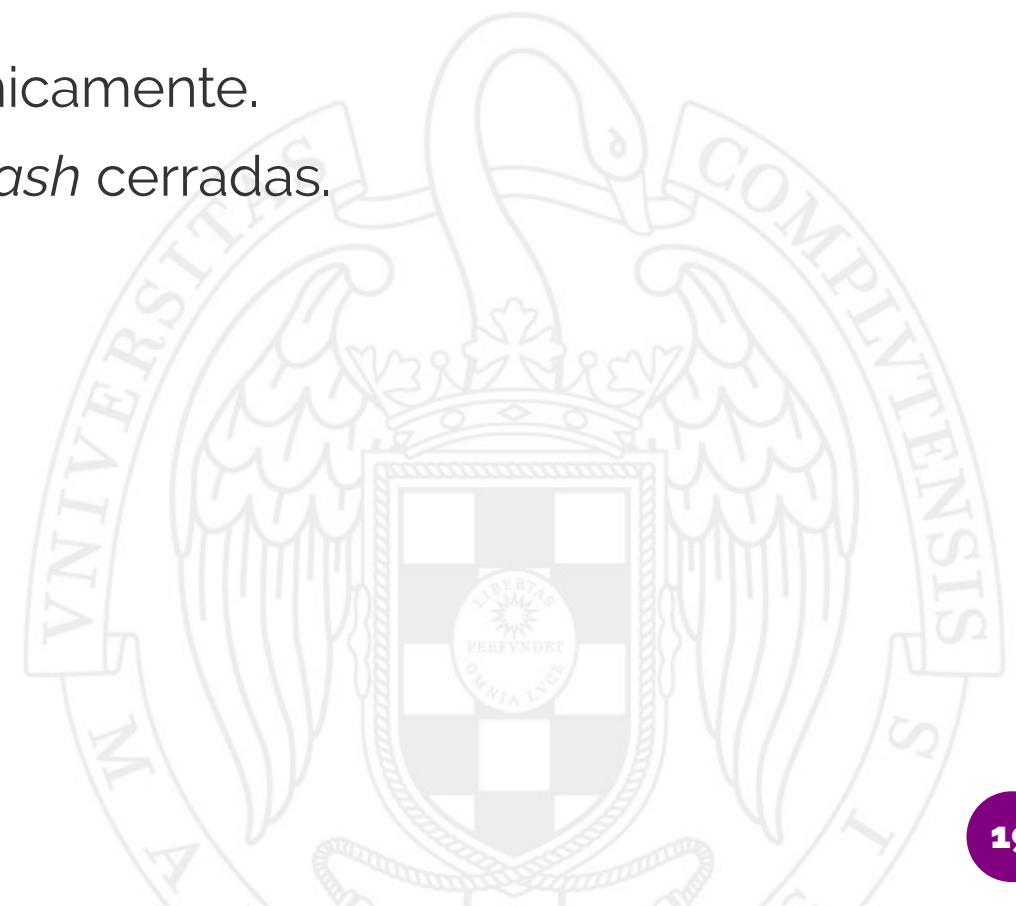
# ¿Cómo solucionamos las colisiones?

- **Tablas hash cerradas:** reubican el par que queremos insertar en una posición alternativa del vector.

0	1	2	3	4	5	6	7	8	9	10	11	12
						135 $v_2$			35 $v_1$	61 $v_3$		

# Implementaciones

- TAD Diccionario utilizando tablas *hash* abiertas.
  - Tablas de tamaño fijo.
  - Tablas redimensionables dinámicamente.
- TAD Diccionario utilizando tablas *hash* cerradas.



ESTRUCTURAS DE DATOS

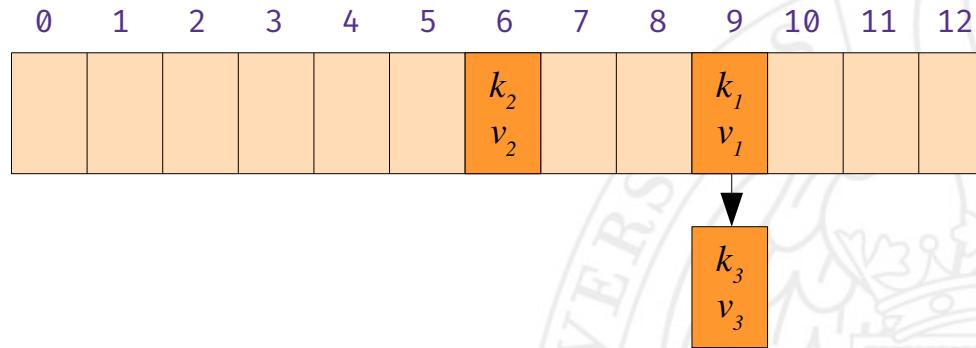
DICCIONARIOS

# Tablas *hash* abiertas

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

# Objetivo

- Implementar el TAD Diccionario mediante una tabla *hash* abierta.



# Recordatorio: TAD Diccionario

- Constructoras:
  - Crear un diccionario vacío: ***create\_empty***
- Mutadoras:
  - Añadir una entrada al diccionario: ***insert***
  - Eliminar una entrada del diccionario: ***erase***
- Observadoras:
  - Saber si existe una entrada con una clave determinada: ***contains***
  - Saber el valor asociado con una clave: ***at***
  - Saber si el diccionario está vacío: ***empty***
  - Saber el número de entradas del diccionario: ***size***

# Clase MapHash: interfaz pública

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:
    MapHash();
    MapHash(const MapHash &other);
    ~MapHash();

    void insert(const MapEntry &entry);
    void erase(const K &key);

    bool contains(const K &key) const;
    const V & at(const K &key) const;
    V & at(const K &key);
    V & operator[](const K &key);

    int size() const;
    bool empty() const;

    MapHash & operator=(const MapHash &other);
    void display(std::ostream &out) const;

private:
    // ...
};
```

```
struct MapEntry {
    K key;
    V value;

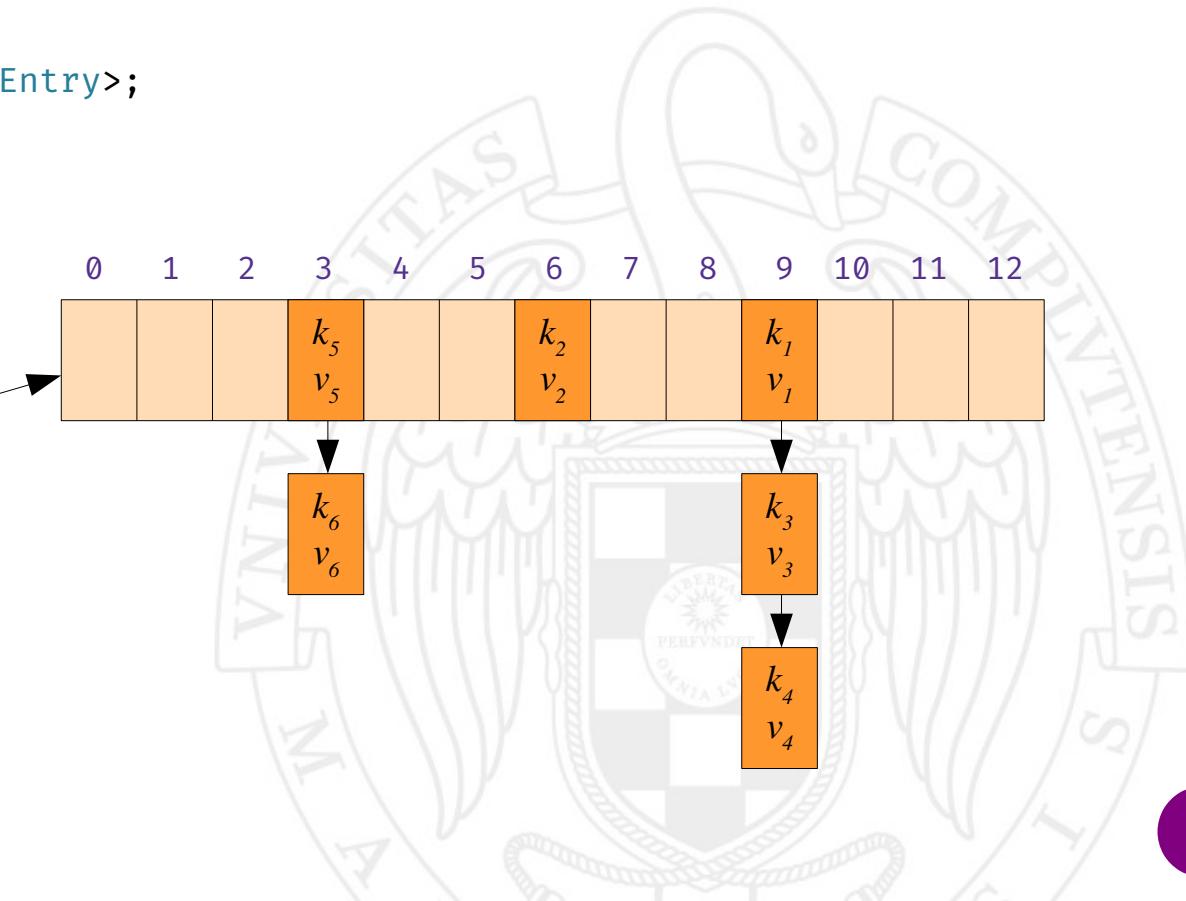
    MapEntry(K key, V value);
    MapEntry(K key);
};
```

# Clase MapHash: representación privada

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {

private:
    using List = std::forward_list<MapEntry>;
    List *buckets;
    int num_elems;
    Hash hash;
};
```

buckets: •  
num\_elems: 6  
hash: ...



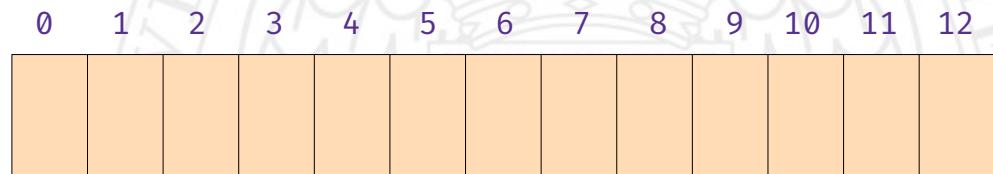
# Clase MapHash: constructores

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:
    MapHash(): num_elems(0), buckets(new List[CAPACITY]) { };

    MapHash(const MapHash &other): num_elems(other.num_elems),
                                    hash(other.hash),
                                    buckets(new List[CAPACITY]) {
        std::copy(other.buckets, other.buckets + CAPACITY, buckets);
    };

    ~MapHash() {
        delete[] buckets;
    }

private:
    List *buckets;
    int num_elems;
    Hash hash;
    // ...
};
```



# Clase MapHash: búsqueda

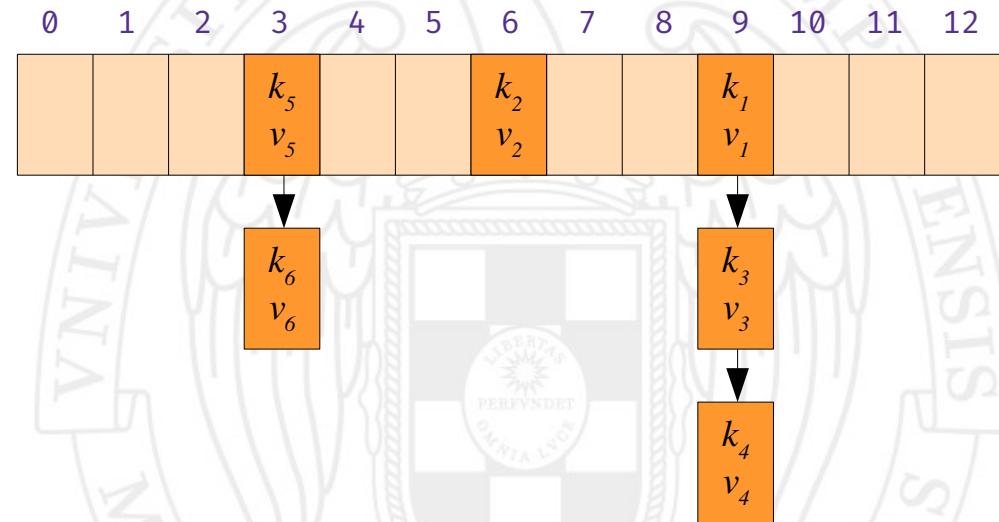
```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:

    const V & at(const K &key) const {
        int h = hash(key) % CAPACITY;
        const List &list = buckets[h];

        auto it = find_in_list(list, key);

        assert (it != list.end());
        return it->value;
    }

private:
    List *buckets;
    int num_elems;
    Hash hash;
    // ...
};
```



# Clase MapHash: búsqueda

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:

    const V & at(const K &key) const {
        int h = hash(key) % CAPACITY;
        const List &list = buckets[h];

        auto it = find_in_list(list, key);

        assert (it != list.end());
        return it->value;
    }

private:
    List *buckets;
    int num_elems;
    Hash hash;
    // ...
};
```

```
List::const_iterator find_in_list(const List &list, const K &key) {
    auto it = list.begin();
    while (it != list.end() && it->key != key) {
        ++it;
    }
    return it;
}
```

# Clase MapHash: inserción

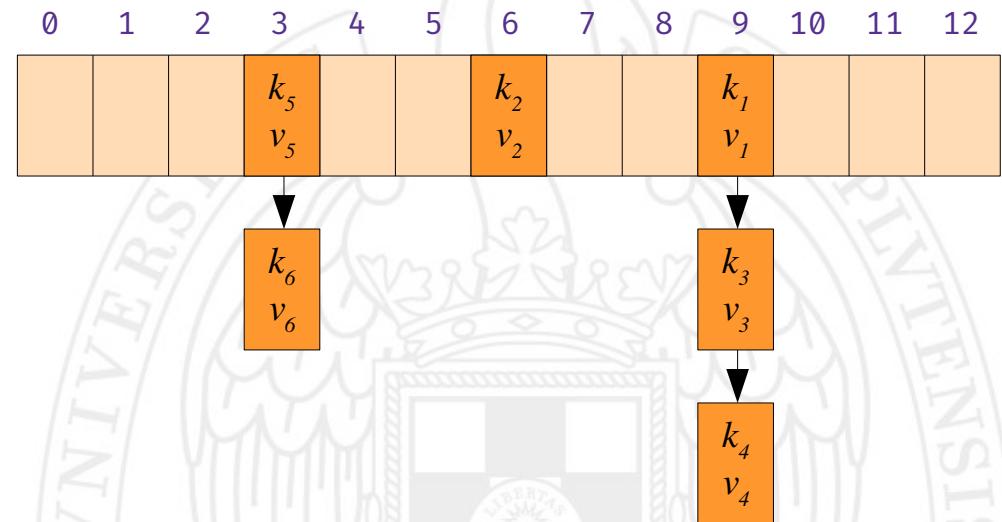
```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:

    void insert(const MapEntry &entry) {
        int h = hash(entry.key) % CAPACITY;
        List &list = buckets[h];

        auto it = find_in_list(list, entry.key);

        if (it == list.end()) {
            list.push_front(entry);
            num_elems++;
        }
    }

private:
    List *buckets;
    int num_elems;
    Hash hash;
    // ...
};
```



# Clase MapHash: inserción

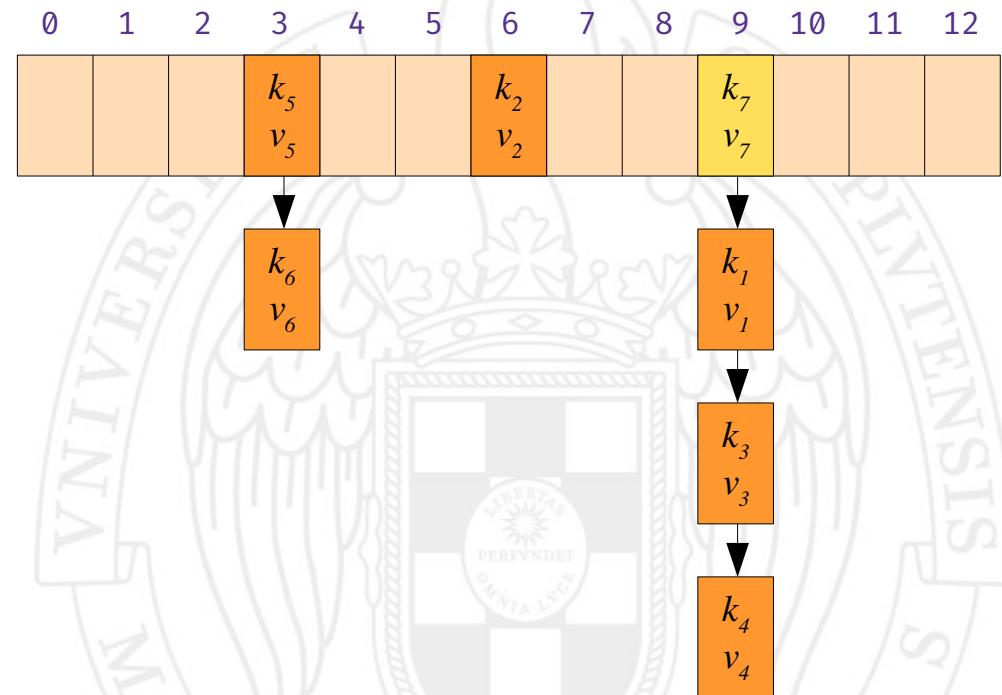
```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:

    void insert(const MapEntry &entry) {
        int h = hash(entry.key) % CAPACITY;
        List &list = buckets[h];

        auto it = find_in_list(list, entry.key);

        if (it == list.end()) {
            list.push_front(entry);
            num_elems++;
        }
    }

private:
    List *buckets;
    int num_elems;
    Hash hash;
    // ...
};
```



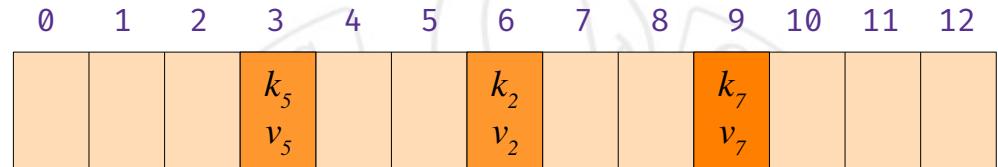
# Clase MapHash: borrado

- Similar a la inserción.
- La clase `forward_list` no tiene método `erase(it)`.
- Pero sí tiene método `erase_after(it)`, que elimina el elemento situado después del apuntado por el iterador.

# Clase MapHash: borrado

```
void erase(const K &key) {
    int h = hash(key) % CAPACITY;
    List &list = buckets[h];
    if (!list.empty()) {
        if (list.front().key == key) {
            list.pop_front();
            num_elems--;
        } else {
            auto it_prev = list.begin();
            auto it_next = ++list.begin();

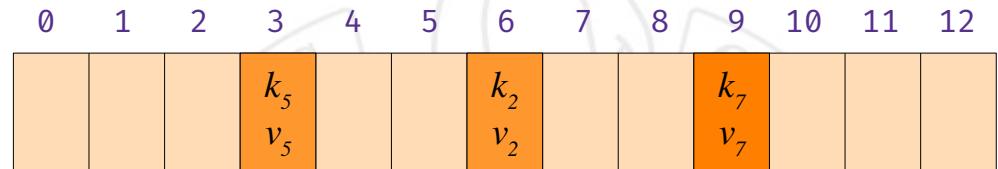
            while (it_next != list.end() && it_next->key != key) {
                it_prev++;
                it_next++;
            }
            if (it_next != list.end()) {
                list.erase_after(it_prev);
                num_elems--;
            }
        }
    }
}
```



# Clase MapHash: borrado

```
void erase(const K &key) {
    int h = hash(key) % CAPACITY;
    List &list = buckets[h];
    if (!list.empty()) {
        if (list.front().key == key) {
            list.pop_front();
            num_elems--;
        } else {
            auto it_prev = list.begin();
            auto it_next = ++list.begin();

            while (it_next != list.end() && it_next->key != key) {
                it_prev++;
                it_next++;
            }
            if (it_next != list.end()) {
                list.erase_after(it_prev);
                num_elems--;
            }
        }
    }
}
```



ESTRUCTURAS DE DATOS

DICCIONARIOS

# Tablas *hash* cerradas

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

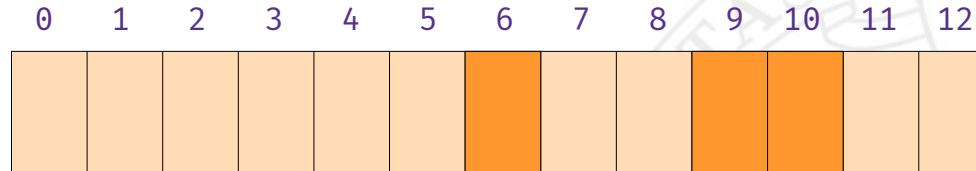
# Objetivo

- Implementar el TAD Diccionario mediante una tabla hash cerrada.

0	1	2	3	4	5	6	7	8	9	10	11	12
						$k_2$ $v_2$			$k_1$ $v_1$	$k_3$ $v_3$		

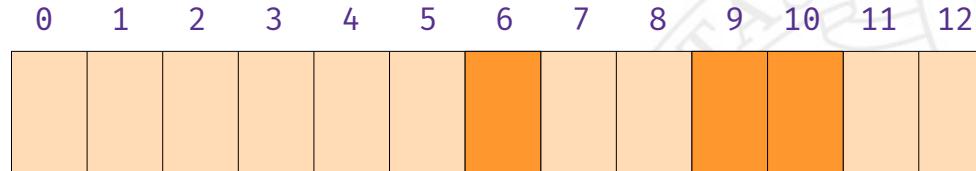
# Idea de implementación

- Existen posiciones **libres** y **ocupadas**.



# Inserción

- Si la entrada está ocupada, insertamos en la siguiente. Si también está ocupada, miramos en la siguiente, etc. hasta encontrar una posición libre.



# Ejemplo

- Insertamos claves 32, 49, 9, 61, 37, 23 (en este orden).

0	1	2	3	4	5	6	7	8	9	10	11	12
23 —						32 —			9 —	49 —	61 —	37 —

# Búsqueda

- Buscamos en el cajón  $h(k) \bmod CAPACITY$ .
- A partir de ahí, buscamos en entradas sucesivas hasta encontrar la clave o llegar a una posición vacía.

0	1	2	3	4	5	6	7	8	9	10	11	12
23 —						32 —			9 —	49 —	61 —	37 —

- Ejemplo: buscamos 23 y 63.

# ¡Cuidado con el borrado!

- Si eliminamos la entrada sin más, podemos imposibilitar la búsqueda de claves que vienen después.
- Ejemplo: borramos 61.

0	1	2	3	4	5	6	7	8	9	10	11	12
23 -						32 -			9 -	49 -	61 -	37 -

- ¿Y si ahora buscamos 23?

# Solución

- Distinguir entre entradas **libres** y entradas **eliminadas**.
- La búsqueda se detiene cuando llegamos a una posición libre.
- ...pero podemos escribir en entradas eliminadas.

0	1	2	3	4	5	6	7	8	9	10	11	12
23 —						32 —			9 —	49 —		37 —

- ¿Y si ahora buscamos 23?
- ¿Y si ahora insertamos la clave 36?

# Clase MapHash: interfaz pública

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:
    MapHash();
    MapHash(const MapHash &other);
    ~MapHash();

    void insert(const MapEntry &entry);
    void erase(const K &key);

    bool contains(const K &key) const;
    const V & at(const K &key) const;
    V & at(const K &key);
    V & operator[](const K &key);

    int size() const;
    bool empty() const;

    MapHash & operator=(const MapHash &other);
    void display(std::ostream &out) const;

private:
    // ...
};
```

```
struct MapEntry {
    K key;
    V value;

    MapEntry(K key, V value);
    MapEntry(K key);
};
```

# Clase MapHash: representación privada

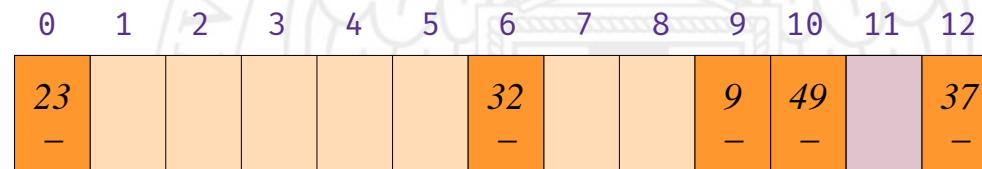
```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {

private:
    enum class State { empty, occupied, deleted };

    struct Bucket {
        State state;
        MapEntry entry;

        Bucket(): state(State::empty) { }
    };

    Bucket *buckets;
    Hash hash;
    int num_elems;
};
```



# Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key)
```

pos\_to\_insert

pos\_found

- Busca una clave `key` recibida como parámetro en el vector `buckets`.
- Componentes del objeto `pair` de salida:

- **pos\_found**

Posición del vector en la que se ha encontrado la clave. Si no se encuentra, es -1.

- **pos\_to\_insert**

Posición del vector en el que se debería insertar la clave, en caso de ser necesario (o -1 si el vector está lleno)

# Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key) const {
    int h = hash(key) % CAPACITY;

    int pos_to_insert = -1;
    int pos_found = -1;

    while (pos_found == -1 && buckets[h].state != State::empty) {
        if (pos_to_insert == -1 && buckets[h].state == State::deleted) {
            pos_to_insert = h;
        }

        if (buckets[h].state == State::occupied && buckets[h].entry.key == key) {
            pos_found = h;
        }
    }

    h = (h + 1) % CAPACITY;
}

if (pos_found == -1 && pos_to_insert == -1) {
    pos_to_insert = h;
}
return {pos_to_insert, pos_found};
}
```

# Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key) const {
    int h = hash(key) % CAPACITY;

    int pos_to_insert = -1;
    int pos_found = -1;

    while (pos_found == -1 && buckets[h].state != State::empty) {
        if (pos_to_insert == -1 && buckets[h].state == State::deleted) {
            pos_to_insert = h;
        }

        if (buckets[h].state == State::occupied && buckets[h].entry.key == key) {
            pos_found = h;
        }

        h = (h + 1) % CAPACITY;
    }

    if (pos_found == -1 && pos_to_insert == -1) {
        pos_to_insert = h;
    }

    return {pos_to_insert, pos_found};
}
```

Repetimos mientras no hayamos encontrado la clave, o hayamos llegado a una posición vacía

# Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key) const {
    int h = hash(key) % CAPACITY;

    int pos_to_insert = -1;
    int pos_found = -1;

    while (pos_found == -1 && buckets[h].state != State::empty) {
        if (pos_to_insert == -1 && buckets[h].state == State::deleted) {
            pos_to_insert = h;
        }

        if (buckets[h].state == State::occupied && buckets[h].entry.key == key) {
            pos_found = h;
        }

        h = (h + 1) % CAPACITY;
    }

    if (pos_found == -1 && pos_to_insert == -1) {
        pos_to_insert = h;
    }

    return {pos_to_insert, pos_found};
}
```

Cambiamos `pos_insert` a la primera posición eliminada que encontramos

# Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key) const {
    int h = hash(key) % CAPACITY;

    int pos_to_insert = -1;
    int pos_found = -1;

    while (pos_found == -1 && buckets[h].state != State::empty) {
        if (pos_to_insert == -1 && buckets[h].state == State::deleted) {
            pos_to_insert = h;
        }

        if (buckets[h].state == State::occupied && buckets[h].entry.key == key) {
            pos_found = h;
        }
    }

    h = (h + 1) % CAPACITY;
}

if (pos_found == -1 && pos_to_insert == -1) {
    pos_to_insert = h;
}

return {pos_to_insert, pos_found};
}
```

Si encontramos la clave key en la posición actual, establecemos pos\_found

# Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key) const {
    int h = hash(key) % CAPACITY;

    int pos_to_insert = -1;
    int pos_found = -1;

    while (pos_found == -1 && buckets[h].state != State::empty) {
        if (pos_to_insert == -1 && buckets[h].state == State::deleted) {
            pos_to_insert = h;
        }

        if (buckets[h].state == State::occupied && buckets[h].entry.key == key) {
            pos_found = h;
        }

        h = (h + 1) % CAPACITY;
    }

    if (pos_found == -1 && pos_to_insert == -1) {
        pos_to_insert = h;
    }

    return {pos_to_insert, pos_found};
}
```

Pasamos a la posición siguiente. Si llegamos al final del vector, volvemos a la posición 0.

# Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key) const {
    int h = hash(key) % CAPACITY;

    int pos_to_insert = -1;
    int pos_found = -1;

    while (pos_found == -1 && buckets[h].state != State::empty) {
        if (pos_to_insert == -1 && buckets[h].state == State::deleted) {
            pos_to_insert = h;
        }

        if (buckets[h].state == State::occupied && buckets[h].entry.key == key) {
            pos_found = h;
        }

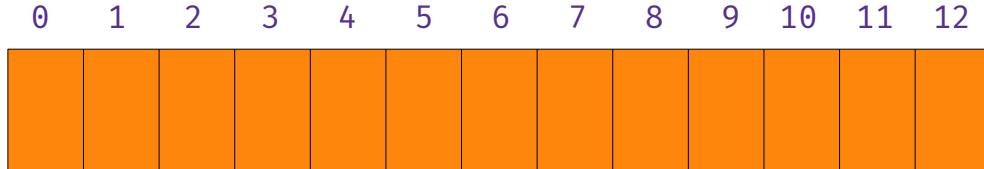
        h = (h + 1) % CAPACITY;
    }

    if (pos_found == -1 && pos_to_insert == -1) {
        pos_to_insert = h;
    }

    return {pos_to_insert, pos_found};
}
```

Si al final hemos llegado a una posición libre, y no hemos pasado por ninguna borrada, establecemos pos\_to\_insert

# ¿Y si el vector está lleno?



- Se van buscando posiciones de manera circular.
- ¡La función no termina!
- Debemos acotar el número de iteraciones.

# Función auxiliar de búsqueda de clave

```
std::pair<int, int> search_pos(const K &key) const {
    int h = hash(key) % CAPACITY;
    int cont = 0;

    int pos_to_insert = -1;
    int pos_found = -1;

    while (cont < CAPACITY && pos_found == -1 && buckets[h].state != State::empty) {
        if (pos_to_insert == -1 && buckets[h].state == State::deleted) {
            pos_to_insert = h;
        }

        if (buckets[h].state == State::occupied && buckets[h].entry.key == key) {
            pos_found = h;
        }

        h = (h + 1) % CAPACITY;
        cont++;
    }

    if (cont < CAPACITY && pos_to_insert == -1) {
        pos_to_insert = h;
    }
    return {pos_to_insert, pos_found};
}
```

# Método insert

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:

    void insert(const MapEntry &entry) {
        auto [pos_to_insert, pos_found] = search_pos(entry.key);
        if (pos_found == -1) {
            assert (pos_to_insert != -1);
            buckets[pos_to_insert].state = State::occupied;
            buckets[pos_to_insert].entry = entry;
            num_elems++;
        }
    }

private:
    // ...
};
```



# Método at

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:

    const V & at(const K &key) const {
        auto [pos_to_insert, pos_found] = search_pos(key);
        assert (pos_found != -1);
        return buckets[pos_found].entry.value;
    }

private:
    // ...
};
```



# Método erase

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
public:

    void erase(const K &key) {
        auto [pos_to_insert, pos_found] = search_pos(key);
        if (pos_found != -1) {
            buckets[pos_found].state = State::deleted;
            num_elems--;
        }
    }

private:
    // ...
};
```



# Búsqueda de posiciones alternativas

# Recordatorio

- Las tablas hash cerradas se basan en calcular una **posición inicial**  $p_0$  en el vector y buscar una clave allí.

$$p_0 = h(k) \bmod \text{CAPACITY}$$

- Si la clave no se encuentra, se busca en **posiciones alternativas**  $p_1, p_2, \text{ etc.}$

En nuestro caso:

$$p_1 = (h(k) + 1) \bmod \text{CAPACITY}$$

$$p_2 = (h(k) + 2) \bmod \text{CAPACITY}$$

$$p_3 = (h(k) + 3) \bmod \text{CAPACITY}$$

*etc.*

# Recordatorio

- Las tablas hash cerradas se basan en calcular una **posición inicial**  $p_0$  en el vector y buscar una clave allí.

$$p_0 = h(k) \bmod \text{CAPACITY}$$

- Si la clave no se encuentra, se busca en **posiciones alternativas**  $p_1, p_2, \text{ etc.}$

En general:

$$p_i = (h(k) + i) \bmod \text{CAPACITY}$$

# Sondeo lineal

- Existen otras posibilidades para la búsqueda de posiciones alternativas.
- La que hemos utilizado es un caso particular del **sondeo lineal**.

$$p_i = (h(k) + ci) \bmod \text{CAPACITY}$$

- Por ejemplo, si  $c = 1$ ,  $h(k) = 10$ ,  $\text{CAPACITY} = 13$ .

10    11    12    0    1    2    3    ...

# Sondeo lineal

- Existen otras alternativas para la búsqueda de posiciones alternativas.
- La que hemos utilizado es un caso particular del **sondeo lineal**.

$$p_i = (h(k) + ci) \bmod \text{CAPACITY}$$

- Por ejemplo, si  $c = 2$ ,  $h(k) = 10$ ,  $\text{CAPACITY} = 13$ .

10    12    1    3    5    7    9    11    ...

# Sondeo lineal

- Existen otras alternativas para la búsqueda de posiciones alternativas.
- La que hemos utilizado es un caso particular del **sondeo lineal**.

$$p_i = (h(k) + ci) \bmod \text{CAPACITY}$$

- Por ejemplo, si  $c = 5$ ,  $h(k) = 10$ ,  $\text{CAPACITY} = 13$ .

10    2    7    12    4    9    ...

# Sondeo lineal

- Existen otras alternativas para la búsqueda de posiciones alternativas.
- La que hemos utilizado es un caso particular del **sondeo lineal**.

$$p_i = (h(k) + ci) \bmod CAPACITY$$

- En general, si  $\text{mcd}(c, CAPACITY) = 1$ , el sondeo lineal garantiza el recorrido de todas las posiciones del array, en caso de ser necesario.
- En general, esto ocurre cuando  $CAPACITY$  es primo.

# Sondeo cuadrático

- Utiliza la siguiente fórmula:

$$p_i = (h(k) + i^2) \bmod CAPACITY$$

- Por ejemplo, si  $h(k) = 10$ ,  $CAPACITY = 13$ .

10    11    1    6    0    9    7    7 ...

# Sondeo cuadrático

- Utiliza la siguiente fórmula:

$$p_i = (h(k) + i^2) \bmod CAPACITY$$

- Aquí no se garantiza el recorrido de todas las posiciones del vector, pero sí al menos la mitad de ellas.

# Doble redispersión

- Utiliza una segunda función hash  $h'$  para la búsqueda de posiciones alternativas.

$$p_i = (h(k) + i h'(k)) \bmod \text{CAPACITY}$$

- Garantiza el recorrido de todas las posiciones si  $\text{CAPACITY}$  es primo y  $h'(k)$  nunca devuelve 0.

# Más información

- R. Peña  
*Diseño de Programas. Formalismo y Abstracción (3<sup>a</sup> edición)*  
Pearson Educación (2005)  
Sección 8.1.3
- [https://en.wikipedia.org/wiki/Linear\\_probing](https://en.wikipedia.org/wiki/Linear_probing)
- [https://en.wikipedia.org/wiki/Quadratic\\_probing](https://en.wikipedia.org/wiki/Quadratic_probing)
- [https://en.wikipedia.org/wiki/Double\\_hashing](https://en.wikipedia.org/wiki/Double_hashing)

ESTRUCTURAS DE DATOS

DICCIONARIOS

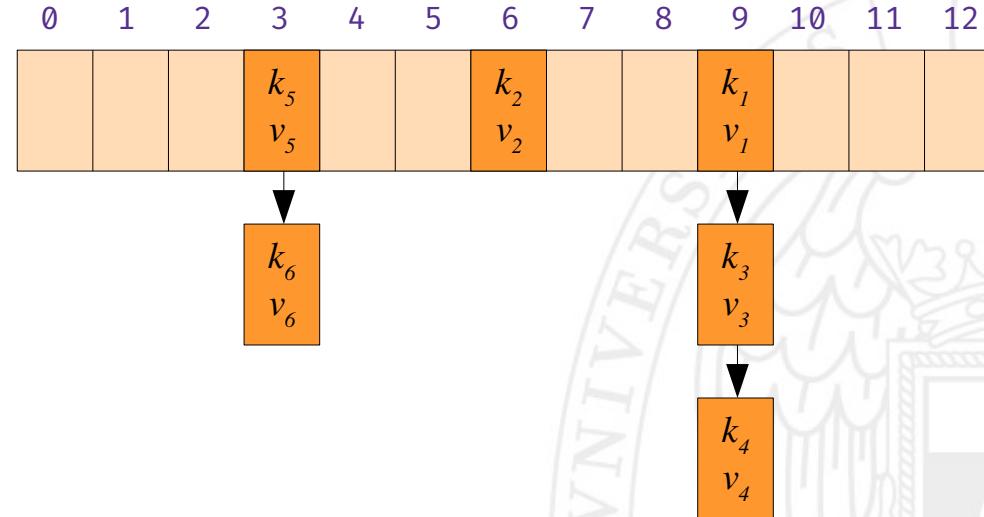
# Análisis de coste en tablas *hash*

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

# Tablas *hash* abiertas

# Recordatorio

- Una tabla *hash* abierta asocia cada cajón con una **lista de entradas**.
- En caso de colisión entre claves, las entradas acaban en la misma lista.



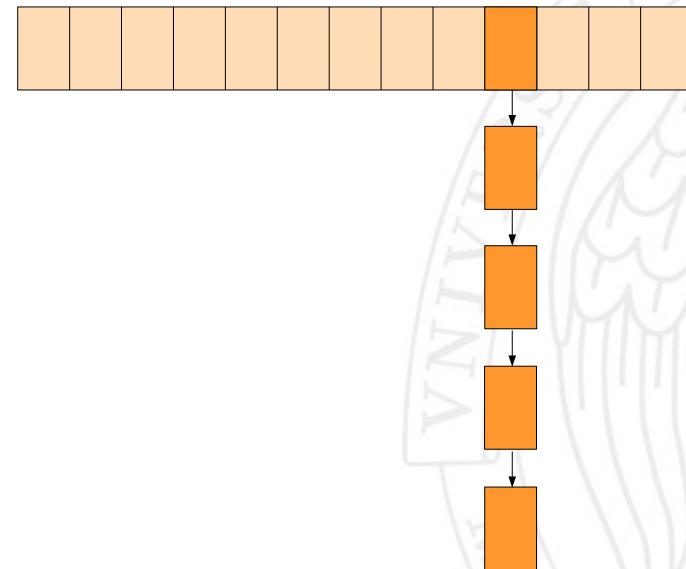
# Factor de carga

- El **factor de carga**  $\alpha$  de una tabla hash es el cociente entre el número de entradas en la tabla y el número de cajones.
- Sean:
  - $n$  - número de entradas en la tabla
  - $m$  - número de cajones
- Expresaremos el coste de los algoritmos en función del factor de carga.

$$\alpha = \frac{n}{m}$$

# Dispersión uniforme

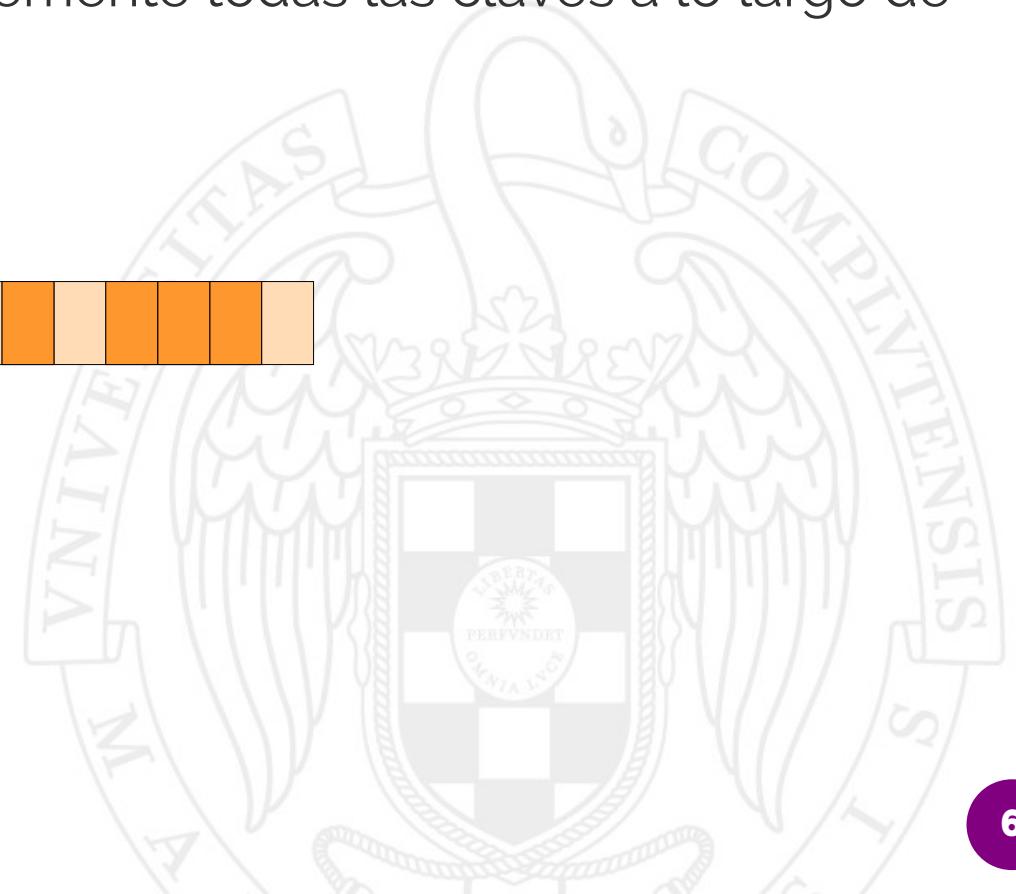
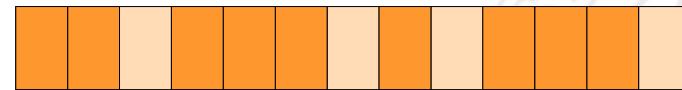
- La eficiencia de una tabla *hash* viene determinada por las propiedades de la función *hash* utilizada.
- Una función *hash* nefasta enviaría todas las claves al mismo cajón.



# Dispersión uniforme

## Suposición de dispersión uniforme:

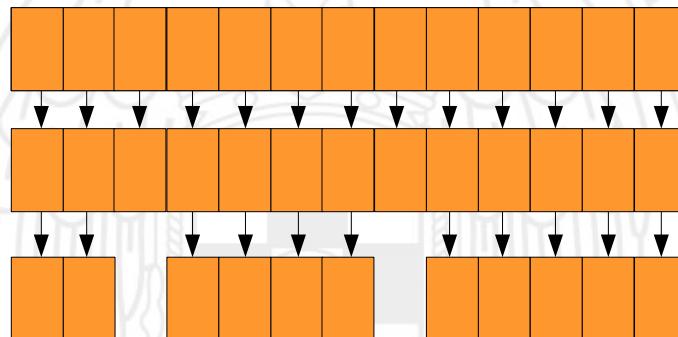
La función *hash* distribuye uniformemente todas las claves a lo largo de los cajones de la tabla.



# Longitud media de las listas

- Supongamos que tenemos  $n$  elementos y  $m$  cajones, y que la propiedad de distribución uniforme se cumple.
- Sea  $N_i$  la longitud de la lista del cajón  $i$ -ésimo.
- ¿Cuál es el valor promedio de  $N_i$ ?

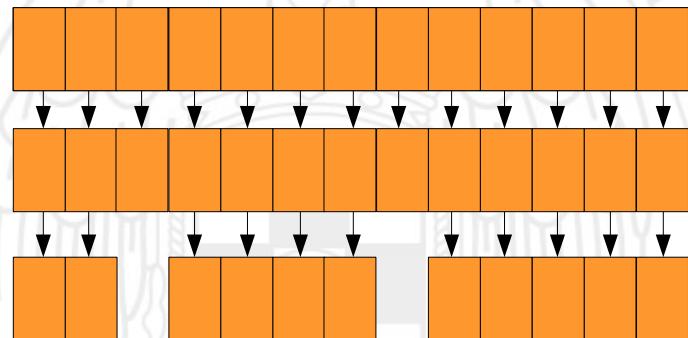
$$E[N_i] = \frac{n}{m} = \alpha$$



# Búsqueda de una clave (fallo)

- Supongamos que realizamos una búsqueda de una clave que no se encuentra en la tabla.
- El coste debe recorrer una de las listas en su totalidad.
- Por tanto, el coste medio es proporcional a  $\alpha$ .
- Similarmente para la inserción y borrado.

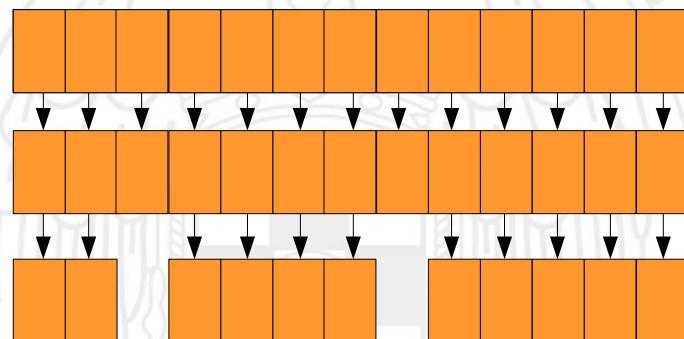
$$O(1 + \alpha)$$



# Búsqueda de una clave (éxito)

- Supongamos que realizamos una búsqueda de una clave que sí se encuentra en la tabla.
- El número medio de elementos recorridos es  $1 + \frac{\alpha}{2} - \frac{\alpha}{2n}$
- Similarmente para la inserción y borrado.

$$1 + \frac{\alpha}{2} - \frac{\alpha}{2n} \in O(1 + \alpha)$$



# Conclusión

- El coste de todas las operaciones está acotado por  $\alpha$ .
- **Si conseguimos mantener  $\alpha$  acotado, el coste de las operaciones será constante.**
- ¿Cómo conseguimos mantener  $\alpha$  acotado?

Haciendo que el número de cajones aumente proporcionalmente con el número de entradas → *Tabla dinámicamente redimensionable*.

# Tablas *hash* cerradas

# Recordatorio

- Una tabla *hash* cerrada contiene, en cada cajón, una única entrada.
- En caso de colisión entre claves, las entradas acaban en cajones distintos según la estrategia de redispersión utilizada.

0	1	2	3	4	5	6	7	8	9	10	11	12
			$k_5$ $v_5$	$k_6$ $v_6$		$k_2$ $v_2$			$k_1$ $v_1$	$k_3$ $v_3$	$k_4$ $v_4$	

# Factor de carga

- Sean:

$n$  - número de entradas en la tabla

$m$  – número de cajones

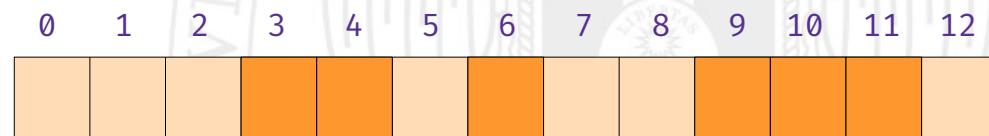
$$\alpha = \frac{n}{m}$$

- En una *tabla hash* cerrada, siempre se cumple que  $\alpha \leq 1$ .

# Búsqueda de una clave (fallo)

- Sea  $N$  una variable aleatoria que denota el número de intentos infructuosos hasta que llegamos a un cajón vacío.

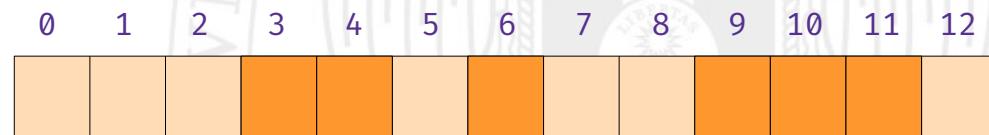
$$P\{N \geq 1\} = \frac{n}{m}$$



# Búsqueda de una clave (fallo)

- Sea  $N$  una variable aleatoria que denota el número de intentos infructuosos hasta que llegamos a un cajón vacío.

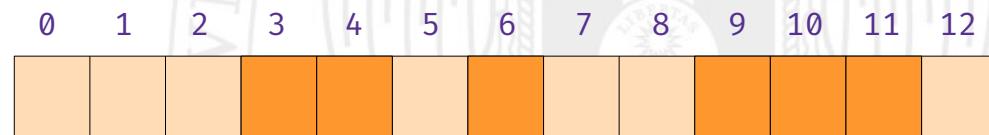
$$P\{N \geq 2\} = \frac{n}{m} * \frac{n - 1}{m - 1}$$



# Búsqueda de una clave (fallo)

- Sea  $N$  una variable aleatoria que denota el número de intentos infructuosos hasta que llegamos a un cajón vacío.

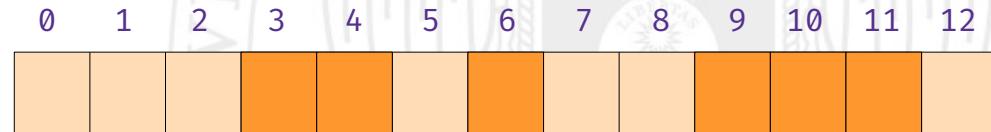
$$P\{N \geq 3\} = \frac{n}{m} * \frac{n-1}{m-1} * \frac{n-2}{m-2}$$



# Búsqueda de una clave (fallo)

- Sea  $N$  una variable aleatoria que denota el número de intentos infructuosos hasta que llegamos a un cajón vacío.

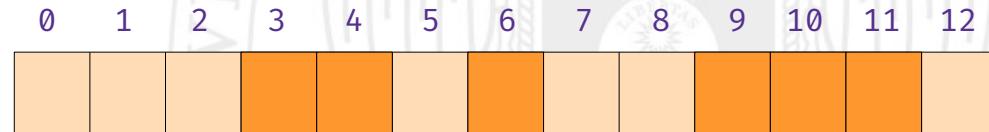
$$P\{N \geq i\} = \frac{n}{m} * \frac{n-1}{m-1} * \frac{n-2}{m-2} * \cdots * \frac{n-i+1}{m-i+1}$$



# Búsqueda de una clave (fallo)

- Sea  $N$  una variable aleatoria que denota el número de intentos infructuosos hasta que llegamos a un cajón vacío.

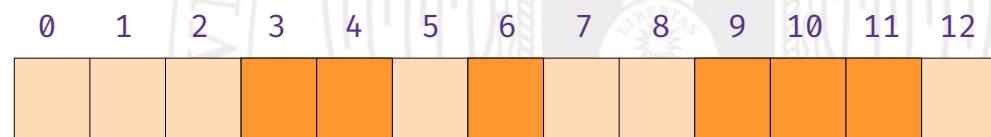
$$P\{N \geq i\} \leq \alpha^i$$



# Búsqueda de una clave (fallo)

- Promedio de cajones visitados para buscar una clave:

$$1 + E[N] = 1 + \sum_{i=1}^{\infty} P\{N \geq i\} \leq 1 + \sum_{i=1}^{\infty} \alpha^i = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1 - \alpha}$$



# Búsqueda de una clave (fallo)

- Promedio de cajones visitados para buscar una clave:

$$1 + E[N] = 1 + \sum_{i=1}^{\infty} P\{N \geq i\} \leq 1 + \sum_{i=1}^{\infty} \alpha^i = \sum_{i=0}^{\infty} \alpha^i = \frac{1}{1 - \alpha}$$

- Si  $\alpha = 0.9$ , entonces se visitan 10 cajones en el caso medio.
- Si  $\alpha = 0.8$ , entonces se visitan 5 cajones en el caso medio.
- Si  $\alpha = 0.5$ , entonces se visitan 2 cajones en el caso medio.

# Búsqueda de una clave (éxito)

- Promedio de cajones visitados para buscar una clave:

$$\frac{1}{\alpha} * \ln \frac{1}{1 - \alpha}$$

- Si  $\alpha = 0.9$ , entonces se visitan 2.56 cajones en el caso medio.
- Si  $\alpha = 0.8$ , entonces se visitan 2 cajones en el caso medio.
- Si  $\alpha = 0.5$ , entonces se visitan 1.38 cajones en el caso medio.

# Conclusión

- Si conseguimos mantener  $\alpha$  inferior a 1, el coste de las operaciones será constante.
- Con  $\alpha \leq 0.8$  se obtienen constantes razonables.
- ¿Cómo conseguimos mantener  $\alpha$  constante?
  - Haciendo que el número de cajones aumente proporcionalmente con el número de entradas → *Tabla dinámicamente redimensionable*.

# Bibliografía

- T. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein  
*Introduction to Algorithms (3<sup>a</sup> edición)*  
The MIT Press (2009)  
Capítulo 11
- R. Peña  
*Diseño de Programas. Formalismo y Abstracción (3<sup>a</sup> edición)*  
Pearson Educación (2005)  
Sección 8.1.3



ESTRUCTURAS DE DATOS

DICCIONARIOS

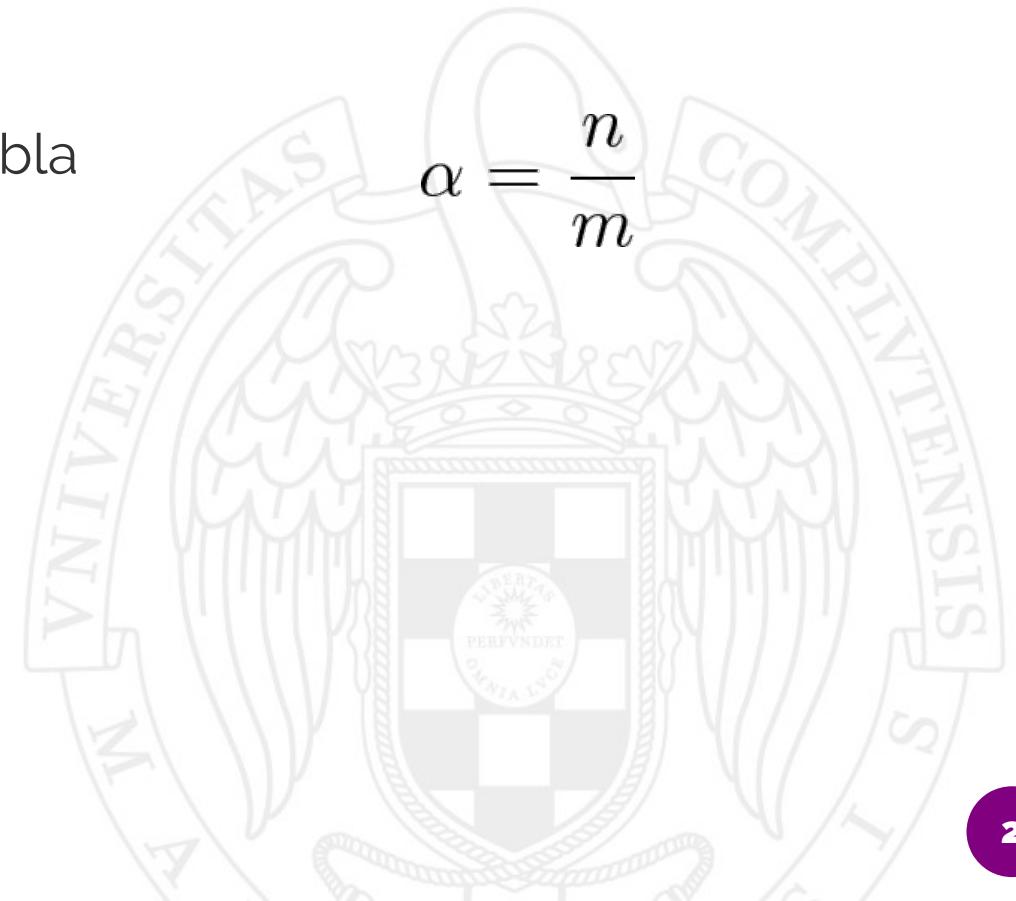
# Tablas *hash* redimensionables

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

# Recordatorio: factor de carga

- El **factor de carga**  $\alpha$  de una tabla hash es el cociente entre el número de entradas en la tabla y el número de cajones.
- Sean:
  - $n$  - número de entradas en la tabla
  - $m$  - número de cajones

$$\alpha = \frac{n}{m}$$



# Tablas redimensionables

- En el caso medio, las operaciones en una tabla hash abierta tienen coste  $O(1 + \alpha)$ .
- Por tanto, conseguimos coste constante en el caso medio si mantenemos el factor de carga acotado.
- Una **tabla hash redimensionable** es una tabla que se amplía cada vez que el factor de carga supera un determinado valor umbral.

# Implementación

```
const int INITIAL_CAPACITY = 31;  
const double MAX_LOAD_FACTOR = 0.8;
```

Factor de carga máximo permitido

```
template <typename K, typename V, typename Hash = std::hash<K>>  
class MapHash {  
    ...  
  
private:  
    using List = std::forward_list<MapEntry>;  
  
    Hash hash;  
  
    List *buckets;  
    int num_elems;  
    int capacity;
```

Tamaño del vector

# Implementación de insert (antes)

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
    ...
    void insert(const MapEntry &entry) {
        int h = hash(entry.key) % capacity;

        auto it = find_in_list(buckets[h], entry.key);

        if (it == buckets[h].end()) {
            buckets[h].push_front(entry);
            num_elems++;
        }
    }

private:
    ...
};
```

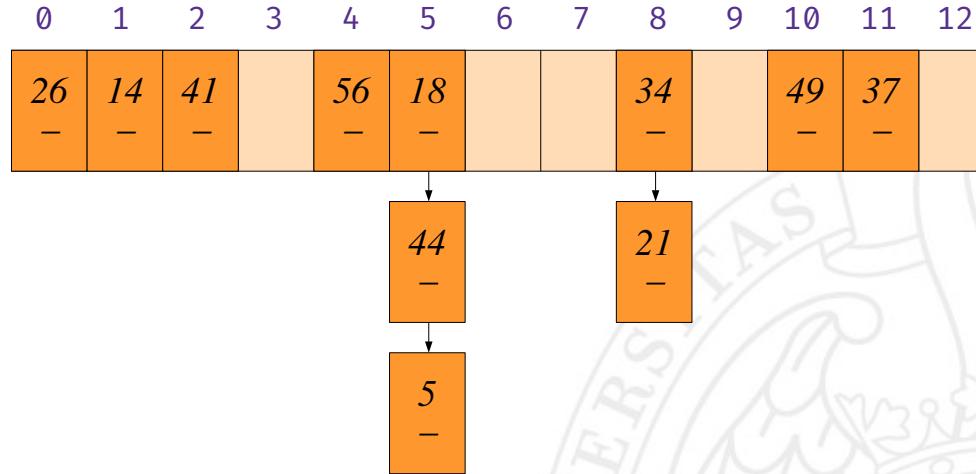


# Implementación de insert (después)

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
    ...
    void insert(const MapEntry &entry) {
        int h = hash(entry.key) % capacity;
        auto it = find_in_list(buckets[h], entry.key);
        if (it == buckets[h].end()) {
            num_elems++;
            resize_if_necessary();
            h = hash(entry.key) % capacity;
            buckets[h].push_front(entry);
        }
    }
private:
    ...
};
```



# Ejemplo de redimensionamiento



# Ejemplo de redimensionamiento

0	1	2	3	4	5	6	7	8	9	10	11	12
26 —	14 —	41 —		56 —	18 —			34 —		49 —	37 —	



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28

# Ejemplo de redimensionamiento

0	1	2	3	4	5	6	7	8	9	10	11	12
26	14	41		56	18			34		49	37	
-	-	-		-	-			-		-	-	



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
					5			37				41		14	44		18		49	21						26	56	



# Método auxiliar de redimensionamiento

```
template <typename K, typename V, typename Hash = std::hash<K>>
class MapHash {
private:

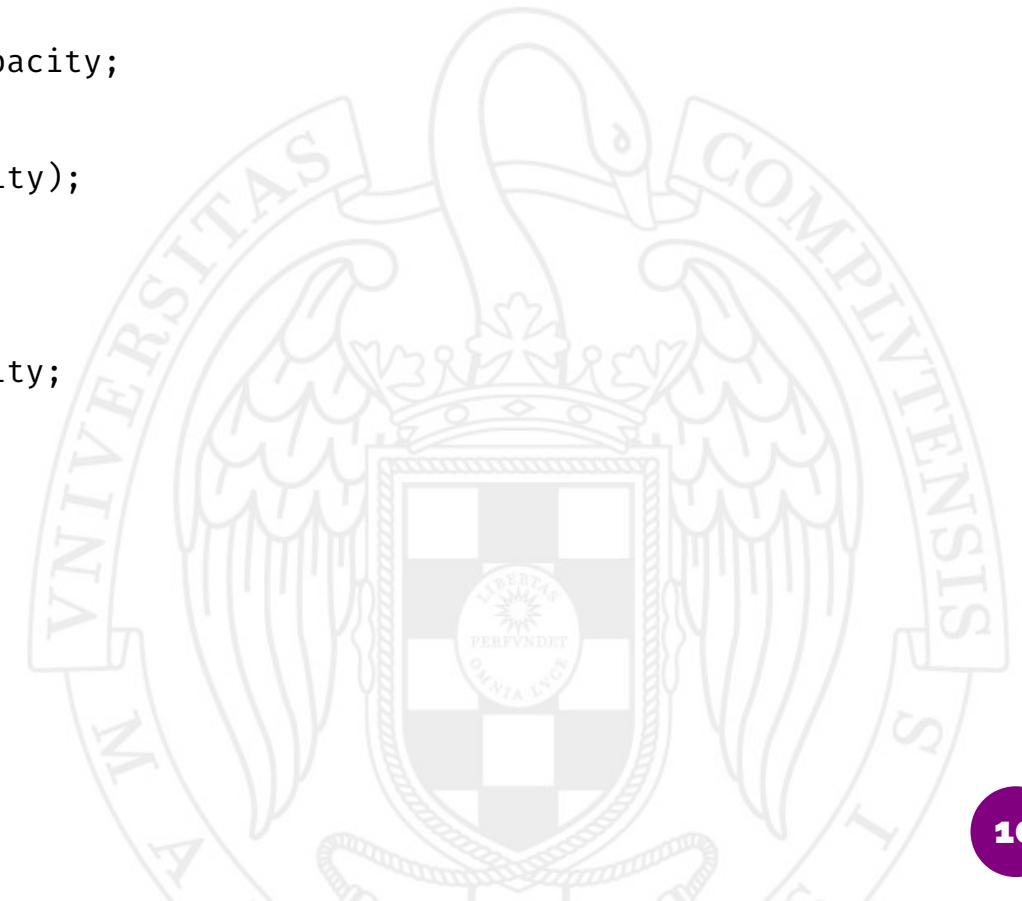
    void resize_if_necessary() {
        double load_factor = ((double)num_elems) / capacity;
        if (load_factor < MAX_LOAD_FACTOR) return;

        int new_capacity = next_prime_after(2 * capacity);
        List *new_array = new List[new_capacity];

        for (int i = 0; i < capacity; i++) {
            for (const MapEntry &entry : buckets[i]) {
                int new_pos = hash(entry.key) % new_capacity;
                new_array[new_pos].push_front(entry);
            }
        }

        capacity = new_capacity;
        delete[] buckets;
        buckets = new_array;
    }

    ...
};
```



# Costes en tiempo

- Suponiendo **dispersión uniforme**

Operación	Tabla hash
<i>constructor</i>	$O(1)$
<i>empty</i>	$O(1)$
<i>size</i>	$O(1)$
<i>contains</i>	$O(1)$
<i>at</i>	$O(1)$
<i>operator[ ]</i>	$O(1) / O(n)$
<i>insert</i>	$O(1) / O(n)$
<i>erase</i>	$O(1)$

$n$  = número de entradas en la tabla

# Costes amortizados en tiempo

- Suponiendo **dispersión uniforme**.

Operación	Tabla <i>hash</i>
<i>constructor</i>	$O(1)$
<i>empty</i>	$O(1)$
<i>size</i>	$O(1)$
<i>contains</i>	$O(1)$
<i>at</i>	$O(1)$
<i>operator[ ]</i>	$O(1)$
<i>insert</i>	$O(1)$
<i>erase</i>	$O(1)$

$n$  = número de entradas en la tabla

ESTRUCTURAS DE DATOS

APLICACIONES DE TIPOS ABSTRACTOS DE DATOS

# Gestión de una academia (1)

Manuel Montenegro Montes

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

# Versión inicial

# Requisitos

- Academia que ofrece una serie de cursos.
- Cada curso tiene un límite de plazas.
- Operaciones soportadas:
  - Crear una academia vacía (sin cursos ni estudiantes).
  - Añadir un curso a la academia.
  - Eliminar un curso de la academia.
  - Matricular a un estudiante en un curso.
  - Saber el número de plazas libres de un curso.
  - Obtener un listado de personas matriculadas en un curso, ordenado alfabéticamente por apellido.

# Métricas de coste

- $M$  = número de cursos total.
- $NC$  = número de estudiantes máximo por curso.

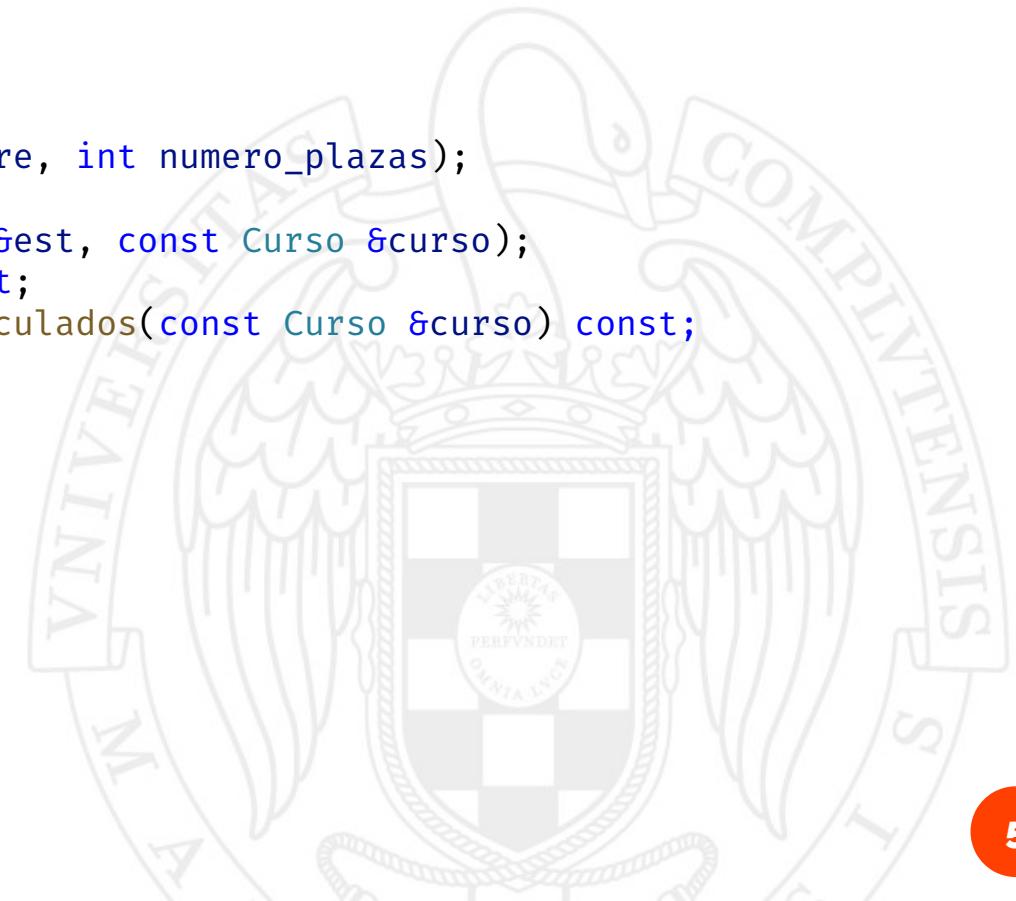


# Interfaz

```
using Estudiante = std::string;
using Curso = std::string;

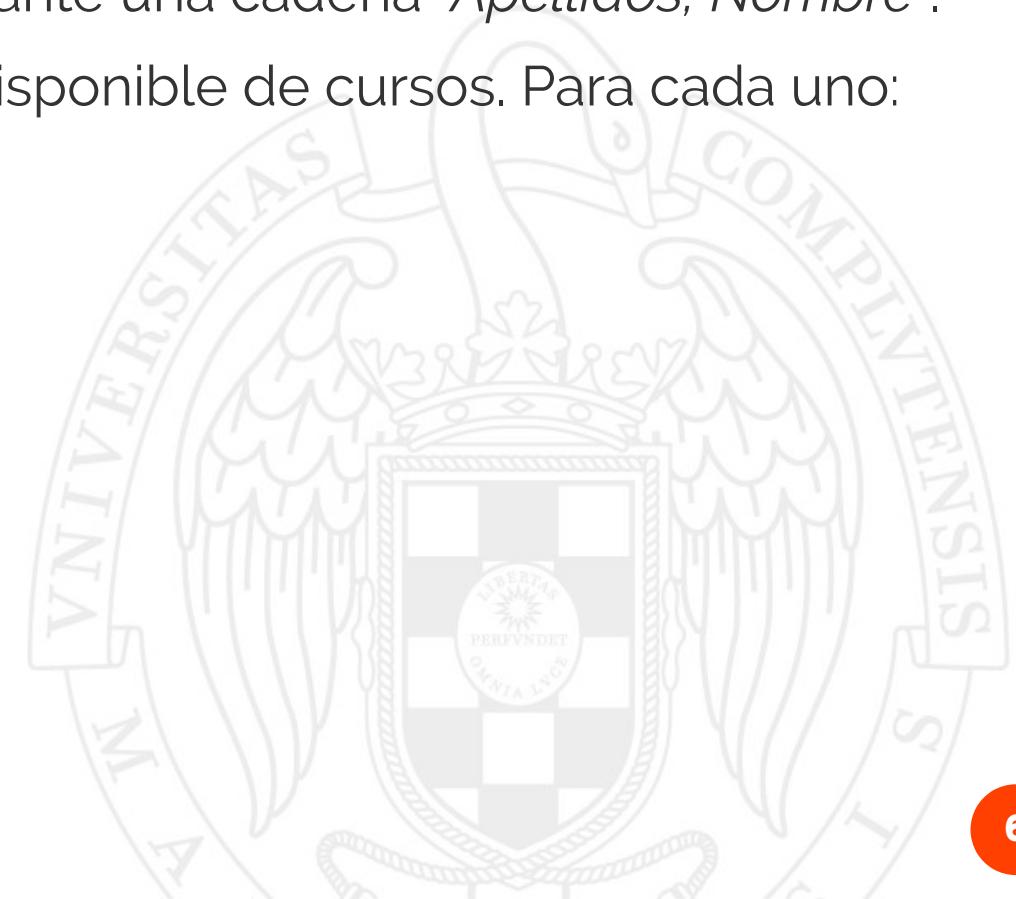
class Academia {
public:
    Academia();
    void anyadir_curso(const std::string &nombre, int numero_plazas);
    void eliminar_curso(const Curso &curso);
    void matricular_en_curso(const Estudiante &est, const Curso &curso);
    int plazas_libres(const Curso &curso) const;
    std::vector<std::string> estudiantes_matriculados(const Curso &curso) const;

private:
    ...
}
```



# Representación

- Cada curso se identifica mediante su nombre.
- Cada estudiante se identifica mediante una cadena “*Apellidos, Nombre*”.
- Debemos almacenar el catálogo disponible de cursos. Para cada uno:
  - Número de plazas total.
  - Estudiantes matriculados.



# Colección de cursos

```
using Estudiante = std::string;
using Curso = std::string;

class Academia {
public:
    Academia();
    void anyadir_curso(nombre, numero_plazas);
    void eliminar_curso(curso);
    void matricular_en_curso(est, curso);
    int plazas_libres(curso);
    vector<...> estudiantes_matriculados(curso);

private:
    ...
}
```

- ¿Qué TAD necesitamos para almacenar los cursos?
  - Lista.
  - Pila / cola / doble cola.
  - Conjunto.
  - Diccionario.
  - Multiconjunto.
  - Multidiccionario.

# Colección de cursos

```
using Estudiante = std::string;
using Curso = std::string;

class Academia {
public:
    Academia();
    void anyadir_curso(nombre, numero_plazas);
    void eliminar_curso(curso);
    void matricular_en_curso(est, curso);
    int plazas_libres(curso);
    vector<...> estudiantes_matriculados(curso);

private:
    ...
}
```

- ¿Necesitamos recorrer los cursos en un determinado orden?
  - map
  - unordered\_map

# Colección de cursos: representación

```
using Estudiante = std::string;
using Curso = std::string;

class Academia {
public:
    ...
private:
    struct InfoCurso {
        std::string nombre;
        int numero_plazas;
        ??? estudiantes;
    };
    InfoCurso(const std::string &nombre,
              int numero_plazas);
};

std::unordered_map<Curso, InfoCurso> cursos;
}
```



# Colección de estudiantes

```
using Estudiante = std::string;
using Curso = std::string;

class Academia {
public:
    ...
private:
    struct InfoCurso {
        std::string nombre;
        int numero_plazas;
        ??? estudiantes;
    };
    InfoCurso(const std::string &nombre,
              int numero_plazas);
};

std::unordered_map<Curso, InfoCurso> cursos;
```

- ¿Qué TAD necesitamos para almacenar la colección de estudiantes?
  - Lista.
  - Pila / cola / doble cola.
  - Conjunto.
  - Diccionario.
  - Multiconjunto.
  - Multidiccionario.

# Colección de estudiantes

```
using Estudiante = std::string;
using Curso = std::string;

class Academia {
public:
    ...
private:
    struct InfoCurso {
        std::string nombre;
        int numero_plazas;
        ??? estudiantes;
    };
    InfoCurso(const std::string &nombre,
              int numero_plazas);
};

std::unordered_map<Curso, InfoCurso> cursos;
}
```

- ¿Necesitamos mantener los estudiantes matriculados en un determinado orden?
  - set
  - unordered\_set



# Colección de estudiantes: representación

```
using Estudiante = std::string;
using Curso = std::string;

class Academia {
public:
    ...
private:
    struct InfoCurso {
        std::string nombre;
        int numero_plazas;
        std::set<Estudiante> estudiantes;

        InfoCurso(const std::string &nombre,
                  int numero_plazas);
    };
    std::unordered_map<Curso, InfoCurso> cursos;
}
```



# Añadir un curso

```
class Academia {  
public:  
  
    void anyadir_curso(const std::string &nombre, int numero_plazas) {  
        if (cursos.contains(nombre)) {  
            throw std::domain_error("curso ya existente");  
        }  
        cursos.insert({nombre, InfoCurso(nombre, numero_plazas)});  
    }  
  
    ...  
private:  
    ...  
    std::unordered_map<Curso, InfoCurso> cursos;  
}
```



# Eliminar un curso

```
class Academia {  
public:  
  
    void eliminar_curso(const Curso &curso) {  
        cursos.erase(curso);  
    }  
  
    ...  
  
private:  
    ...  
    std::unordered_map<Curso, InfoCurso> cursos;  
}
```



# Matrícula en un curso

```
class Academia {  
public:  
    void matricular_en_curso(const Estudiante &est, const Curso &curso) {  
        if (!cursos.contains(curso)) {  
            throw std::domain_error("curso no existente");  
        }  
        InfoCurso &info_curso = cursos.at(curso);  
        if (info_curso.estudiantes.contains(est)) {  
            throw std::domain_error("estudiante ya matriculado");  
        }  
  
        if (info_curso.estudiantes.size() ≥ info_curso.numero_plazas) {  
            throw std::domain_error("no hay plazas disponibles");  
        }  
  
        info_curso.estudiantes.insert(est);  
    }  
    ...  
private:  
    ...  
    std::unordered_map<Curso, InfoCurso> cursos;  
}
```

# Matrícula en un curso

```
class Academia {  
public:  
    void matricular_en_curso(const Estudiante &est, const Curso &curso) {  
        auto it = cursos.find(curso);  
        if (it == cursos.end()) {  
            throw std::domain_error("curso no existente");  
        }  
        InfoCurso &info_curso = it->second;  
        if (info_curso.estudiantes.contains(est)) {  
            throw std::domain_error("estudiante ya matriculado");  
        }  
  
        if (info_curso.estudiantes.size() ≥ info_curso.numero_plazas) {  
            throw std::domain_error("no hay plazas disponibles");  
        }  
  
        info_curso.estudiantes.insert(est);  
    }  
...  
private:  
...  
    std::unordered_map<Curso, InfoCurso> cursos;  
}
```



# Matrícula en un curso

```
class Academia {  
public:  
  
    void matricular_en_curso(const Estudiante &est, const Curso &curso) {  
        InfoCurso &info_curso = buscar_curso(curso);  
        if (info_curso.estudiantes.contains(est)) {  
            throw std::domain_error("estudiante ya matriculado");  
        }  
  
        if (info_curso.estudiantes.size() ≥ info_curso.numero_plazas) {  
            throw std::domain_error("no hay plazas disponibles");  
        }  
  
        info_curso.estudiantes.insert(est);  
    }  
    ...  
private:  
    ...  
    std::unordered_map<Curso, InfoCurso> cursos;  
}
```



# Número de plazas disponibles

```
class Academia {  
public:  
  
    int plazas_libres(const Curso &curso) {  
        const InfoCurso &info_curso = buscar_curso(curso);  
        return info_curso.numero_plazas - info_curso.estudiantes.size();  
    }  
  
    ...  
private:  
    ...  
    std::unordered_map<Curso, InfoCurso> cursos;  
}
```



# Estudiantes matriculados

```
class Academia {  
public:  
  
    std::vector<std::string> estudiantes_matriculados(const Curso &curso) const {  
        const InfoCurso &info_curso = buscar_curso(curso);  
        std::vector<std::string> result;  
        for (const Estudiante &est: info_curso.estudiantes) {  
            result.push_back(est);  
        }  
  
        return result;  
    }  
  
    ...  
private:  
    ...  
    std::unordered_map<Curso, InfoCurso> cursos;  
}
```



# Estudiantes matriculados

```
class Academia {  
public:  
  
    std::vector<std::string> estudiantes_matriculados(const Curso &curso) const {  
        const InfoCurso &info_curso = buscar_curso(curso);  
        std::vector<std::string> result;  
        std::copy(info_curso.estudiantes.begin(), info_curso.estudiantes.end(),  
                 std::back_insert_iterator<std::vector<std::string>>(result));  
        return result;  
    }  
    ...  
private:  
    ...  
    std::unordered_map<Curso, InfoCurso> cursos;  
}
```

ESTRUCTURAS DE DATOS

APLICACIONES DE TIPOS ABSTRACTOS DE DATOS

# Gestión de una academia (2)

Manuel Montenegro Montes

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

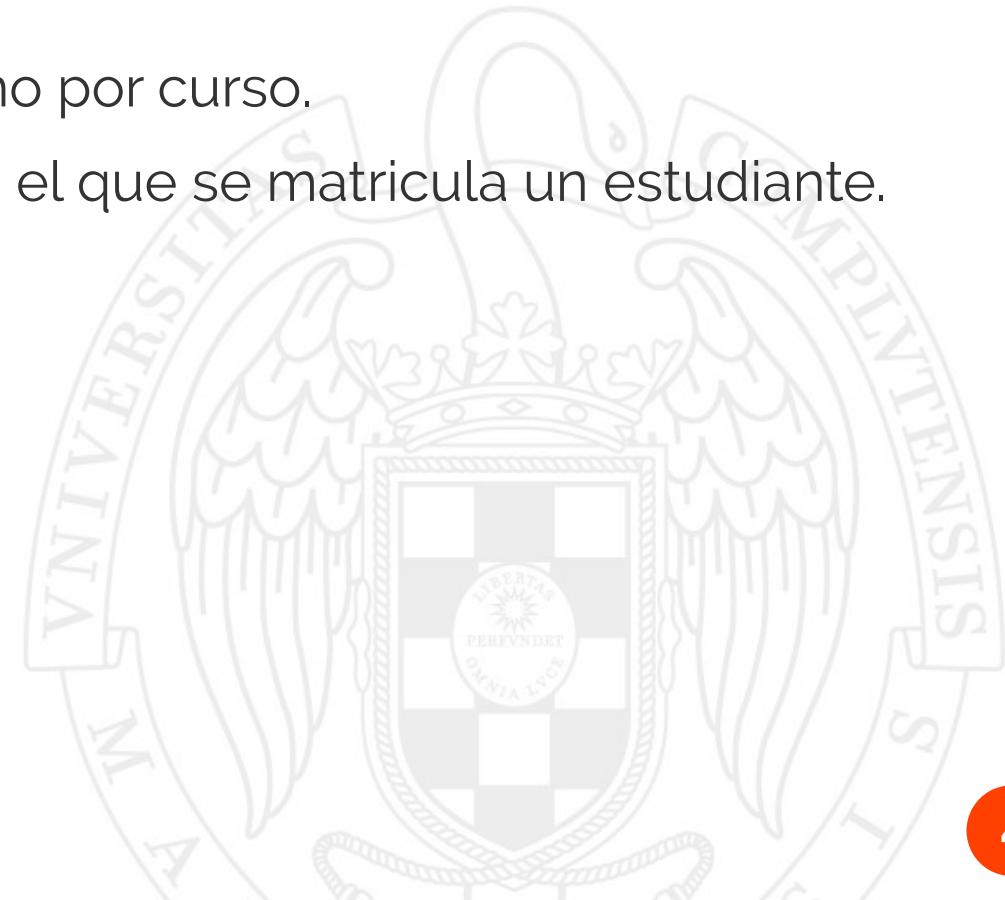
# Registro de estudiantes

# Requisitos

- Queremos mantener un registro de estudiantes.
- Antes de matricularse en un curso, los estudiantes han de estar registrados en la academia.
- Los estudiantes no se identificarán por nombre y apellidos. El identificador de un estudiante es su número de documento de identidad (NIF, NIE, etc.)
- Operaciones soportadas:
  - Añadir un estudiante a la academia.
  - Obtener un listado (ordenado alfabéticamente) de los cursos en los que está matriculado un estudiante.

# Métricas de coste

- $M$  = número de cursos total.
- $N$  = número de estudiantes total.
- $NC$  = número de estudiantes máximo por curso.
- $MC$  = número de cursos máximo en el que se matricula un estudiante.



# Registro de estudiantes

```
using Estudiante = std::string; ← Ahora representa el NIF  
using Curso = std::string;
```

```
class Academia {  
public:  
    ...  
  
private:  
    struct InfoCurso { ... };  
  
    struct InfoEstudiante {  
        Estudiante id_est;  
        std::string nombre;  
        std::string apellidos;  
  
        InfoEstudiante(const Estudiante &id_est,  
                       const std::string &nombre, const std::string &apellidos);  
    };  
  
    std::unordered_map<Curso, InfoCurso> cursos;  
    std::unordered_map<Estudiante, InfoEstudiante> estudiantes;  
}
```



# Cambios

- `estudiantes_matriculados(curso)`

Debemos obtener los *nombres y apellidos*.

- Registro InfoCurso.

Ahora se almacenan los NIFs de los/as estudiantes matriculados/as en InfoCurso.

El conjunto de estudiantes matriculados puede ser `unordered_set`.

- 

- `matricular_en_curso(id_est, curso)`

Ahora es necesario comprobar si el estudiante está registrado.

Pasa a tener coste  $O(1)$ .

# Obtener estudiantes matriculados

```
class Academia {  
public:  
    std::vector<std::string> estudiantes_matriculados(const Curso &curso) const {  
        const InfoCurso &info_curso = buscar_curso(curso);  
        std::vector<std::string> result;  
        for (const Estudiante &id_est: info_curso.estudiantes) {  
            const InfoEstudiante &info_est = estudiantes.at(id_est);  
            result.push_back(info_est.apellidos + ", " + info_est.nombre);  
        }  
  
        std::sort(result.begin(), result.end());  
        return result;  
    }  
    ...  
private:  
    ...  
    std::unordered_map<Curso, InfoCurso> cursos;  
}
```

# Obtener cursos de un estudiante

```
class Academia {  
public:  
    std::vector<std::string> cursos_estudiante(const Estudiante &id_est) const {  
        std::vector<std::string> result;  
  
        for (auto entrada: cursos) {  
            const InfoCurso &info_curso = entrada.second;  
            if (info_curso.estudiantes.contains(id_est)) {  
                result.push_back(info_curso.nombre);  
            }  
        }  
        sort(result.begin(), result.end());  
        return result;  
    }  
    ...  
private:  
    ...  
    std::unordered_map<Curso, InfoCurso> cursos;  
}
```



# Cursos matriculados por cada estudiante

```
class Academia {  
public:  
    ...  
  
private:  
    struct InfoCurso { ... };  
  
    struct InfoEstudiante {  
        Estudiante id_est;  
        std::string nombre;  
        std::string apellidos;  
  
        std::set<std::string> cursos;  
  
        InfoEstudiante(const Estudiante &id_est,  
                       const std::string &nombre, const std::string &apellidos);  
    };  
  
    ...  
}
```



# Obtener estudiantes matriculados (cambios)

```
class Academia {  
public:  
    std::vector<std::string> cursos_estudiante(const Estudiante &id_est) const {  
        const InfoEstudiante &info_est = buscar_estudiante(id_est);  
        std::vector<std::string> result;  
        copy(info_est.cursos.begin(), info_est.cursos.end(),  
             std::back_insert_iterator<std::vector<std::string>>(result));  
  
        return result;  
    }  
    ...  
private:  
    ...  
    std::unordered_map<Curso, InfoCurso> cursos;  
}
```



# Matrícula en un curso (cambios)

```
class Academia {  
public:  
    void matricular_en_curso(const Estudiante &est, const Curso &curso) {  
        InfoCurso &info_curso = buscar_curso(curso);  
        InfoEstudiante &info_est = buscar_estudiante(est);  
        if (info_curso.estudiantes.contains(est)) {  
            throw std::domain_error("estudiante ya matriculado");  
        }  
  
        if (info_curso.estudiantes.size() ≥ info_curso.numero_plazas) {  
            throw std::domain_error("no hay plazas disponibles");  
        }  
  
        info_curso.estudiantes.insert(est);  
        info_est.cursos.insert(curso);  
    }  
...  
private:  
...  
    std::unordered_map<Curso, InfoCurso> cursos;  
}
```



# Eliminar un curso (cambios)

```
class Academia {  
public:  
    void eliminar_curso(const Curso &curso) {  
        auto it = cursos.find(curso);  
        if (it != cursos.end()) {  
            InfoCurso &info_curso = it->second;  
            for (Estudiante id_est : info_curso.estudiantes) {  
                estudiantes.at(id_est).cursos.erase(curso);  
            }  
            cursos.erase(it);  
        }  
    }  
    ...  
private:  
    ...  
    std::unordered_map<Curso, InfoCurso> cursos;  
}
```



ESTRUCTURAS DE DATOS

APLICACIONES DE TIPOS ABSTRACTOS DE DATOS

# Gestión de una academia (3)

Manuel Montenegro Montes

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

# **Lista de espera**

# Requisitos

- Cada curso tiene una lista de espera.
- Si un estudiante se matricula en un curso y no hay plazas disponibles, se le pone en lista de espera.
- Cuando un estudiante se da de baja en un curso, se matricula automáticamente al primero de la lista de espera (si existe).
- Operaciones añadidas o modificadas:
  - Dar de baja a un estudiante.
  - La operación de matrícula de un curso devuelve un booleano indicando si el estudiante ha sido matriculado o está en lista de espera.

# Lista de espera en cursos

```
class Academia {  
public:  
    ...  
  
private:  
    struct InfoCurso {  
        std::string nombre;  
        int numero_plazas;  
        std::unordered_set<Estudiante> estudiantes;  
        std::queue<Estudiante> lista_espera;  
  
        InfoCurso(const std::string &nombre,  
                  int numero_plazas);  
    };  
  
    std::unordered_map<Curso, InfoCurso> cursos;  
}
```



# Matrícula en un curso (cambios)

```
class Academia {  
public:  
    bool matricular_en_curso(const Estudiante &est, const Curso &curso) {  
        InfoCurso &info_curso = buscar_curso(curso);  
        InfoEstudiante &info_est = buscar_estudiante(est);  
        if (info_curso.estudiantes.contains(est)) {  
            throw std::domain_error("estudiante ya matriculado");  
        }  
  
        if (info_curso.estudiantes.size() < info_curso.numero_plazas) {  
            info_curso.estudiantes.insert(est);  
            info_est.cursos.insert(curso);  
            return true;  
        } else {  
            info_curso.lista_espera.push(est);  
            return false;  
        }  
    }  
    ...  
private:  
    ...  
    std::unordered_map<Curso, InfoCurso> cursos;  
}
```



# Darse de baja en un curso

```
class Academia {  
public:  
    void dar_de_baja_en_curso(const Estudiante &id_est, const Curso &nombre_curso) {  
        InfoCurso &curso = buscar_curso(nombre_curso);  
  
        auto it_estudiante = curso.estudiantes.find(id_est);  
        if (it_estudiante != curso.estudiantes.end()) {  
            curso.estudiantes.erase(it_estudiante);  
            it_estudiante->cursos.erase(curso.nombre);  
  
            while (!curso.lista_espera.empty() && curso.estudiantes.size() < curso.numero_plazas) {  
                const Estudiante &nif_primer = curso.lista_espera.front();  
                curso.lista_espera.pop();  
                if (!curso.estudiantes.contains(nif_primer)) {  
                    curso.estudiantes.insert(nif_primer);  
                    estudiantes.at(nif_primer).cursos.insert(curso.nombre);  
                }  
            }  
        }  
    }  
...  
private:  
    ...  
    std::unordered_map<Curso, InfoCurso> cursos;  
}
```

ESTRUCTURAS DE DATOS

APLICACIONES DE TIPOS ABSTRACTOS DE DATOS

# Líneas de metro

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

# Requisitos

- Queremos implementar un sistema de gestión de líneas de metro.
- Contendrá información sobre líneas del suburbano, paradas disponibles en cada línea, y horarios de salida de trenes en cada línea.

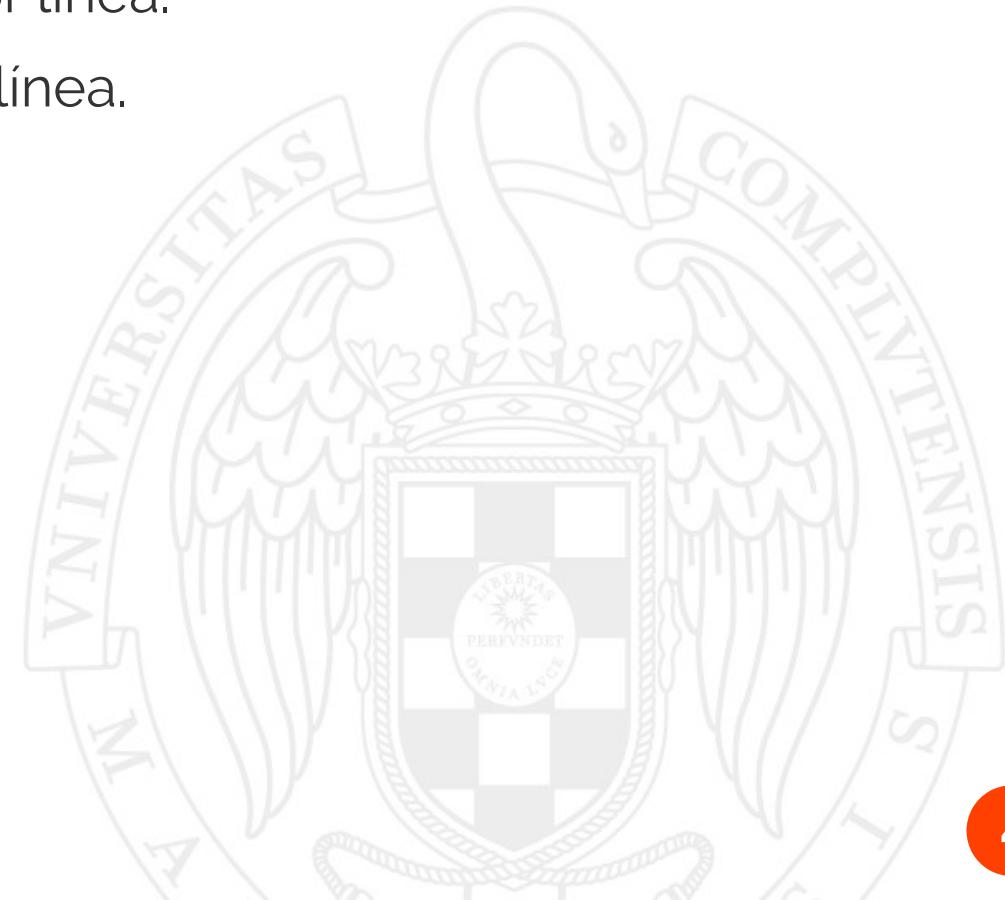


# Operaciones

- Crear un sistema de líneas de metro vacío.
- Añadir una nueva línea de metro.
- Añadir una parada a una nueva línea.
  - Se indicará el tiempo de recorrido (en segundos) desde la parada anterior (cero si es la primera parada).
- Añadir una nueva hora de salida en una línea (hora de salida desde cabecera).
- Obtener el número de trenes que salen diariamente en una línea.
- Obtener el tiempo de espera hasta el próximo tren en una parada determinada.

# Métricas de coste

- $L$  = Número de líneas.
- $P$  = Número de paradas máximo por línea.
- $T$  = Número máximo de trenes por línea.



# TAD de gestión de horarios

# Interfaz del TAD Hora

```
class Hora {  
public:  
    Hora(int horas, int minutos, int segundos);  
    int horas() const;  
    int minutos() const;  
    int segundos() const;  
  
    Hora operator+(int segs) const;  
    Hora operator-(int segs) const;  
    int operator-(const Hora& otra) const;  
  
    bool operator==(const Hora &otra) const;  
    bool operator<(const Hora &otra) const;  
  
private:  
    ...  
};
```



# Representación del TAD Hora

```
class Hora {  
public:  
    ...  
  
private:  
    int num_segundos;  
  
    Hora(int num_segundos);  
};
```

Segundos transcurridos desde  
la hora 00:00:00

# Representación del TAD Hora

```
class Hora {  
public:  
    ...  
  
private:  
    int num_segundos;  
  
    Hora(int num_segundos);  
};
```

Constructor privado

# Implementación del TAD Hora

```
class Hora {  
public:  
  
    Hora(int horas, int minutos, int segundos): num_segundos(horas * 3600 + minutos * 60 + segundos) {  
  
        if (horas < 0 || minutos < 0 || minutos ≥ 60 || segundos < 0 || segundos ≥ 60) {  
            throw std::domain_error("hora no válida");  
        }  
  
    }  
  
    int horas() const { return num_segundos / 3600; }  
    int minutos() const { return (num_segundos / 60) % 60; }  
    int segundos() const { return num_segundos % 60; }  
  
private:  
    ...  
};
```

# Implementación del TAD Hora

```
class Hora {  
public:  
  
...  
  
Hora operator+(int segs) const {  
    return Hora(num_segundos + segs);  
}  
  
int operator-(const Hora& otra) const {  
    return num_segundos - otra.num_segundos;  
}  
  
Hora operator-(int segs) const {  
    return Hora(num_segundos - segs);  
}  
  
private:  
...  
};
```



# Implementación del TAD Hora

```
class Hora {  
public:  
  
...  
  
    bool operator==(const Hora &otra) const {  
        return num_segundos == otra.num_segundos;  
    }  
  
    bool operator<(const Hora &otra) const {  
        return num_segundos < otra.num_segundos;  
    }  
  
private:  
...  
};
```



# TAD de gestión de líneas de metro

# Interfaz

```
class Metro {  
public:  
    Metro();  
  
    void nueva_linea(const Linea &nombre);  
  
    void nueva_parada(const Linea &nombre, const Parada &nueva_parada, int tiempo_desde_anterior);  
  
    void nuevo_tren(const Linea &nombre, const Hora &hora_salida);  
  
    int numero_trenes(const Linea &nombre) const;  
  
    int tiempo_proximo_tren(const Linea &linea, const Parada &parada, const Hora &hora_actual);  
  
private:  
    ...  
};
```

# Colección de líneas

- Guardamos, para cada línea, la siguiente información:
  - Nombre (o número) de línea, que la identifica.
  - Paradas de esa línea.
  - Horarios de salida de esa línea.
- Cada operación del TAD Metro necesita acceder a la información de una línea. Necesitamos acceso rápido a esa información.
- Solución: diccionario que asocia nombres de líneas con información de cada línea.

# Representación

```
using Linea = std::string;

class Metro {
public:
    ...

private:
    struct InfoLinea { ... };
    std::unordered_map<Linea, InfoLinea> lineas;
};
```



# Colección de líneas

- InfoLinea debe contener:
  - Nombre (o número) de línea.
  - Paradas de esa línea.
  - Horarios de salida de esa línea.
- ¿Cómo almacenamos la colección de paradas?
  - Existe un orden entre las paradas; viene dado por el orden en el que las inserte.
  - Tenemos que recorrer las paradas hasta una determinada posición.

# Colección de líneas

- InfoLinea debe contener:
  - Nombre (o número) de línea.
  - Paradas de esa línea.
  - Horarios de salida de esa línea.
- ¿Cómo almacenamos la colección de horarios de salida?
  - Existe un orden entre los horarios, pero no viene dado por el orden en el que se inserten.
  - Necesitamos acceso eficiente al tren que sale después de una determinada hora.

# Representación

```
using Linea = std::string;

class Metro {
public:
    ...

private:
    struct InfoLinea {
        Linea nombre;
        std::set<Hora> salida_trenes;
        std::list<InfoParada> paradas;

        InfoLinea(const Linea &nombre): nombre(nombre) { }

    };

    std::unordered_map<Linea, InfoLinea> lineas;
};
```



# Representación

```
using Parada = std::string;

class Metro {
public:
    ...
private:
    struct InfoParada {
        Parada nombre;
        int tiempo_desde_anterior;

        InfoParada(const Parada &nombre, int tiempo_desde_anterior);
    };

    struct InfoLinea { ... };
    std::unordered_map<Linea, InfoLinea> lineas;
};
```



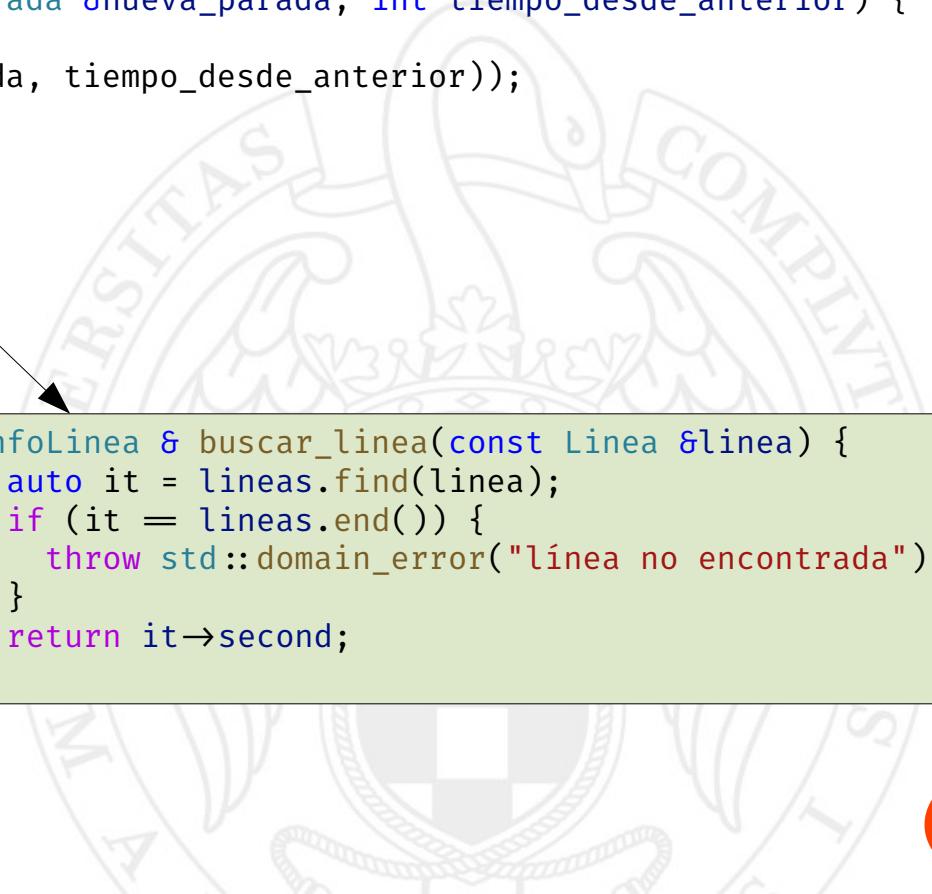
# Añadir una nueva línea

```
class Metro {  
public:  
  
    void nueva_linea(const Linea &nombre) {  
        if (lineas.contains(nombre)) {  
            throw std::domain_error("linea ya existente");  
        }  
        lineas.insert({nombre, InfoLinea(nombre)});  
    }  
  
private:  
    ...  
    std::unordered_map<Linea, InfoLinea> lineas;  
};
```



# Añadir una nueva parada

```
class Metro {  
public:  
  
    void nueva_parada(const Linea &nombre, const Parada &nueva_parada, int tiempo_desde_anterior) {  
        InfoLinea &linea = buscar_linea(nombre);  
        linea.paradas.push_back(InfoParada(nueva_parada, tiempo_desde_anterior));  
    }  
  
private:  
    ...  
    std::unordered_map<Linea, InfoLinea> lineas;  
};
```



```
InfoLinea & buscar_linea(const Linea &linea) {  
    auto it = lineas.find(linea);  
    if (it == lineas.end()) {  
        throw std::domain_error("línea no encontrada");  
    }  
    return it->second;  
}
```

# Añadir un nuevo horario de salida

```
class Metro {  
public:  
  
    void nuevo_tren(const Linea &nombre, const Hora &hora_salida) {  
        InfoLinea &linea = buscar_linea(nombre);  
        linea.salida_trenes.insert(hora_salida);  
    }  
  
    int numero_trenes(const Linea &nombre) const {  
        const InfoLinea &linea = buscar_linea(nombre);  
        return linea.salida_trenes.size();  
    }  
  
private:  
    ...  
    std::unordered_map<Linea, InfoLinea> lineas;  
};
```



# Tiempo hasta el próximo tren

- Necesitamos un método auxiliar que calcule el tiempo de trayecto desde la cabecera de línea hasta una parada dada.

```
int buscar_parada(const InfoLinea &info_linea, const Parada &parada) {  
    int segs_desde_cabecera = 0;  
  
    auto it = info_linea.paradas.begin();  
    while (it != info_linea.paradas.end() && it->nombre != parada) {  
        segs_desde_cabecera += it->tiempo_desde_anterior;  
        ++it;  
    }  
  
    if (it == info_linea.paradas.end()) {  
        throw std::domain_error("parada no encontrada");  
    }  
  
    segs_desde_cabecera += it->tiempo_desde_anterior;  
  
    return segs_desde_cabecera;  
}
```

# Tiempo hasta el próximo tren

```
class Metro {  
public:  
  
    int tiempo_proximo_tren(const Linea &linea, const Parada &parada, const Hora &hora_actual) {  
        const InfoLinea &info_linea = buscar_linea(linea);  
        int segs_desde_cabecera = buscar_parada(info_linea, parada);  
        Hora hora_salida = hora_actual - segs_desde_cabecera;  
  
        auto it = info_linea.salida_trenes.lower_bound(hora_salida);  
        if (it == info_linea.salida_trenes.end()) {  
            return -1;  
        }  
  
        const Hora &hora_salida_siguiente = *it;  
        const Hora &hora_parada_siguiente = hora_salida_siguiente + segs_desde_cabecera;  
        return hora_parada_siguiente - hora_actual;  
    }  
  
private:  
    ...  
    std::unordered_map<Linea, InfoLinea> lineas;  
};
```

# Representación alternativa

# Representación alternativa

- En lugar de almacenar el tiempo de recorrido desde la parada anterior, podemos almacenar el tiempo desde la cabecera de línea.
- Podemos cambiar la lista de paradas por un diccionario que asocia cada parada con el tiempo de recorrido desde la cabecera.
- Necesitamos almacenar, para cada línea, el tiempo total de recorrido desde la cabecera hasta la última parada.

```
struct InfoLinea {  
    Linea nombre;  
    std::set<Hora> salida_trenes;  
    std::list<InfoParada> paradas;  
    ...  
};
```

```
struct InfoLinea {  
    Linea nombre;  
    std::set<Hora> salida_trenes;  
    std::list<InfoParada> paradas;  
    int tiempo_total;  
    std::unordered_map<Parada, int> tiempos_desde_cabecera;  
    ...  
};
```

# Añadir una nueva parada (modificado)

```
class Metro {  
public:  
  
    void nueva_parada(const Linea &nombre, const Parada &nueva_parada, int tiempo_desde_anterior) {  
        InfoLinea &linea = buscar_linea(nombre);  
        linea.tiempo_total += tiempo_desde_anterior;  
        linea.tiempos_desde_cabecera.insert({nueva_parada, linea.tiempo_total});  
    }  
  
private:  
    ...  
    std::unordered_map<Linea, InfoLinea> lineas;  
};
```

# Tiempo hasta el próximo tren

```
class Metro {  
public:  
  
    int tiempo_proximo_tren(const Linea &linea, const Parada &parada, const Hora &hora_actual) {  
        const InfoLinea &info_linea = buscar_linea(linea);  
        int segs_desde_cabecera = buscar_parada(info_linea, parada);  
        Hora hora_salida = hora_actual - segs_desde_cabecera;  
  
        auto it = info_linea.salida_trenes.lower_bound(hora_salida);  
        if (it == info_linea.salida_trenes.end()) {  
            return -1;  
        }  
  
        const Hora &hora_salida_siguiente = *it;  
        const Hora &hora_parada_siguiente = hora_salida_siguiente + segs_desde_cabecera;  
        return hora_parada_siguiente - hora_actual;  
    }  
  
private:  
    ...  
    std::unordered_map<Linea, InfoLinea> lineas;  
};
```

# Tiempo hasta el próximo tren

```
class Metro {  
public:  
  
    int tiempo_proximo_tren(const Linea &linea, const Parada &parada, const Hora &hora_actual) {  
        const InfoLinea &info_linea = buscar_linea(linea);  
        int segs_desde_cabecera = buscar_parada(info_linea, parada);  
        Hora hora_salida = hora_actual - segs_desde_cabecera;  
  
        auto it = info_linea.salida_trenes.lower_bound(hora_salida);  
        if (it == info_linea.salida_trenes.end()) {  
            return -1;  
        }  
  
        const Hora &hora_salida_s  
        const Hora &hora_parada_s  
        return hora_parada_sigui  
    }  
  
private:  
    ...  
    std::unordered_map<Linea, InfoLinea> lineas,  
};
```

```
int buscar_parada(const InfoLinea &info_linea, const Parada &parada) {  
    auto it = info_linea.tiempos_desde_cabecera.find(parada);  
    if (it == info_linea.tiempos_desde_cabecera.end()) {  
        throw std::domain_error("parada no encontrada");  
    }  
    return it->second;  
}
```

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Atributos y Métodos

Manuel Montenegro Montes

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

# Definición de una clase



# Definición de una clase: atributos

```
class Fecha {  
    int dia;  
    int mes;  
    int anyo;  
};
```



# Definición de una clase: métodos

```
class Fecha {  
    int dia;  
    int mes;  
    int anyo;  
  
    int get_dia();  
    void set_dia(int dia);  
    int get_mes();  
    void set_mes(int mes);  
    int get_anyo();  
    void set_anyo(int anyo);  
};
```

- Aquí se están **declarando** los métodos, pero no aparecen sus **implementaciones**.
- Por defecto, todos los atributos y métodos son privados.

# Modificadores de acceso

```
class Fecha {  
    int dia;  
    int mes;  
    int anyo;  
  
public:  
    int get_dia();  
    void set_dia(int dia);  
    int get_mes();  
    void set_mes(int mes);  
    int get_anyo();  
    void set_anyo(int anyo);  
};
```

- Hay tres tipos de modificadores:
  - **public**:
  - **private**:
  - **protected**:
- Afectan a los métodos y atributos situados a continuación del modificador.

# Modificadores de acceso

```
class Fecha {  
public:  
    int get_dia();  
    void set_dia(int dia);  
    int get_mes();  
    void set_mes(int mes);  
    int get_anyo();  
    void set_anyo(int anyo);  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```

- Hay tres tipos de modificadores:
  - public:
  - private:
  - protected:
- Afectan a los métodos y atributos situados a continuación del modificador.

# Implementación de métodos

```
class Fecha {  
public:  
    int get_dia() {  
        return dia;  
    }  
  
    void set_dia(int dia) {  
        this→dia = dia;  
    }  
  
    // Igualmente para mes y año  
    // ...  
  
private:  
    // ...  
};
```

- Posibilidad 1: Implementación dentro de la definición de clase.
- “Estilo Java”



# Implementación de métodos

```
class Fecha {  
public:  
    int get_dia();  
    void set_dia(int dia);  
    //  
    // ...  
private:  
    // ...  
};  
  
int Fecha::get_dia() {  
    return dia;  
}  
  
void Fecha::set_dia(int dia) {  
    this->dia = dia;  
}
```

- Posibilidad 2: Implementación fuera de la definición de clase.



# ¡No son equivalentes!

- Implementaciones dentro de la clase: se consideran métodos **inline**.

<https://www.geeksforgeeks.org/inline-functions-cpp/>

- Son más eficientes, pero incrementan el tamaño del código.

- **Consejo:**

- Métodos cortos (p.ej. acceso, modificación) pueden definirse dentro de la clase.
  - Métodos largos deben definirse fuera de la clase.

# **Uso de una clase: instancias**



# Creación de instancias

```
int main() {  
    Fecha f;  
    f.set_dia(28);  
    f.set_mes(8);  
    f.set_anyo(2019);  
  
    std::cout << "Día: " << f.get_dia() << std::endl;  
    std::cout << "Mes: " << f.get_mes() << std::endl;  
    std::cout << "Año: " << f.get_anyo() << std::endl;  
}
```

Día: 28  
Mes: 8  
Año: 2019

# **Salida con formato**



# Un nuevo método: imprimir

```
class Fecha {  
public:  
    int get_dia();  
    void set_dia(int dia);  
    int get_mes();  
    void set_mes(int mes);  
    int get_anyo();  
    void set_anyo(int anyo);  
  
    void imprimir();  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```



# Un nuevo método: imprimir

```
void Fecha::imprimir() {  
    std::cout << dia << "/" << mes << "/" << anyo;  
}
```

# Un nuevo método: imprimir

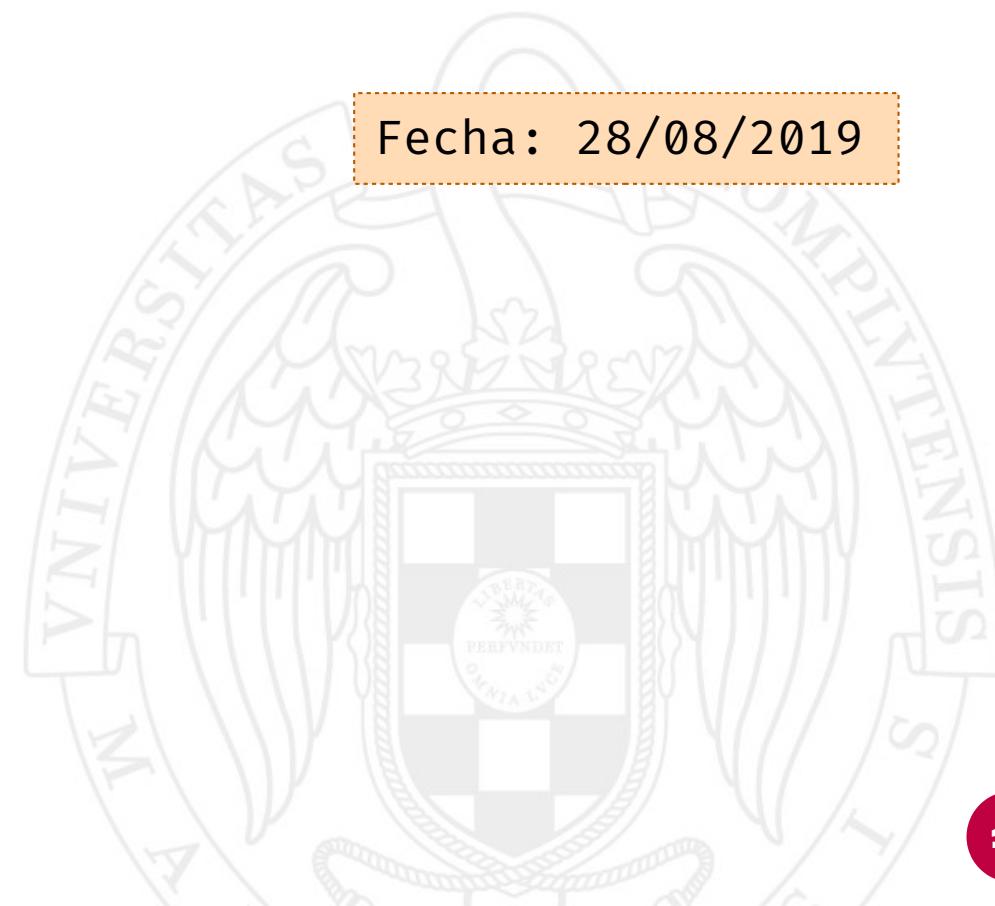
```
void Fecha::imprimir() {  
    std::cout << std::setfill('0') << std::setw(2) << dia << "/"  
        << std::setw(2) << mes << "/"  
        << std::setw(4) << anyo;  
}
```

```
#include <iostream>  
#include <iomanip>
```

# Uso del método imprimir

```
int main() {  
    Fecha f;  
    f.set_dia(28);  
    f.set_mes(8);  
    f.set_anyo(2019);  
  
    std::cout << "Fecha: ";  
    f.imprimir();  
    std::cout << std::endl;  
}
```

Fecha: 28/08/2019



ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Constructores Listas de Inicialización

Manuel Montenegro Montes

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

# Recordatorio: clase Fecha

```
class Fecha {  
public:  
    int get_dia();  
    void set_dia(int dia);  
    int get_mes();  
    void set_mes(int mes);  
    int get_anyo();  
    void set_anyo(int anyo);  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```



# Recordatorio: clase Fecha

```
int main() {  
    Fecha f;  
    f.set_dia(28);  
    f.set_mes(8);  
    f.set_anyo(2019);  
  
    std::cout << "Fecha: ";  
    f.imprimir();  
    std::cout << std::endl;  
}
```

- Hemos inicializado los atributos del objeto tras su creación, mediante los métodos set.
- ¿Y si se me hubiera olvidado llamar a estos métodos?
- **¿Existe alguna manera de asegurarnos de que el objeto está inicializado tras su creación?**
- Sí: **constructores**

# Tipos de constructores

- **Constructor por defecto** (sin parámetros).
- **Constructor paramétrico.**
- **Constructor de copia.**
- **Constructor *move*.**
- **Constructor de conversión.**



# Constructor por defecto



# Constructor por defecto

```
class Fecha {  
public:  
    Fecha() {  
        dia = 1;  
        mes = 1;  
        anyo = 1900;  
    }  
  
    // ...  
private:  
    int dia;  
    int mes;  
    int anyo;  
}
```

- Todos los constructores tienen el mismo nombre que la clase.
- No tienen tipo de retorno.
- El **constructor por defecto** no tiene parámetros.

# Constructor por defecto

```
class Fecha {  
public:  
    Fecha();  
    // ...  
private:  
    int dia;  
    int mes;  
    int anyo;  
}  
  
Fecha::Fecha() {  
    dia = 1;  
    mes = 1;  
    anyo = 1900;  
}
```

- Otra posibilidad: definir la implementación fuera de la clase.



# Uso del constructor por defecto

```
int main() {  
    Fecha f;  
    f.imprimir();  
}
```

01/01/1900



# Constructor con parámetros

# Constructor con parámetros

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo) {  
        this->dia = dia;  
        this->mes = mes;  
        this->anyo = anyo;  
    }  
  
    // ...  
private:  
    int dia;  
    int mes;  
    int anyo;  
}
```



# Sobrecarga de constructores

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo) {  
        this->dia = dia;  
        this->mes = mes;  
        this->anyo = anyo;  
    }  
  
    Fecha(int anyo) {  
        this->dia = 1;  
        this->mes = 1;  
        this->anyo = anyo;  
    }  
    // ...  
private:  
    int dia;  
    int mes;  
    int anyo;  
}
```



# Delegación de constructores

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo) {  
        this->dia = dia;  
        this->mes = mes;  
        this->anyo = anyo;  
    }  
  
    Fecha(int anyo): Fecha(1, 1, anyo) {  
        // vacío  
    }  
  
    // ...  
private:  
    int dia;  
    int mes;  
    int anyo;  
}
```



# Uso del constructor con parámetros

```
int main() {  
    Fecha f;  
    f.imprimir();  
  
    return 0;  
}
```

Error: no hay constructor por defecto



# Uso del constructor con parámetros

```
int main() {  
    Fecha f1(28, 8, 2019);  
    Fecha f2(2019);  
  
    f1.imprimir();  
    std::cout << " ";  
    f2.imprimir();  
  
    return 0;  
}
```

28/08/2019 01/01/2019



# Uso del constructor con parámetros

```
int main() {  
    Fecha f1 = {28, 8, 2019};  
    Fecha f2 = {2019};  
  
    f1.imprimir();  
    std::cout << " "  
    f2.imprimir();  
  
    return 0;  
}
```

Sintaxis alternativa

# Paso de objetos a funciones

```
bool es_navidad(Fecha f) {
    return f.get_dia() == 25 && f.get_mes() == 12;
}

int main() {
    Fecha f = {25, 12, 2019};
    if (es_navidad(f)) {
        std::cout << "Feliz navidad!" << std::endl;
    }
    return 0;
}
```



# Paso de objetos a funciones

```
bool es_navidad(Fecha f) {  
    return f.get_dia() = 25 && f.get_mes() = 12;  
}  
  
int main() {  
    if (es_navidad({25, 12, 2019})) {  
        std::cout << "Feliz navidad!" << std::endl;  
    }  
    return 0;  
}
```

Creación de objeto en el argumento

# Listas de inicialización

# Una nueva clase: Persona

```
class Persona {  
  
private:  
    std::string nombre;  
    Fecha fecha_nacimiento;  
};
```

```
int main() {  
    Persona p;  
    ...  
}
```

El constructor por defecto no  
puede inicializar fecha\_nacimiento

# Añadiendo un constructor a Persona

```
class Persona {  
public:  
    Persona(std::string nombre, int dia, int mes, int anyo) {  
        this->nombre = nombre;  
        ... ???  
    }  
  
private:  
    std::string nombre;  
    Fecha fecha_nacimiento;  
};
```

- ¿Cómo indico que quiero llamar al constructor de Fecha pasándole dia, mes y anyo?

# Llamando al constructor de Fecha

```
class Persona {  
public:  
    Persona(std::string nombre, int dia, int mes, int anyo)  
        : fecha_nacimiento(dia, mes, anyo) {  
            this->nombre = nombre;  
    }  
  
private:  
    std::string nombre;  
    Fecha fecha_nacimiento;  
};
```

- Al crear el objeto Persona, se llamará al constructor de Fecha con los tres argumentos indicados.

# Llamando al constructor de Fecha

```
class Persona {  
public:  
    Persona(std::string nombre, int dia, int mes, int anyo)  
        : nombre(nombre), fecha_nacimiento(dia, mes, anyo) {}  
  
private:  
    std::string nombre;  
    Fecha fecha_nacimiento;  
};
```

- Podemos utilizar la misma sintaxis con el resto de los atributos.
- A esto se le llama **lista de inicialización**.

# Listas de inicialización

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo) {  
        this->dia = dia;  
        this->mes = mes;  
        this->anyo = anyo;  
    }  
  
    Fecha(int anyo): Fecha(1, 1, anyo) {}  
  
    // ...  
private:  
    int dia;  
    int mes;  
    int anyo;  
}
```



# Listas de inicialización

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo): dia(dia), mes(mes), anyo(anyo) {}  
  
    Fecha(int anyo): Fecha(1, 1, anyo) {}  
  
    // ...  
private:  
    int dia;  
    int mes;  
    int anyo;  
}
```



ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Métodos constantes

Manuel Montenegro Montes

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

# Recordatorio: clase Fecha

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo);  
    Fecha(int anyo);  
  
    int get_dia();  
    void set_dia(int dia);  
    int get_mes();  
    void set_mes(int mes);  
    int get_anyo();  
    void set_anyo(int anyo);  
    void imprimir();  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```



# Paso de objetos por valor

```
bool es_navidad(Fecha f) {  
    return f.get_dia() == 25  
        && f.get_mes() == 12;  
}  
  
int main() {  
    Fecha mi_fecha(25, 12, 2000);  
    if (es_navidad(mi_fecha)) {  
        std::cout << "Feliz navidad!"  
            << std::endl;  
    }  
    return 0;  
}
```

- La función `es_navidad` recibe su argumento **por valor**.
- Al pasar por valor una instancia de una clase se hace una copia del argumento.
  - ¿Cómo? Constructor de copia.
- Si queremos evitar eso, debemos pasar el parámetro **por referencia**.

# Paso de objetos por referencia



# Paso de objetos por referencia

```
bool es_navidad(Fecha &f) {  
    return f.get_dia() == 25  
        && f.get_mes() == 12;  
}  
  
int main() {  
    Fecha mi_fecha(25, 12, 2000);  
    if (es_navidad(mi_fecha)) {  
        std::cout << "Feliz navidad!"  
            << std::endl;  
    }  
    return 0;  
}
```

- Mediante el símbolo & indicamos que el parámetro f se recibe por referencia.
- Con esto se evita hacer una copia de mi\_fecha.
- ¡Ojo! Cualquier cambio que es\_navidad realice en f se reflejará también en mi\_fecha.
  - En este caso, podemos ver que es\_navidad no está alterando el objeto f.

# ¿Y si no conocemos la implementación?

- ¿Cuál de estas dos funciones te inspira más confianza?

```
bool compara(Fecha f1, Fecha f2);
```

```
bool compara(Fecha &f1, Fecha &f2);
```

- La primera garantiza que no va a alterar el estado de los objetos Fecha que reciba, ya que va a trabajar sobre copias de los mismos.
- La segunda no ofrece esa garantía, aunque se ahorra la copia de los argumentos.
- **¿Podemos conseguir los beneficios de ambas versiones?**

# Referencias constantes

```
bool compara(const Fecha &f1, const Fecha &f2);
```

- Una **referencia constante** no permite modificar el estado del objeto apuntado por la referencia.
- El compilador comprueba que `compara` no modifique los atributos de los objetos `f1` y `f2`.
- Con esto:
  - Nos ahorramos copias de los argumentos, porque se pasan por referencia.
  - El que llame a la función `compara` tiene la certeza de que sus objetos no se van a ver modificados.

# Paso de objetos por valor

```
bool es_navidad(const Fecha &f) {  
    return f.get_dia() == 25  
        && f.get_mes() == 12;   
}
```

- Hacemos que la función `es_navidad` reciba su parámetro como referencia constante.
- ... pero el compilador protesta sobre nuestra definición.
- El compilador no sabe si los métodos `get_dia()` o `get_mes()` alteran el estado de `f`.

# Métodos constantes

# Métodos constantes

```
class Fecha {  
public:  
    ...  
    int get_dia() const { return dia; }  
    int get_mes() const { return mes; }  
    int get_anyo() const { return anyo; }  
    void imprimir() const;  
    ...  
  
private:  
    ...  
};
```

- Se declaran añadiendo la palabra **const** tras la lista de parámetros.
- Con esto se indica que el método no altera el estado del objeto.
- El compilador comprueba:
  - que el método no modifique los atributos del objeto.
  - que el método no llame a otros métodos de ese mismo objeto, salvo que también sean constantes.

# Métodos constantes

```
class Fecha {  
public:  
    ...  
    int get_dia() const { return dia; }  
    int get_mes() const { return mes; }  
    int get_anyo() const { return anyo; }  
    void imprimir() const;  
    ...  
  
private:  
    ...  
};  
  
void Fecha::imprimir() const {  
    ...  
}
```

- Si un método se implementa fuera de la clase, es necesario poner **const** tanto en su declaración, como en su implementación.



# Llamadas a métodos constantes

```
bool es_navidad(const Fecha &f) {  
    return f.get_dia() == 25  
        && f.get_mes() == 12; ✓  
}
```

- Si una referencia a un objeto es constante:
  - No podemos modificar sus atributos públicos a través de esa referencia.
  - Solamente podemos llamar a los métodos `const` de esa referencia.

# ¿Qué métodos deben ser const?

- Todos los que no modifiquen el estado del objeto que recibe la llamada al método (`this`).
- Este tipo de métodos reciben el nombre de **observadores**.
- Incluye, entre otros:
  - Métodos de acceso (`get`).
  - Métodos para imprimir el objeto por pantalla o a otro flujo de salida.
  - Métodos de conversión a otro objeto (por ejemplo, `to_string()`).

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Objetos y memoria dinámica

Manuel Montenegro Montes

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

# Regiones de memoria: pila y *heap*



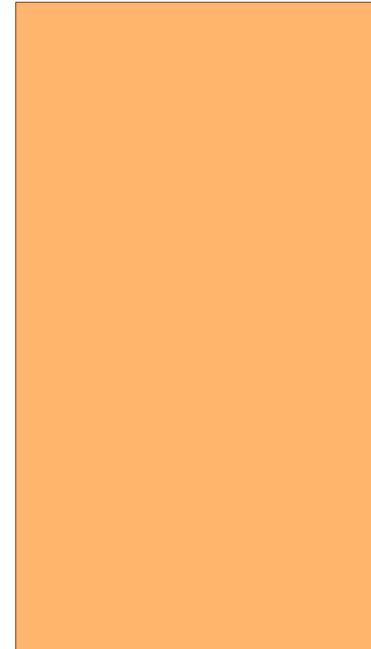
# Regiones de memoria

- **Memoria principal (global)**: variables globales.
  - Se reserva al iniciarse el programa, y se libera al finalizarse.
- **Pila**: variables locales, parámetros.
  - Se reserva y libera a medida que estas variables entran en ámbito y salen de ámbito, respectivamente.
- **Heap**: memoria dinámica.
  - Se reserva y libera manualmente mediante `new` y `delete`.
  - Solamente es accesible a través de punteros.

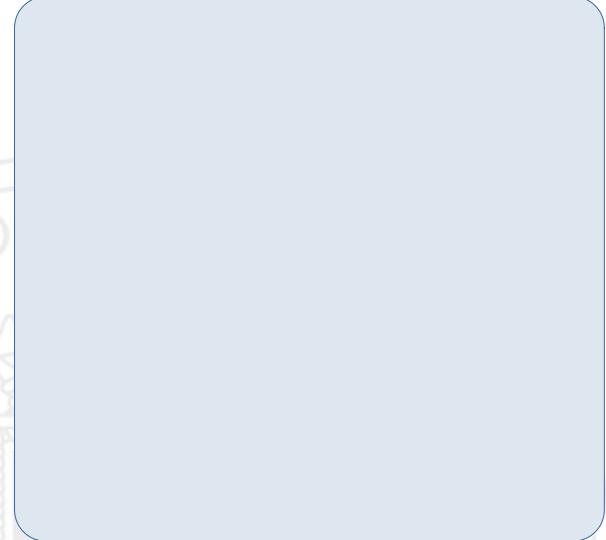
# Regiones de memoria

```
int main() {  
    int x = 3;  
    int *y = new int;  
    *y = 3;  
    int *z = &x;  
  
    delete y;  
    return 0;  
}
```

Pila



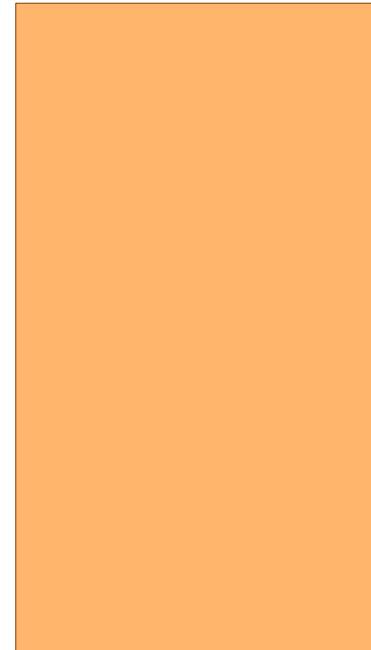
Heap



# Regiones de memoria

```
int main() {  
    int *xs = new int[4];  
    xs[0] = 3;  
    xs[1] = 7;  
  
    int ys[3];  
  
    delete[] xs;  
    return 0;  
}
```

Pila



Heap



# Creación de objetos en el *heap*

# Recordatorio: clase Fecha

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo);  
    Fecha(int anyo);  
  
    int get_dia() const;  
    void set_dia(int dia);  
    int get_mes() const;  
    void set_mes(int mes);  
    int get_anyo() const;  
    void set_anyo(int anyo);  
    void imprimir();  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```



# Creación de instancias en la pila y heap

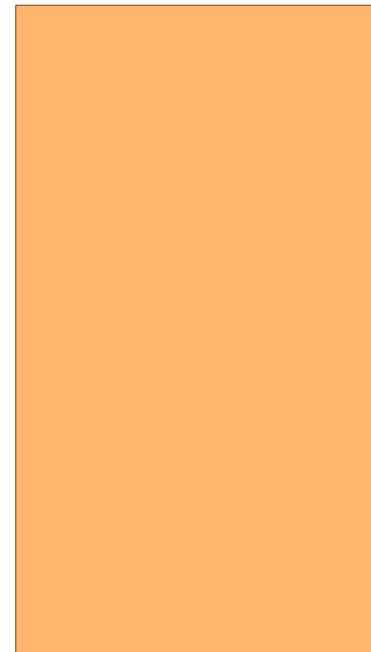
```
int main() {
    Fecha f1(28, 8, 2038);
    Fecha *f2 = new Fecha(10, 6, 2010);

    std::cout << "Fecha 1: ";
    f1.imprimir();
    std::cout << std::endl;

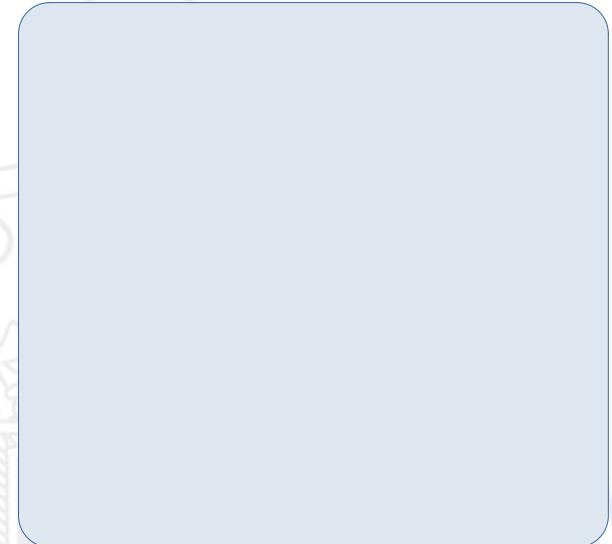
    std::cout << "Fecha 2: ";
    f2->imprimir();
    std::cout << std::endl;

    delete f2;
    return 0;
}
```

Pila



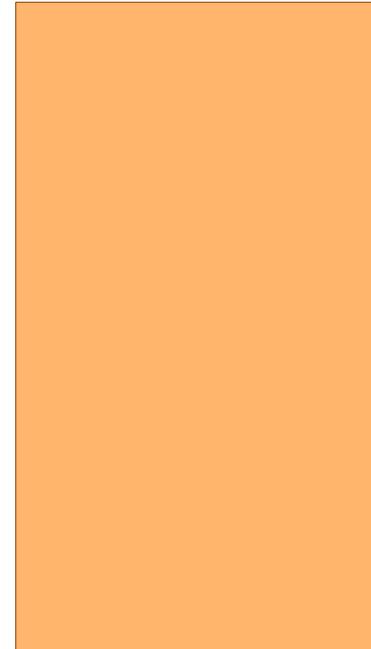
Heap



# Arrays de objetos

```
int main() {  
    Fecha fs[3] =  
        { {2010}, {2011}, {2012} };  
  
    return 0;  
}
```

Pila



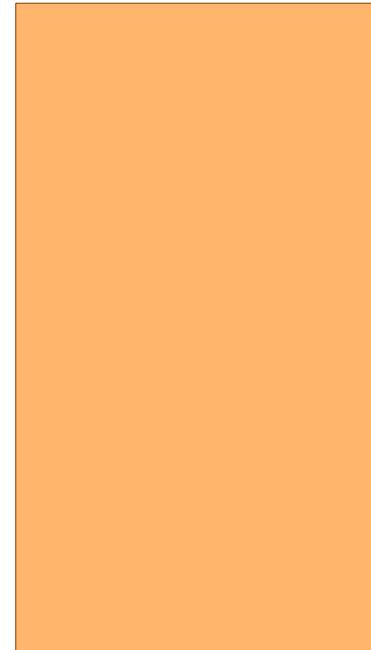
Heap



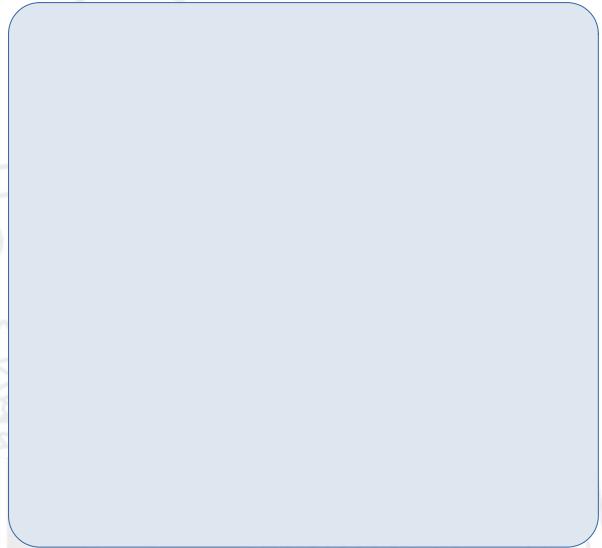
# Arrays de punteros a objetos

```
int main() {  
    Fecha *fs[3];  
    fs[0] = new Fecha(2010);  
    fs[1] = new Fecha(2011);  
    fs[2] = new Fecha(2012);  
  
    delete fs[0];  
    delete fs[1];  
    delete fs[2];  
  
    return 0;  
}
```

Pila



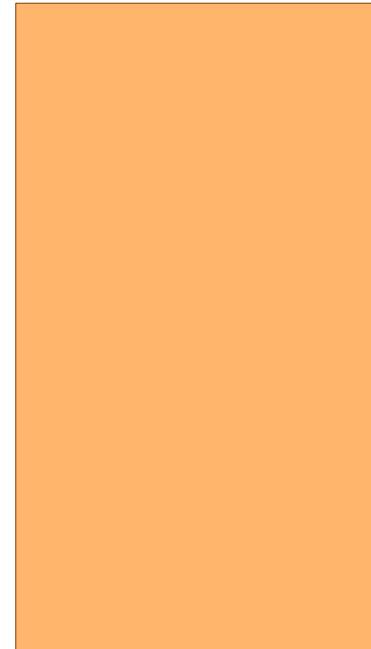
Heap



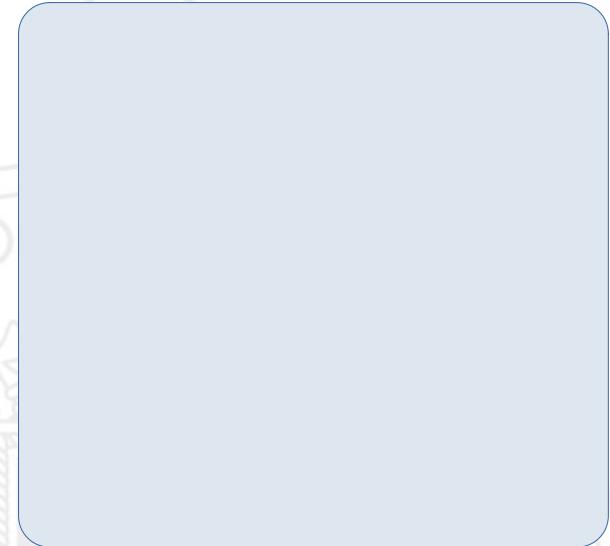
# Arrays dinámicos de punteros a objetos

```
int main() {  
    Fecha **fs = new Fecha*[3];  
    fs[0] = new Fecha(2010);  
    fs[1] = new Fecha(2011);  
    fs[2] = new Fecha(2012);  
  
    delete fs[0];  
    delete fs[1];  
    delete fs[2];  
    delete[] fs;  
  
    return 0;  
}
```

Pila



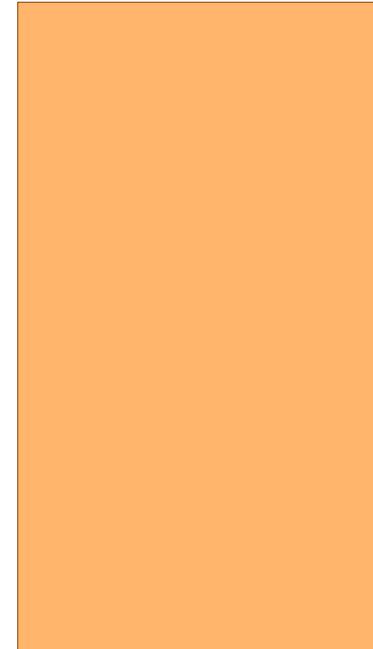
Heap



# Arrays dinámicos de objetos

```
int main() {  
    Fecha *fs = new Fecha[3];  
  
    delete[] fs;  
    return 0;  
}
```

Pila



Heap



# Añadiendo un constructor por defecto

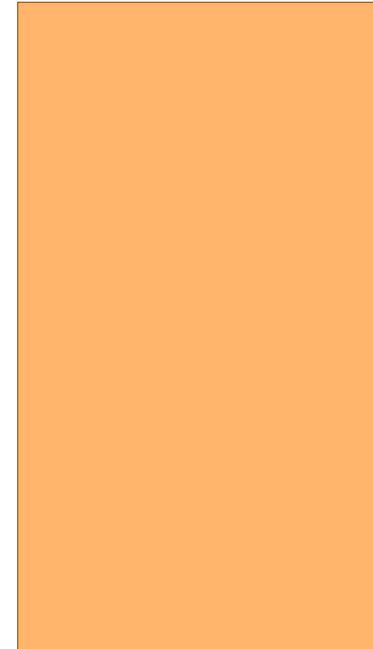
```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo);  
    Fecha(int anyo);  
    Fecha(): Fecha(1, 1, 1900) { }  
  
    int get_dia() const;  
    void set_dia(int dia);  
    int get_mes() const;  
    void set_mes(int mes);  
    int get_anyo() const;  
    void set_anyo(int anyo);  
    void imprimir();  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```



# Arrays dinámicos de objetos

```
int main() {  
    Fecha *fs = new Fecha[3];  
  
    delete[] fs;  
    return 0;  
}
```

Pila



Heap

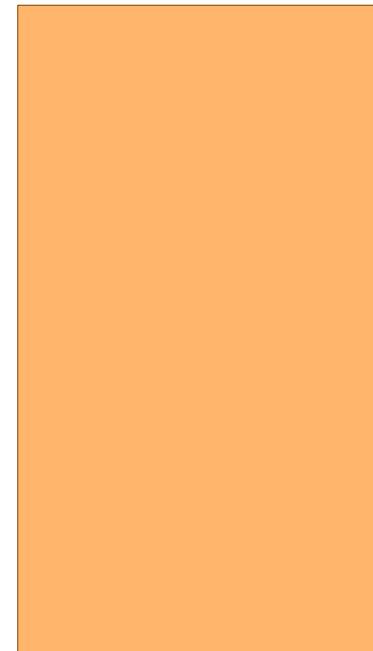


# Compartición de objetos

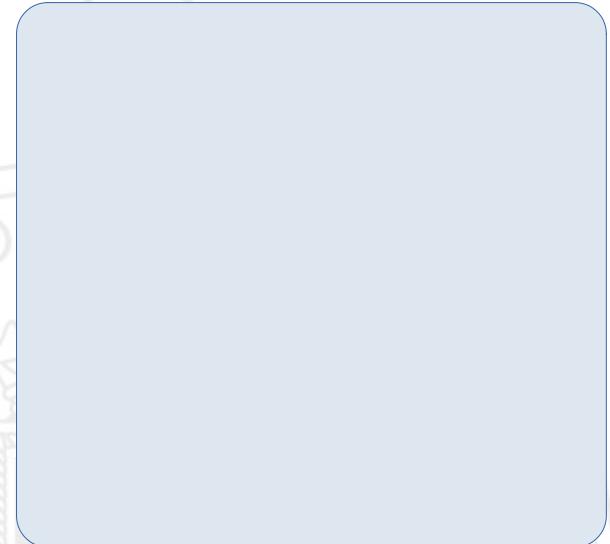
# Compartición de punteros

```
int main() {  
    Fecha *f1 = new Fecha(28, 8, 2019);  
    Fecha *f2 = f1;  
  
    f1→imprimir();  
    f2→imprimir();  
  
    f1→set_dia(1);  
  
    f1→imprimir();  
    f2→imprimir();  
  
    delete f1;  
    // delete f2  
    return 0;  
}
```

Pila



Heap



# Comparación con Java

# Java

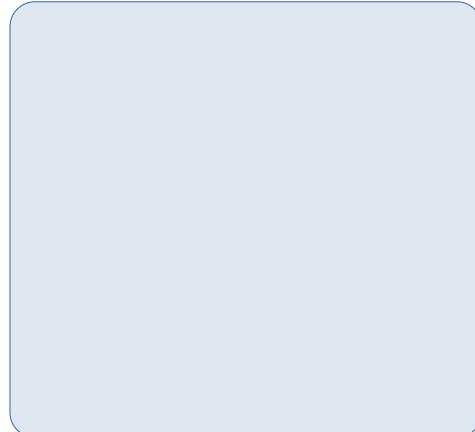
vs

# C++

Pila

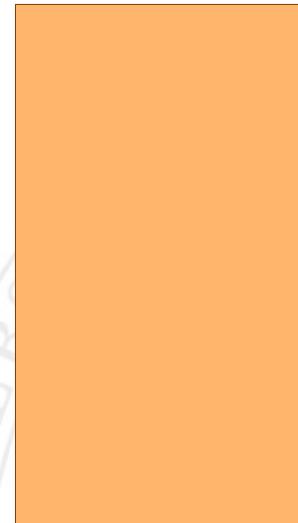


Heap

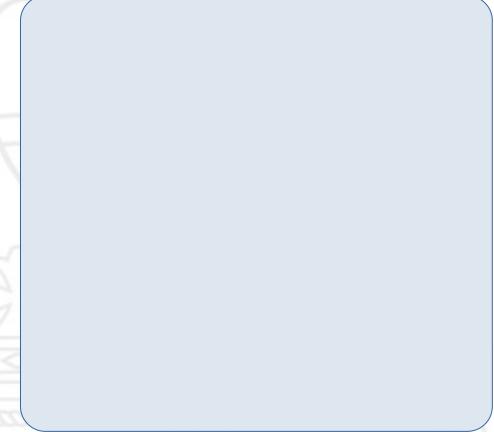


- **Todos los objetos viven en el heap.**
- La pila solo almacena valores básicos o punteros a objetos.

Pila



Heap



- Los objetos pueden almacenarse en el heap o en la pila.

# Manejo de objetos en Java y C++

```
public static void main(String[] args) {  
    Fecha f = new Fecha(20, 3, 2010);  
    f.imprimir();  
}
```

En Java tenemos que crear el objeto *f* en el *heap*, porque todos los objetos se crean allí.

```
int main() {  
    Fecha *f = new Fecha(20, 3, 2010);  
    f->imprimir();  
  
    delete f;  
    return 0;  
}
```

En C++ no es necesario crear el objeto en el *heap*.

# Manejo de objetos en Java y C++

```
public static void main(String[] args) {  
    Fecha f = new Fecha(20, 3, 2010);  
    f.imprimir();  
}
```

En Java tenemos que crear el objeto f en el *heap*, porque todos los objetos se crean allí.

```
int main() {  
    Fecha f(20, 3, 2010);  
    f.imprimir();  
  
    return 0;  
}
```

En C++ es más sencillo crear el objeto f en la pila.

# ¿Cuándo se utiliza el heap en C++?

Lo vamos a utilizar en estas situaciones:

- Cuando el tamaño de un array no es conocido en tiempo de compilación.
- Para estructuras de datos recursivas.
  - Por ejemplo, nodos de árboles y listas enlazadas.

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Destructores

Manuel Montenegro Montes

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

# Recordatorio: clase Fecha

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo);  
    Fecha(int anyo);  
    Fecha();  
  
    int get_dia() const;  
    void set_dia(int dia);  
    int get_mes() const;  
    void set_mes(int mes);  
    int get_anyo() const;  
    void set_anyo(int anyo);  
    void imprimir();  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```



# Recordatorio: clase Persona

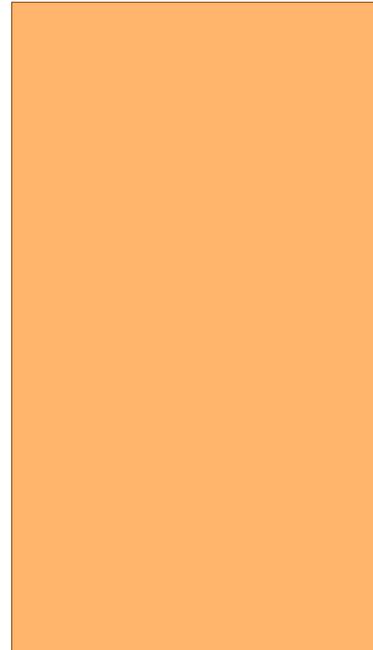
```
class Persona {  
public:  
    Persona(std::string nombre, int dia, int mes, int anyo)  
        : nombre(nombre), fecha_nacimiento(dia, mes, anyo) {}  
  
private:  
    std::string nombre;  
    Fecha fecha_nacimiento;  
};
```



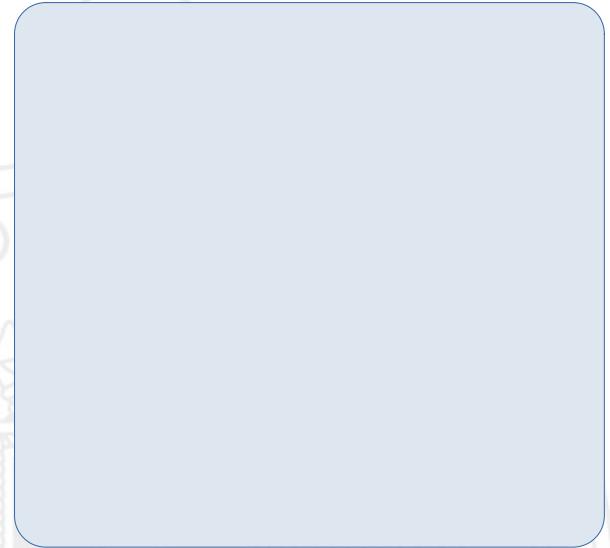
# Ejemplo de uso

```
void ejemplo() {  
    Persona p("David", 15, 3, 1979);  
}
```

Pila



Heap



# Cambio en la representación

```
class Persona {  
public:  
    Persona(std::string nombre, int dia, int mes, int anyo)  
        : nombre(nombre) {  
            this->fecha_nacimiento = new Fecha(dia, mes, anyo);  
    }  
  
private:  
    std::string nombre;  
    Fecha *fecha_nacimiento;  
};
```

# Cambio en la representación

```
class Persona {  
public:  
    Persona(std::string nombre, int dia, int mes, int anyo)  
        : nombre(nombre), fecha_nacimiento(new Fecha(dia, mes, anyo)) {}  
  
private:  
    std::string nombre;  
    Fecha *fecha_nacimiento;  
};
```

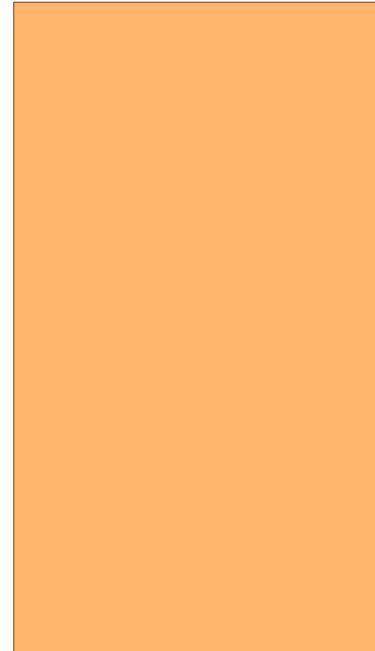


# Ejemplo de uso

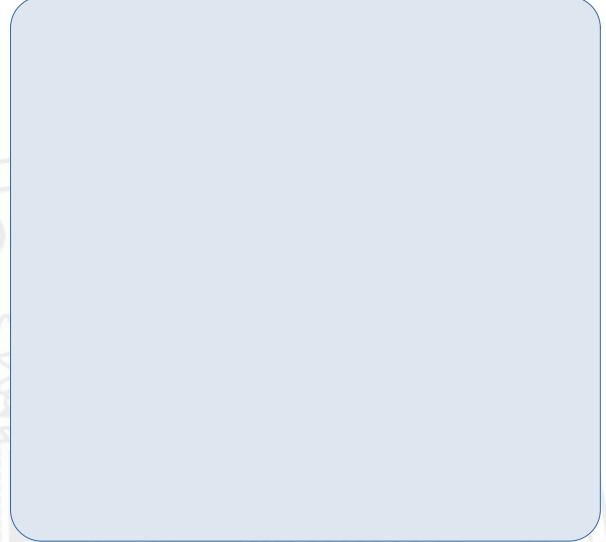
```
void ejemplo() {  
    Persona p("David", 15, 3, 1979);  
}
```

Necesitamos una manera de eliminar el objeto Fecha justo antes de que p salga de ámbito

Pila



Heap



# Destructores en C++

- Un **destructor** es un método especial que es invocado cada vez que el objeto correspondiente se libera.
  - Si el objeto está en la pila, el destructor es invocado cuando la variable que contiene dicho objeto sale de ámbito.
  - Si el objeto está en el *heap*, el destructor es invocado cuando se aplica `delete` sobre el objeto.
- El nombre del método destructor es el mismo que el de la clase en el que está definido, pero anteponiendo el símbolo `~`. 
- El método destructor no tiene ni parámetros, ni tipo de retorno.

# Añadiendo un destructor a Persona

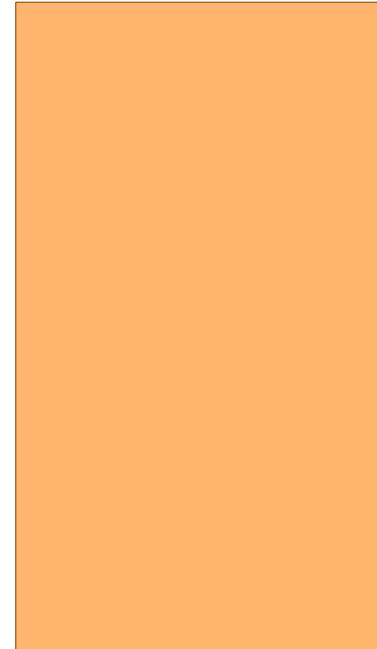
```
class Persona {  
public:  
    Persona(std::string nombre, int dia, int mes, int anyo)  
        : fecha_nacimiento(new Fecha(dia, mes, anyo)) {}  
  
    ~Persona() {  
        delete fecha_nacimiento;  
    }  
  
private:  
    std::string nombre;  
    Fecha *fecha_nacimiento;  
};
```

← Método destructor

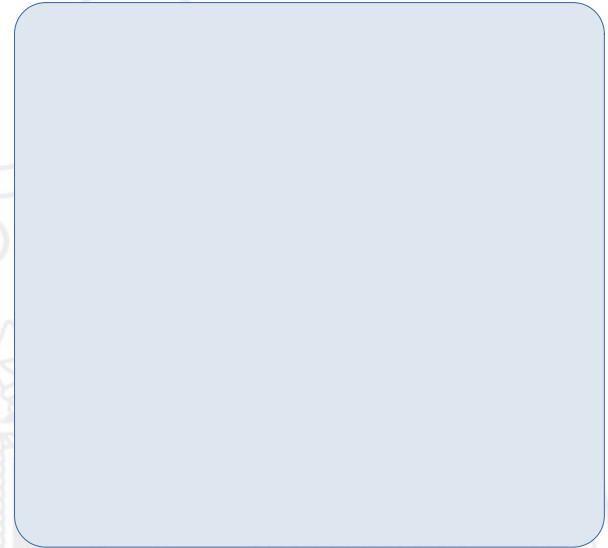
# Ejemplo de uso

```
void ejemplo() {  
    Persona p("David", 15, 3, 1979);  
}
```

Pila



Heap



# ***Resource acquisition is initialization (RAII)***

- En la gran mayoría de casos, la reserva de memoria (`new`) que se realice en el constructor debe tener asociada su liberación (`delete`) en el destructor.
- Excepciones:
  - La memoria reservada se ha liberado antes de invocar el destructor.
  - La memoria reservada está compartida entre varias instancias.
- El principio RAII no solo se aplica a memoria, sino también a otros recursos (apertura/cierre de ficheros, conexiones a bases de datos, etc.)

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Constructores de copia

Manuel Montenegro Montes

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

# Recordatorio: clases Fecha y Persona

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo);  
    Fecha(int anyo);  
    Fecha();  
  
    int get_dia() const;  
    void set_dia(int dia);  
    int get_mes() const;  
    void set_mes(int mes);  
    int get_anyo() const;  
    void set_anyo(int anyo);  
    void imprimir();  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```

```
class Persona {  
public:  
    Persona(std::string nombre,  
            int dia,  
            int mes,  
            int anyo);  
    ~Persona();  
  
    void set_nombre(const std::string &nombre);  
    void set_fecha_nacimiento(int dia,  
                             int mes,  
                             int anyo);  
    void imprimir();  
  
private:  
    std::string nombre;  
    Fecha *fecha_nacimiento;  
};
```

# Ejemplo

```
void modificar_copia(Persona p) {  
    p.set_nombre("Berta");  
    p.set_fecha_nacimiento(10, 10, 2010);  
}  
  
int main() {  
    Persona david("David", 15, 3, 1979);  
    david.imprimir();  
    modificar_copia(david);  
    david.imprimir();  
  
    return 0;  
}
```

Nombre: David  
Fecha de nacimiento: 15/03/1979

Nombre: David  
Fecha de nacimiento: 10/10/2010



# ¿Qué ha pasado?

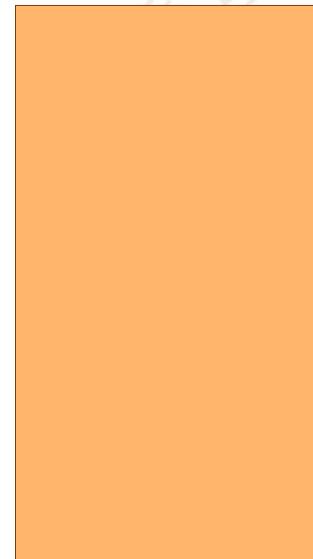
- Cuando se pasa una instancia a una función como parámetro **por valor**, se crea una copia de dicha instancia.
- **¿Cómo se realiza la copia?** Copiando el valor de cada uno de los atributos de la instancia “origen” a la instancia “destino”.

```
void modificar_copia(Persona p) { ... }

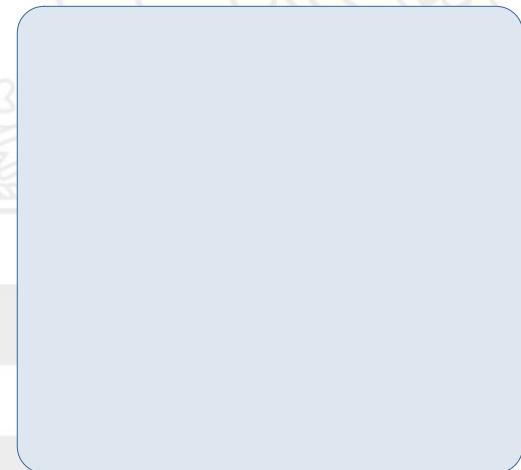
int main() {
    Persona david("David", 15, 3, 1979);
    david.imprimir();
    modificar_copia(david);
    david.imprimir();

    return 0;
}
```

Pila



Heap



# ¿Qué ha pasado?

- Copiar uno a uno los atributos funciona bien en la mayoría de los casos.
- Pero cuando los atributos son punteros a arrays u otras estructuras, solamente se hace una copia del **puntero**, de modo que tanto el objeto original como la copia, **apuntan a la misma estructura**.
- Aún peor: los **destructores** de sendas instancias pueden intentar liberar la estructura compartida **dos veces**.

¿Puede alterarse el modo en el que se realiza la copia en estos casos?

# Tipos de constructores

- **Constructor por defecto** (sin parámetros). ✓
- **Constructor paramétrico.** ✓
- **Constructor de copia.**
- **Constructor *move*.**
- **Constructor de conversión.**



# Constructor de copia

```
class Fecha {  
public:  
    ...  
    Fecha(const Fecha &f);  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```

- Es un método con el mismo nombre que la clase.
- Recibe un único parámetro: una referencia constante a un objeto de la misma clase.
- No devuelve nada.

# Constructor de copia

```
class Fecha {  
public:  
    ...  
    Fecha(const Fecha &f)  
        : dia(f.dia),  
          mes(f.mes),  
          anyo(f.anyo) { }  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```

- En el caso de Fecha, el constructor de copia inicializa los atributos del objeto con los atributos correspondientes del objeto f pasado como parámetro.
- Este es el comportamiento por defecto.

# Constructor de copia

```
class Persona {  
public:  
  
    Persona(const Persona &p)  
        : nombre(p.nombre) {  
        fecha_nacimiento =  
            new Fecha(  
                p.fecha_nacimiento->get_dia(),  
                p.fecha_nacimiento->get_mes(),  
                p.fecha_nacimiento->get_anyo()  
            );  
    }  
  
    ...  
}
```

- En el caso de Persona, el constructor de copia inicializa el atributo `fecha_nacimiento` creando un **nuevo** objeto `Fecha`, e inicializa los valores de este último con los de la fecha de `p`.

# Constructor de copia

```
class Persona {  
public:  
  
    Persona(const Persona &p)  
        : nombre(p.nombre) {  
        fecha_nacimiento =  
            new Fecha(*p.fecha_nacimiento);  
    }  
    ...  
}
```

- También podría haberse llamado explícitamente al constructor de copia de Fecha.



# Constructor de copia

```
class Persona {  
public:  
  
    Persona(const Persona &p)  
        : nombre(p.nombre),  
          fecha_nacimiento(  
              new Fecha(*p.fecha_nacimiento))  
    } { }  
  
}  
...
```

- También podría haberse llamado explícitamente al constructor de copia de Fecha.



# ¿Cuándo se llama al constructor de copia?

- Cuando se invoca explícitamente al crear un objeto.

```
Persona p1("David", 15, 3, 1979);  
Persona p2(p1);
```

- Cuando se declara una variable y se inicializa desde otro objeto.

```
Persona p1("David", 15, 3, 1979);  
Persona p2 = p1;
```

- Cuando se pasa un parámetro por valor.

```
bool es_navidad(Fecha f) { ... }  
...  
Fecha f1(15, 3, 1979);  
if(es_navidad(f1)) { ... }
```

- Cuando se devuelve un objeto como resultado.

```
Fecha nochevieja(int anyo) {  
    Fecha result(31, 12, anyo);  
    return result;  
}
```

# ¿Cuándo NO se llama?

- Cuando se asigna un objeto a una variable inicializada previamente.

```
Persona p1("David", 15, 3, 1979);
Persona p2("Gerardo", 1, 2, 1983);
p2 = p1;
```

No se llama al  
constructor de copia

```
Persona p1("David", 15, 3, 1979);
Persona p2 = p1;
```

Sí se llama al  
constructor de copia

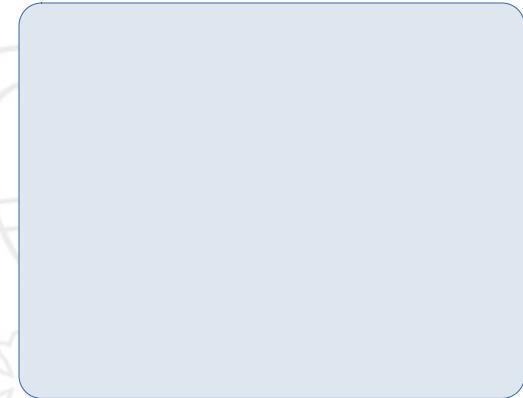
# Volviendo a nuestro ejemplo...

```
void modificar_copia(Persona p) {  
    p.set_nombre("Berta");  
    p.set_fecha_nacimiento(10, 10, 2010);  
}  
  
int main() {  
    Persona david("David", 15, 3, 1979);  
    david.imprimir();  
    modificar_copia(david);  
    david.imprimir();  
  
    return 0;  
}
```

Pila



Heap



Nombre: David  
Fecha de nacimiento: 15/03/1979

Nombre: David  
Fecha de nacimiento: 15/03/1979

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Sobrecarga de operadores

Manuel Montenegro Montes

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

# Ejemplo: números complejos

```
class Complejo {  
public:  
    Complejo(double real, double imag);  
  
    double get_real() const;  
    double get_imag() const;  
    void display() const;  
  
private:  
    double real, imag;  
};
```

Existe la clase `std :: complex`, definida en `<complex>`

# Aritmética con números complejos

```
Complejo suma(const Complejo &z1, const Complejo &z2) {
    return { z1.get_real() + z2.get_real(),
              z1.get_imag() + z2.get_imag() };
}

Complejo multiplica(const Complejo &z1, const Complejo &z2) {
    double z1_real = z1.get_real(), z1_imag = z1.get_imag();
    double z2_real = z2.get_real(), z2_imag = z2.get_imag();
    return { z1_real * z2_real - z1_imag * z2_imag,
              z1_real * z2_imag + z1_imag * z2_real };
}
```

# Ejemplo de uso

```
int main() {  
    Complejo z1(2.0, -3.0), z2(1.0, 0.0);  
    Complejo z3 = suma(z1, z2);  
    Complejo z4 = suma(multiplica(z1, z1), z2);  
  
    z3.display();  
    std::cout << std::endl;  
  
    z4.display();  
    std::cout << std::endl;  
  
    return 0;  
}
```

3-3i

-4-12i



# Uso de operadores

- Con los tipos numéricos básicos (`int`, `double`, etc.) podemos expresar operaciones aritméticas utilizando los operadores `+` y `*` en forma infija.
  - Ejemplo: `x + y * z`
- Con nuestra clase `Complejo` no tenemos la misma suerte:
  - `suma(z1, z2)`
  - `suma(multiplica(z1, z1), z2)`
- Sería más legible poder escribir:
  - `z1 + z2`
  - `z1 * z1 + z2`
- En C++ es posible definir implementaciones personalizadas de los operadores, es decir, **sobrecargarlos**.

# Sobrecargar operadores

# Aritmética con números complejos

```
Complejo suma(const Complejo &z1, const Complejo &z2) {
    return { z1.get_real() + z2.get_real(),
              z1.get_imag() + z2.get_imag() };
}

Complejo multiplica(const Complejo &z1, const Complejo &z2) {
    double z1_real = z1.get_real(), z1_imag = z1.get_imag();
    double z2_real = z2.get_real(), z2_imag = z2.get_imag();
    return { z1_real * z2_real - z1_imag * z2_imag,
              z1_real * z2_imag + z1_imag * z2_real };
}
```

# Aritmética con números complejos

```
Complejo operator+(const Complejo &z1, const Complejo &z2) {
    return { z1.get_real() + z2.get_real(),
              z1.get_imag() + z2.get_imag() };
}

Complejo operator*(const Complejo &z1, const Complejo &z2) {
    double z1_real = z1.get_real(), z1_imag = z1.get_imag();
    double z2_real = z2.get_real(), z2_imag = z2.get_imag();
    return { z1_real * z2_real - z1_imag * z2_imag,
              z1_real * z2_imag + z1_imag * z2_real };
}
```

- Puede sobrecargarse un operador creando una función con nombre **operator[?]**, donde **[?]** es un operador de C++.

# Ejemplo de uso

```
int main() {
    Complejo z1(2.0, -3.0), z2(1.0, 0.0);
    Complejo z3 = suma(z1, z2);
    Complejo z4 = suma(multiplica(z1, z1), z2);

    z3.display();
    std::cout << std::endl;

    z4.display();
    std::cout << std::endl;

    return 0;
}
```



# Ejemplo de uso

```
int main() {  
    Complejo z1(2.0, -3.0), z2(1.0, 0.0);  
    Complejo z3 = z1 + z2;—  
    Complejo z4 = z1 * z1 + z2;  
  
    z3.display();  
    std::cout << std::endl;  
  
    z4.display();  
    std::cout << std::endl;  
  
    return 0;  
}
```

Equivale a  
operator+(z1, z2)

Equivale a  
operator+(operator\*(z1, z1), z2)

# ¿Qué operadores pueden sobrecargarse?

+ - \* / % ^ & | << >>  
== <= >= != < > && || !  
= += -= \*= /=  
++ --  
[] () →  
new delete  
etc.

# Sobrecarga del operador << para E/S

# Generalizando el método `display()`

```
class Complejo {  
public:  
    ...  
    void display() const;  
private:  
    ...  
};
```



```
    void Complejo::display() const {  
        std::cout << real << ... << "i";  
    }
```

- El método `display()` envía una representación en cadena del objeto a la salida estándar (`std :: cout`).
  - ¿Y si quiero escribirla en un fichero (clase `ofstream`)?
  - ¿Y si quiero escribirla un `string` (clase `ostringstream`)?
- Todas heredan de la clase `ostream`.

# Generalizando el método `display()`

```
class Complejo {  
public:  
    ...  
    void display(ostream &out) const; → } {  
private:  
    ...  
};
```

void Complejo::display(ostream &out) const {  
 out << real << ... << "i";  
}

- El método `display()` envía una representación en cadena del objeto a la salida estándar (`std :: cout`).
- ¿Y si quiero escribirla en un fichero (clase `ofstream`)?
- ¿Y si quiero escribirla un `string` (clase `ostringstream`)?  
Todas heredan de la clase `ostream`.

# Actualizando el ejemplo

```
int main() {  
    Complejo z1(2.0, -3.0), z2(1.0, 0.0);  
    Complejo z3 = z1 + z2;  
    Complejo z4 = z1 * z1 + z2;  
  
    z3.display(std::cout);  
    std::cout << std::endl;  
  
    z4.display(std::cout);  
    std::cout << std::endl;  
  
    return 0;  
}
```



# El operador << para E/S

```
std :: cout << "Hola"
```

Instancia de  
ostream

Instancia de  
string

```
std :: cout << z1
```

Instancia de  
ostream

Instancia de  
Complejo

# Sobrecargando << para números complejos

```
void operator<<(std::ostream &out, Complejo &z) {  
    z.display(out);  
}
```

```
int main() {  
    ...  
  
    z3.display(std::cout);  
    std::cout << std::endl;  
  
    z4.display(std::cout);  
    std::cout << std::endl;  
  
    return 0;  
}
```

```
int main() {  
    ...  
  
    std::cout << z3;  
    std::cout << std::endl;  
  
    std::cout << z4;  
    std::cout << std::endl;  
  
    return 0;  
}
```

# Sobrecargando << para números complejos

```
void operator<<(std::ostream &out, Complejo &z) {  
    z.display(out);  
}
```

```
int main() {  
    ...  
  
    z3.display(std::cout);  
    std::cout << std::endl;  
  
    z4.display(std::cout);  
    std::cout << std::endl;  
  
    return 0;  
}
```

```
int main() {  
    ...  
  
    std::cout << z3 << std::endl << z4 << std::endl; X  
  
    return 0;  
}
```

# Sobrecargando << para números complejos

```
std::ostream & operator<<(std::ostream &out, Complejo &z) {  
    z.display(out);  
    return out;  
}
```

```
std::cout << z3 << std::endl << z4 << std::endl; ✓
```

# Sobrecarga dentro de una clase

# Sobrecarga fuera de una clase

- Las definiciones de sobrecarga vistas hasta ahora son funciones que no pertenecen a ninguna clase:

```
Complejo operator+(const Complejo &z1, const Complejo &z2) {  
    return { z1.get_real() + z2.get_real(),  
             z1.get_imag() + z2.get_imag() };  
}
```

# Sobrecarga dentro de una clase

- También habríamos podido definirlas como métodos de la clase Complejo.
- Si lo hacemos así, el primer operando es `this`.
- Ventaja: podemos acceder a los atributos privados.

```
class Complejo {  
public:  
    ...  
  
    Complejo operator+(const Complejo &z2) const {  
        return { real + z2.real, imag + z2.imag };  
    }  
  
private:  
    double real, imag;  
};
```

$z1 + z2$

equivale a

`z1.operator+(z2)`

# ¿Podemos hacer lo mismo con...?

```
Complejo operator*(const Complejo &z1, const Complejo &z2) {  
    double z1_real = z1.get_real(), z1_imag = z1.get_imag();  
    double z2_real = z2.get_real(), z2_imag = z2.get_imag();  
    return { z1_real * z2_real - z1_imag * z2_imag,  
             z1_real * z2_imag + z1_imag * z2_real };  
}
```

```
std::ostream & operator<<(std::ostream &out, Complejo &z) {  
    z.display(out);  
    return out;  
}
```



¡No podemos añadir  
métodos a la clase  
`ostream`!

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Operador de asignación

Manuel Montenegro Montes

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

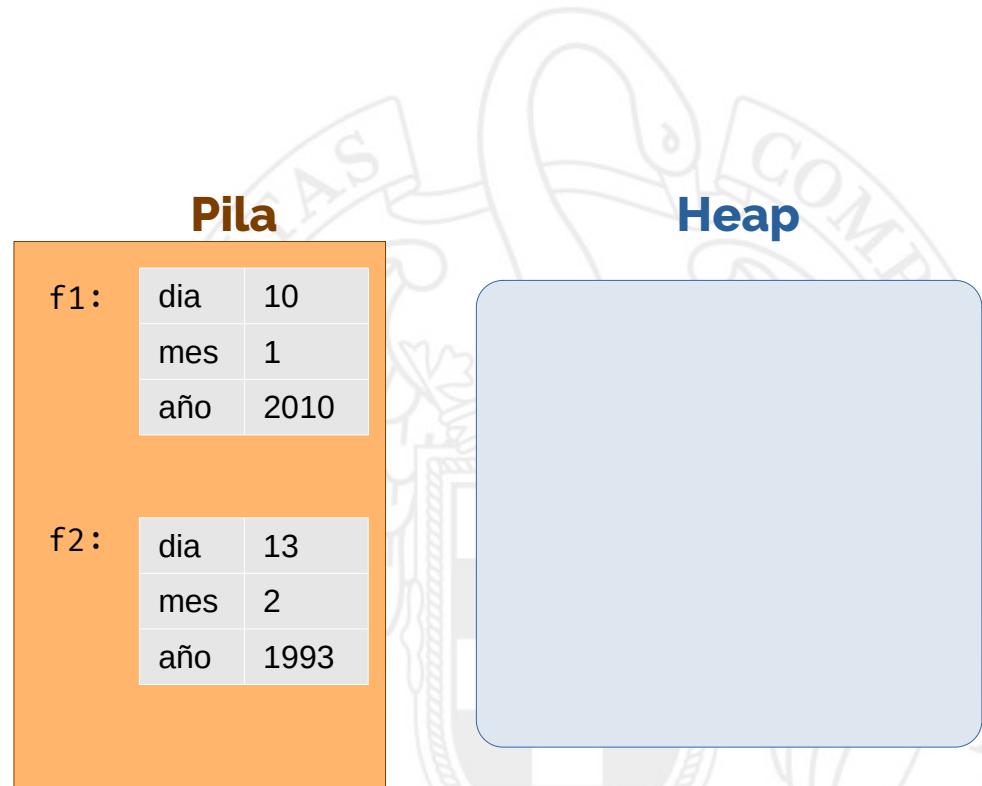
# Recordatorio: clase Fecha

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo);  
    Fecha(int anyo);  
    Fecha();  
  
    int get_dia() const;  
    void set_dia(int dia);  
    int get_mes() const;  
    void set_mes(int mes);  
    int get_anyo() const;  
    void set_anyo(int anyo);  
    void imprimir();  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```



# Asignar un objeto Fecha a otro

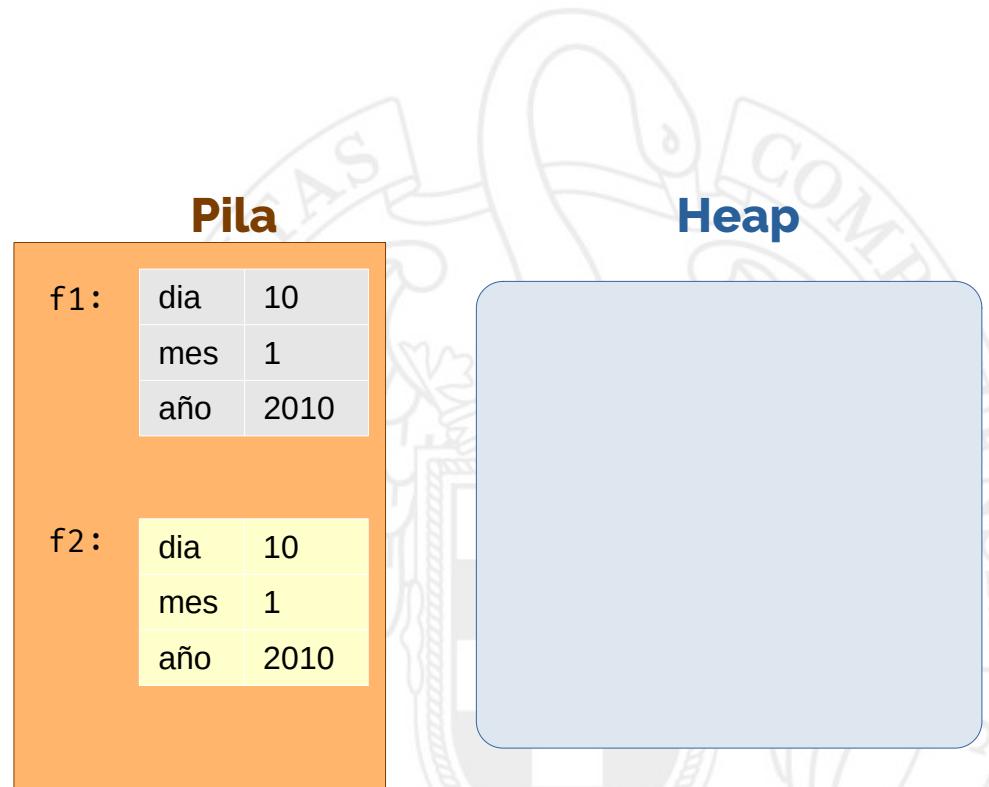
```
Fecha f1(10, 1, 2010);  
Fecha f2(13, 2, 1993);  
  
f2 = f1;
```



# Asignar un objeto Fecha a otro

```
Fecha f1(10, 1, 2010);  
Fecha f2(13, 2, 1993);
```

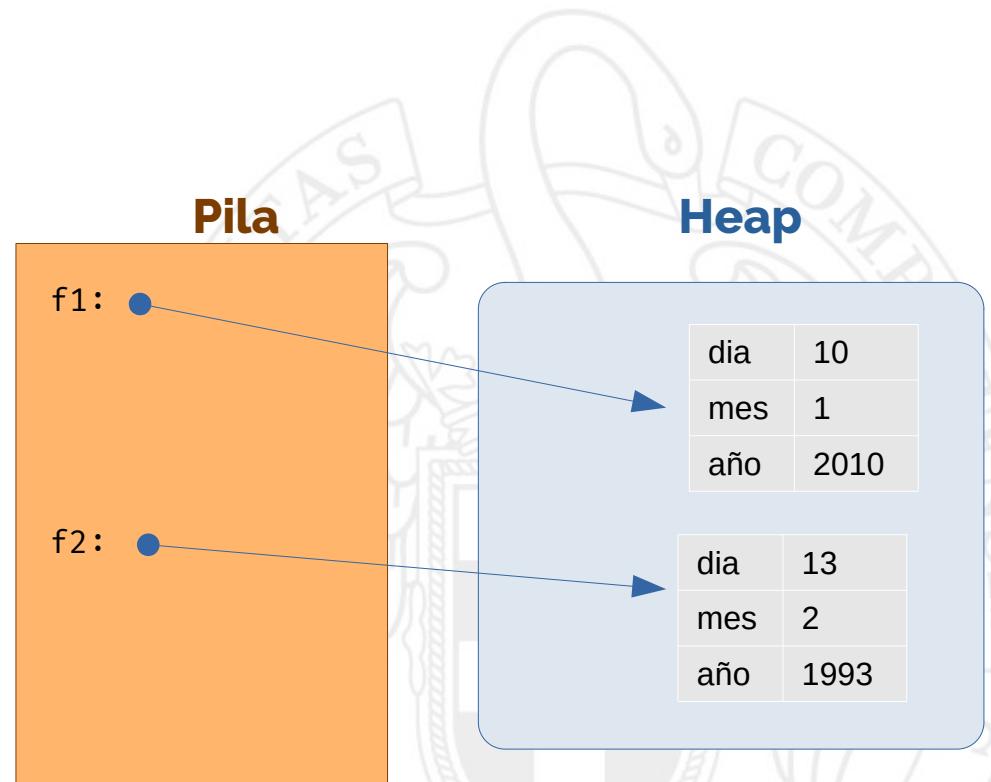
```
f2 = f1;
```



# Asignar un objeto Fecha a otro

```
Fecha *f1 = new Fecha(10, 1, 2010);  
Fecha *f2 = new Fecha(13, 2, 1993);
```

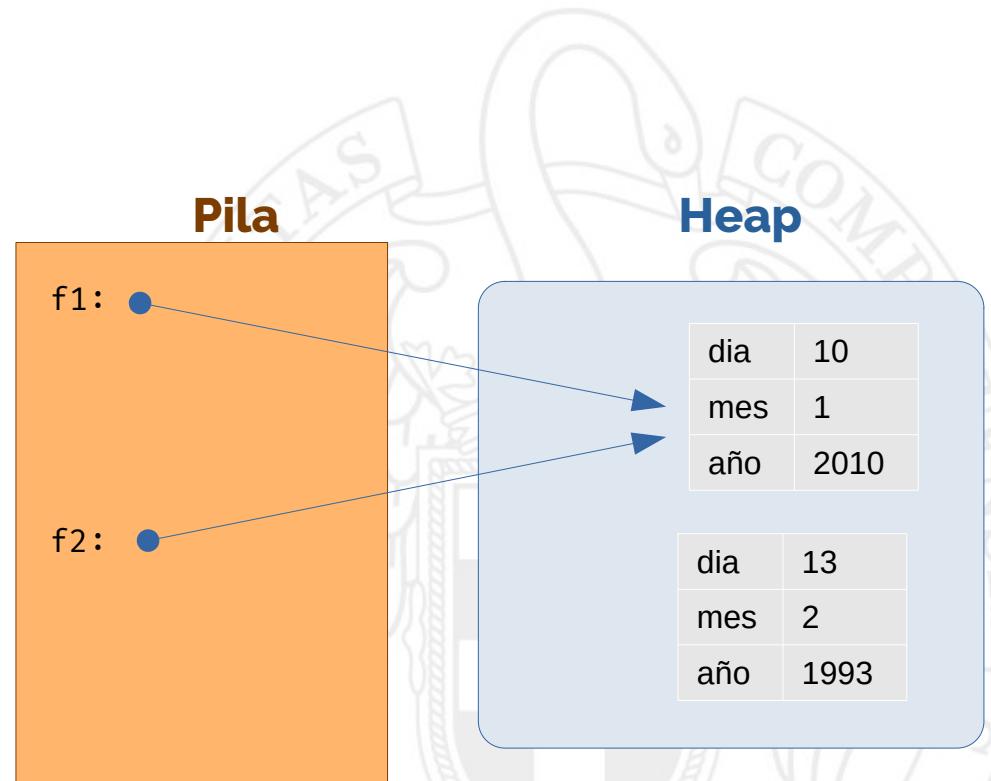
```
f2 = f1;
```



# Asignar un objeto Fecha a otro

```
Fecha *f1 = new Fecha(10, 1, 2010);  
Fecha *f2 = new Fecha(13, 2, 1993);
```

```
f2 = f1;
```



# Recordatorio: clase Persona

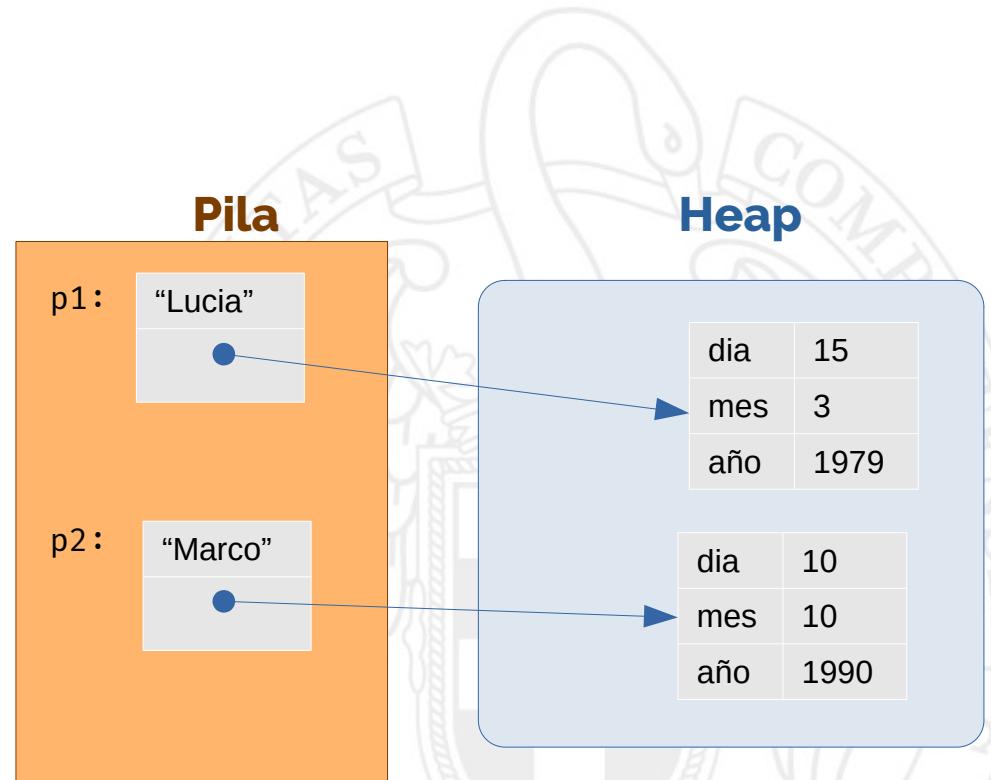
```
class Persona {  
public:  
    Persona(std::string nombre,  
            int dia,  
            int mes,  
            int anyo);  
  
    ~Persona() {  
        delete fecha_nacimiento;  
    }  
  
    ...  
  
private:  
    std::string nombre;  
    Fecha *fecha_nacimiento;  
};
```



# Asignar un objeto Persona a otro

```
Persona p1("Lucía", 15, 3, 1979);  
Persona p2("Marco", 10, 10, 1990);
```

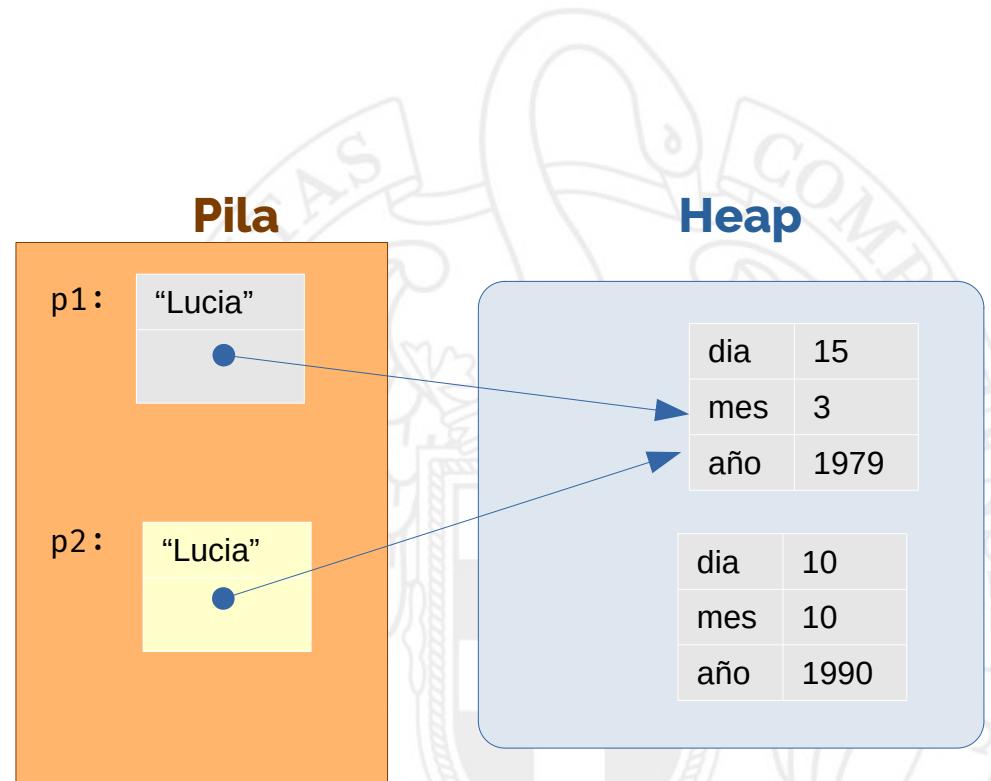
```
p2 = p1;
```



# Asignar un objeto Persona a otro

```
Persona p1("Lucía", 15, 3, 1979);  
Persona p2("Marco", 10, 10, 1990);
```

```
p2 = p1;
```



# Sobrecargando el operador de asignación

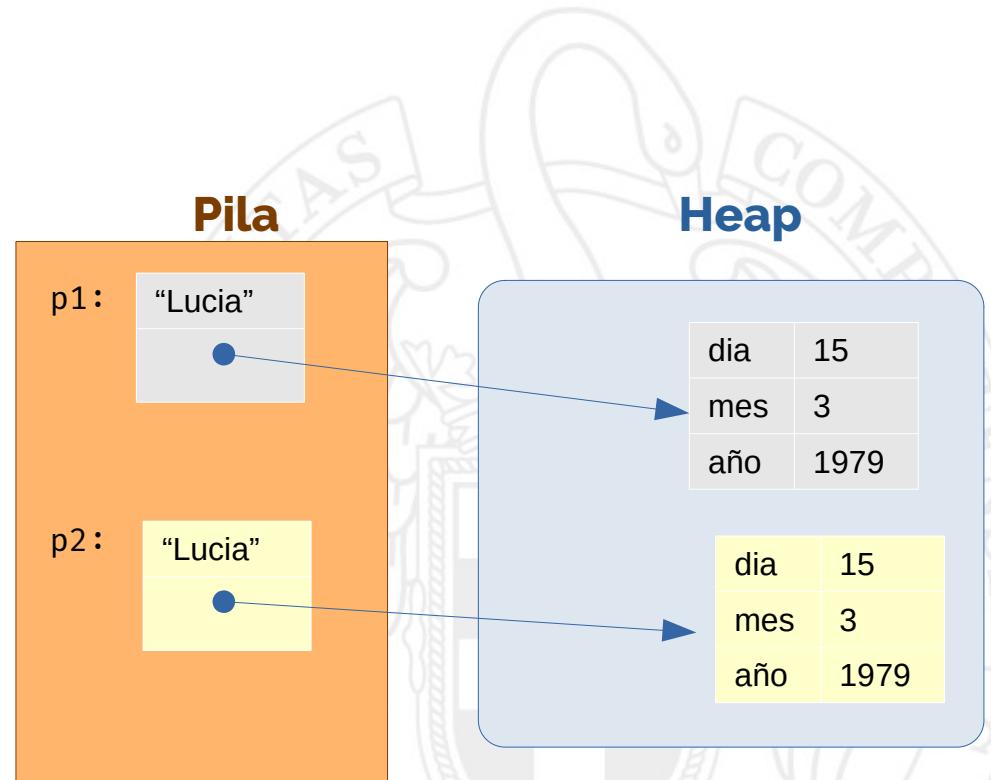
```
class Persona {  
public:  
    ...  
  
    void operator=(const Persona &other) {  
        nombre = other.nombre;  
        fecha_nacimiento→set_dia(other.fecha_nacimiento→get_dia());  
        fecha_nacimiento→set_mes(other.fecha_nacimiento→get_mes());  
        fecha_nacimiento→set_anyo(other.fecha_nacimiento→get_anyo());  
    }  
  
    ...  
  
private:  
    std::string nombre;  
    Fecha *fecha_nacimiento;  
};
```

p2 = p1  
equivale a  
p2.operator=(p1)

# Asignar un objeto Persona a otro

```
Persona p1("Lucía", 15, 3, 1979);  
Persona p2("Marco", 10, 10, 1990);
```

```
p2 = p1;
```



# Otra posibilidad

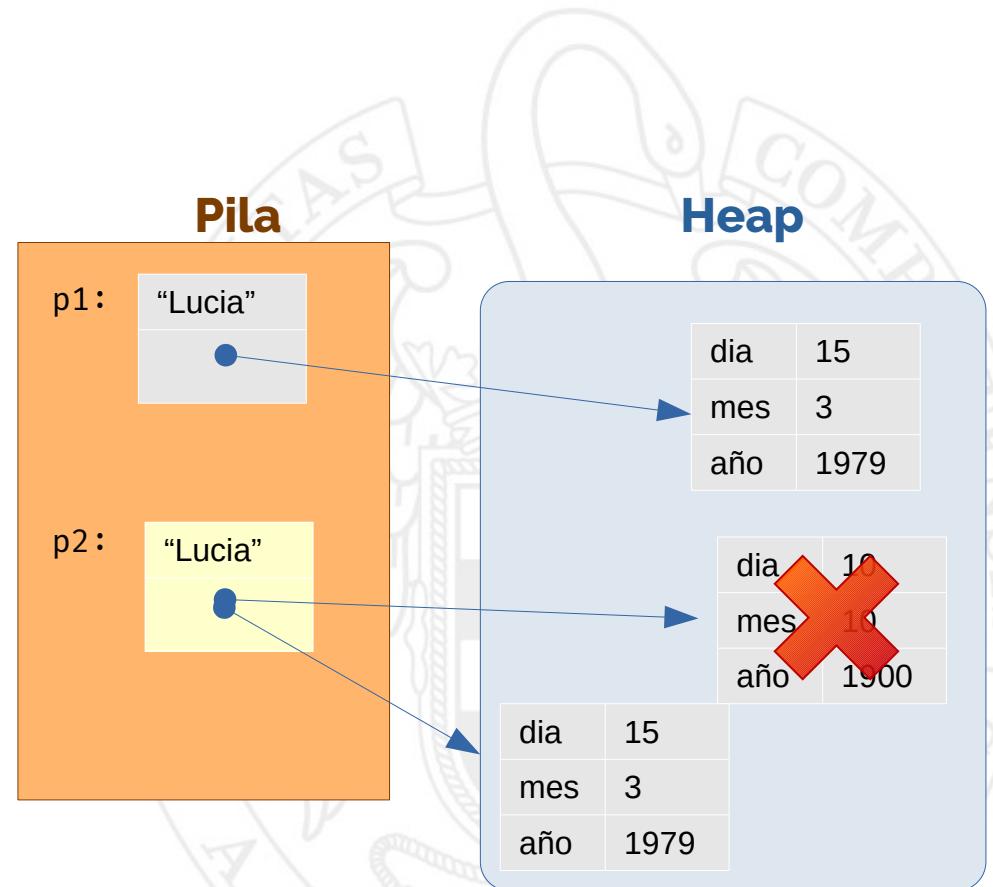
```
class Persona {  
public:  
  
...  
  
void operator=(const Persona &other) {  
    nombre = other.nombre;  
    delete fecha_nacimiento;  
    fecha_nacimiento = new Fecha(*other.fecha_nacimiento);  
}  
  
...  
  
private:  
    std::string nombre;  
    Fecha *fecha_nacimiento;  
};
```



# Asignar un objeto Persona a otro

```
Persona p1("Lucía", 15, 3, 1979);  
Persona p2("Marco", 10, 10, 1990);
```

```
p2 = p1;
```

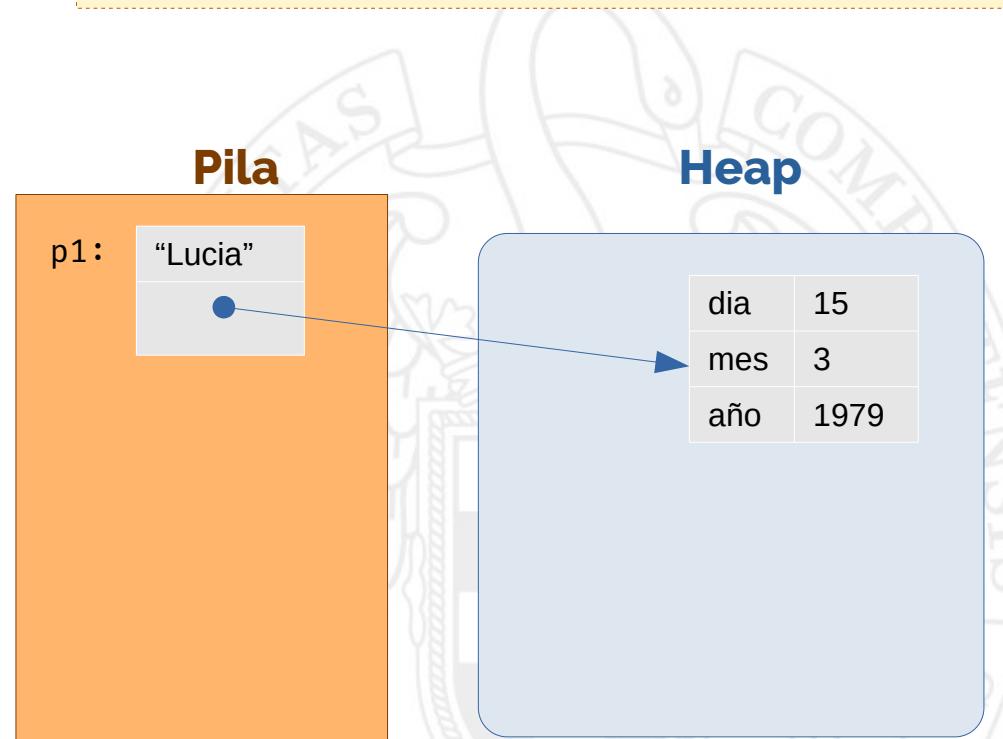


# **El problema de la autoasignación**

# Asignar un objeto persona a sí mismo

```
Persona p1("Lucía", 15, 3, 1979);  
p1 = p1;
```

```
void operator=(const Persona &other) {  
    nombre = other.nombre;  
    delete fecha_nacimiento;  
    fecha_nacimiento = new Fecha(*other.fecha_nacimiento);  
}
```



# Evitando la autoasignación

```
class Persona {  
public:  
  
...  
  
void operator=(const Persona &other) {  
    if (this != &other) {  
        nombre = other.nombre;  
        delete fecha_nacimiento;  
        fecha_nacimiento = new Fecha(*other.fecha_nacimiento);  
    }  
}  
...  
  
private:  
    std::string nombre;  
    Fecha *fecha_nacimiento;  
};
```



# Encadenar asignaciones

# Encadenar asignaciones

```
int x, y, z;  
x = y = z = 0;
```

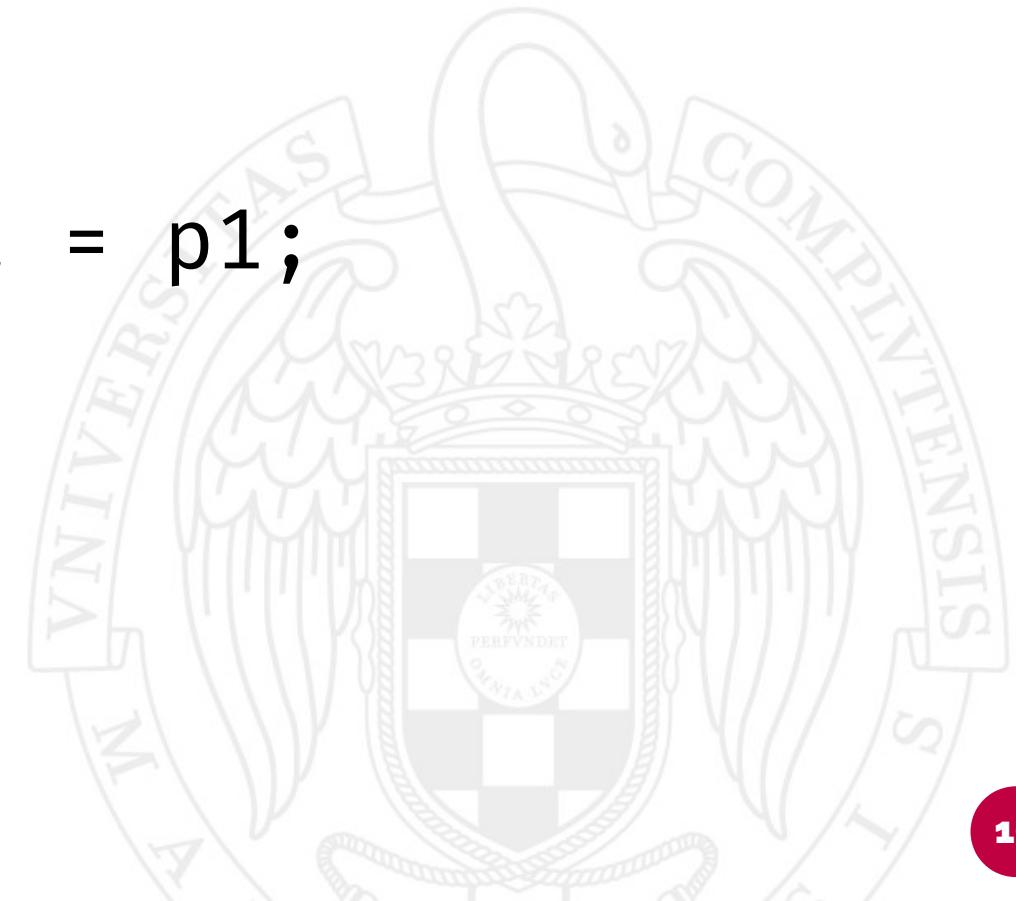
x = y = z = 0;

# ¿Podemos hacer lo mismo con objetos Persona?

```
Persona p1("Lucía", 15, 3, 1979);  
Persona p2("Marco", 10, 10, 1990);  
Persona p3("Laura", 1, 3, 1980);
```

p3 = p2 = p1; 

p3 = p2 = p1;



# Devolviendo referencia a this

```
class Persona {  
public:  
  
...  
  
    Persona & operator=(const Persona &other) {  
        if (this != &other) {  
            nombre = other.nombre;  
            delete fecha_nacimiento;  
            fecha_nacimiento = new Fecha(*other.fecha_nacimiento);  
        }  
  
        return *this;  
    }  
  
...  
  
private:  
    std::string nombre;  
    Fecha *fecha_nacimiento;  
};
```

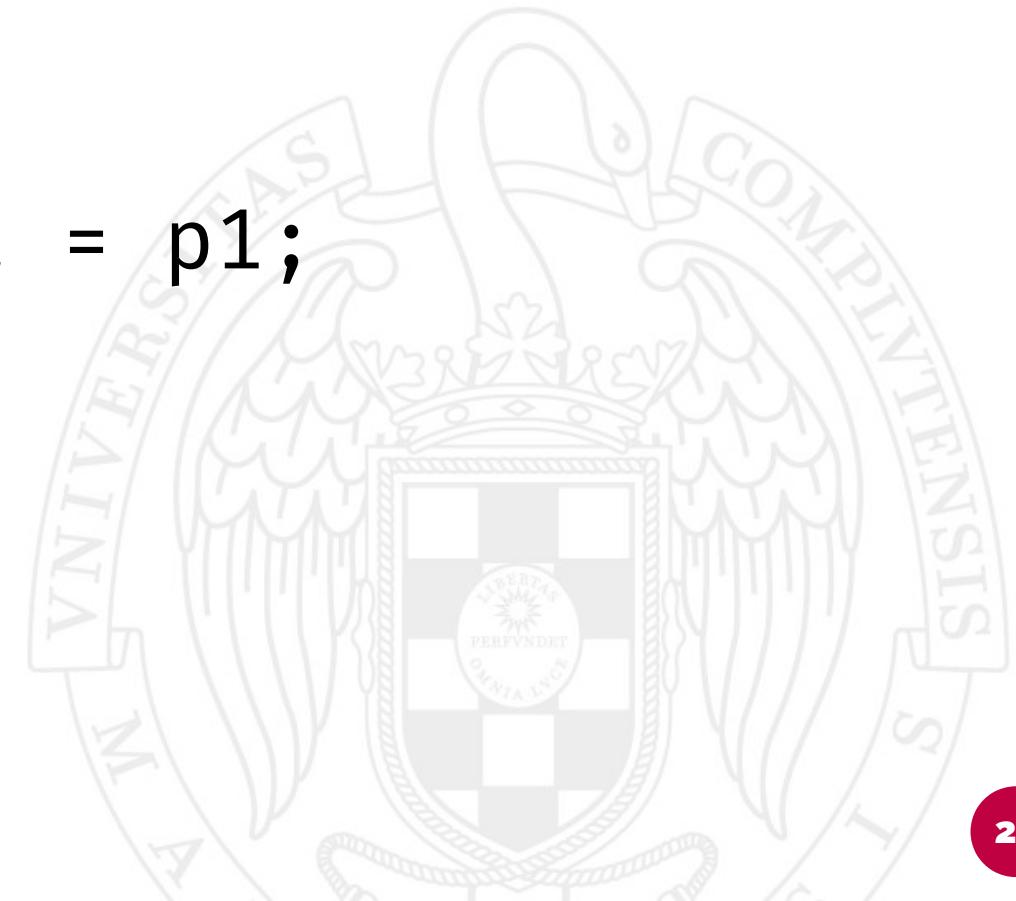


# ¿Podemos hacer lo mismo con objetos Persona?

```
Persona p1("Lucía", 15, 3, 1979);  
Persona p2("Marco", 10, 10, 1990);  
Persona p3("Laura", 1, 3, 1980);
```

p3 = p2 = p1; 

p3 = p2 = p1;



# Constructor de copia    vs.    Operador asignación

- Para crear un objeto nuevo con la misma información que otro existente.

```
Persona p1(...);  
Persona p2 = p1;
```

- No devuelve nada.
- No puede producirse autoasignación:

```
Persona p2 = p2;
```

- Para copiar la información de un objeto existente a otro existente.

```
Persona p1(...);  
Persona p2(...);  
p2 = p1;
```

- Devuelve `*this`.
- Hay que tener en cuenta la autoasignación:

```
p2 = p2;
```

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Plantillas en funciones

Manuel Montenegro Montes

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

# Ejemplo

- Implementamos una función que calcula el mínimo de dos enteros:

```
int min(int a, int b) {  
    if (a ≤ b) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

- ¿Y si quiero calcular el mínimo de dos `float`? ¿y el mínimo de dos `double`?
- ¿Y si quiero calcular el mínimo de dos `string` utilizando el orden lexicográfico? Por ejemplo: `min("AA", "AB") = "AA"`.

# Ejemplo

```
int min(int a, int b) {  
    if (a <= b) {  
        return a;  
    } else {  
        return b;  
    }  
}  
  
float min(float a, float b) {  
    if (a <= b) {  
        return a;  
    } else {  
        return b;  
    }  
}  
  
double min(double a, double b) {  
    if (a <= b) {  
        return a;  
    } else {  
        return b;  
    }  
}  
  
const std::string & min(const std::string &a, const std::string &b) {  
    if (a <= b) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

- ¡Cuanta duplicidad!
- Todas tienen la misma implementación. ¡Solo difieren en los tipos!

# Programación genérica

- Sería deseable tener una única versión genérica, que pudiese funcionar con varios tipos.

```
??? min( ??? a, ??? b) {  
    if (a <= b) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

- **Solución:** plantillas (*templates*) en C++.



# Plantillas en C++

- Son definiciones con «huecos» (**parámetros de plantilla**).
- Se especifican mediante la palabra `template`, seguida de los parámetros de plantilla, y seguida de la definición de función paramétrica.

```
template <typename T>
T min(T a, T b) {
    if (a ≤ b) {
        return a;
    } else {
        return b;
    }
}
```

# Llamada a funciones plantilla

- Basta con indicar el tipo con el que queremos «rellenar» el marcador.

```
min<int>(6, 2)
min<double>(3.3, 5.5)
min<std::string>("Pepito", "Paula")
```

- Cada vez que se hace una llamada a la función genérica, se hace una versión específica para el tipo indicado en el marcador. A esto se le llama **instanciación de plantillas**.

```
template <typename T>
T min(T a, T b) {
    if (a ≤ b) {
        return a;
    } else {
        return b;
    }
}
```

T = int  
→

```
int min<int>(int a, int b) {
    if (a ≤ b) {
        return a;
    } else {
        return b;
    }
}
```

# Instanciación de plantillas

```
template <typename T>
T min(T a, T b) {
    if (a ≤ b) {
        return a;
    } else {
        return b;
    }
}
```

T = int

T = std::string

T = double

```
std::string min(std::string a, std::string b) {
    if (a ≤ b) {
        return a;
    } else {
        return b;
    }
}
```

```
int min<int>(int a, int b) {
    if (a ≤ b) {
        return a;
    } else {
        return b;
    }
}
```

```
double min<double>(double a, double b) {
    if (a ≤ b) {
        return a;
    } else {
        return b;
    }
}
```

# Instanciación de plantillas

- En esta última instancia (con un `string`) podemos indicar un tipo más preciso:

```
std::cout << min<const std::string &>("Pepito", "Ramiro") << std::endl;
```

- O bien modificar nuestra función genérica:

```
template <typename T>
const T & min(const T &a, const T &b) {
    if (a <= b) {
        return a;
    } else {
        return b;
    }
}
```

# Deducción de argumentos de plantilla

- Cada vez que hemos llamado a una función genérica, hemos indicado el tipo con el que debe instanciarse:

```
std::cout << min<std::string>("Pepito", "Ramiro") << std::endl;
```

- C++ permite omitirlo en la mayoría de los casos.
  - En ese caso intenta deducir el argumento de la plantilla.

```
std::cout << min("Pepito", "Ramiro") << std::endl;
```



# ¡Cuidado con las instanciaciones!

- ¿Qué pasa si instancio la plantilla con dos complejos?

```
Complejo z1(1.0, 3.0), z2(4.0, -5.0);  
std::cout << min(z1, z2) << std::endl;
```

- C++ realiza esta instancia:

```
template <typename T>  
const T & min(const T &a, const T &b) {  
    if (a <= b) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

T = Complejo

```
const Complejo & min(const Complejo &a,  
                      const Complejo &b)  
{  
    if (a <= b) {  
        return a;  
    } else {  
        return b;  
    }  
}
```



# ¡Cuidado con las instanciaciones!

- Los errores provocados por instanciaciones incorrectas suelen ser crípticos, largos, y difíciles de interpretar:

Test1.cpp: En la instanciaación de ‘const T& min(const T&, const T&) [con T = Complejo]’:

Test1.cpp:76:28: se requiere desde aquí

Test1.cpp:40:11: error: no match for ‘operator<=’ (operand types are ‘const Complejo’ and ‘const Complejo’)

```
40 |     if (a <= b) {  
|         ~~~~^~~~~~
```

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Plantillas en clases

Manuel Montenegro Montes

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

# Repaso: números complejos

```
class Complejo {  
public:  
    Complejo(double real, double imag);  
  
    double get_real() const;  
    double get_imag() const;  
    void display() const;  
  
    Complejo operator+(const Complejo &z) const;  
    Complejo operator*(const Complejo &z) const;  
  
private:  
    double real, imag;  
};
```

- ¿Y si quisiera también una clase Complejo en la que las partes reales o imaginarias sean float, en lugar de double?
- Para evitar duplicidad de código puedo utilizar plantillas.

# Generalización de una clase

```
template<typename T>
class Complejo {
public:
    Complejo(T real, T imag);

    T get_real() const;
    T get_imag() const;

    void display(std::ostream &out) const;

    Complejo operator+(const Complejo &z) const;
    Complejo operator*(const Complejo &z) const;

private:
    T real, imag;
};
```



# Generalización de los métodos

```
template<typename T>
class Complejo {
public:
    ...
    T get_real() const {
        return real;
    };

    T get_imag() const {
        return imag;
    };

    ...
private:
    T real, imag;
};
```

- Si el método se implementa dentro de la clase, no es necesario hacer nada nuevo.

# Generalización de los métodos

```
template<typename T>
class Complejo {
public:
    ...
    Complejo operator+(const Complejo &z1) const;
    Complejo operator*(const Complejo &z1) const;
    ...
private:
    T real, imag;
};

template<typename T>
Complejo<T> Complejo<T>::operator+(const Complejo<T> &z) const {
    return { real + z.real, imag + z.imag };
}
```

- Si el método se implementa fuera de la clase, es necesario indicar que el método también es una plantilla.

# Uso de una clase genérica

- A la hora de crear una instancia de una clase genérica, hay que indicar el tipo con el que se instancia:

```
Complejo<double> z1(2.0, -3.0), z2(1.0, 0.0);
```

```
Complejo<float> z4(2.0, -3.0);
```



*En las clases, es **obligatorio** indicar el tipo con el que instanciar la plantilla*

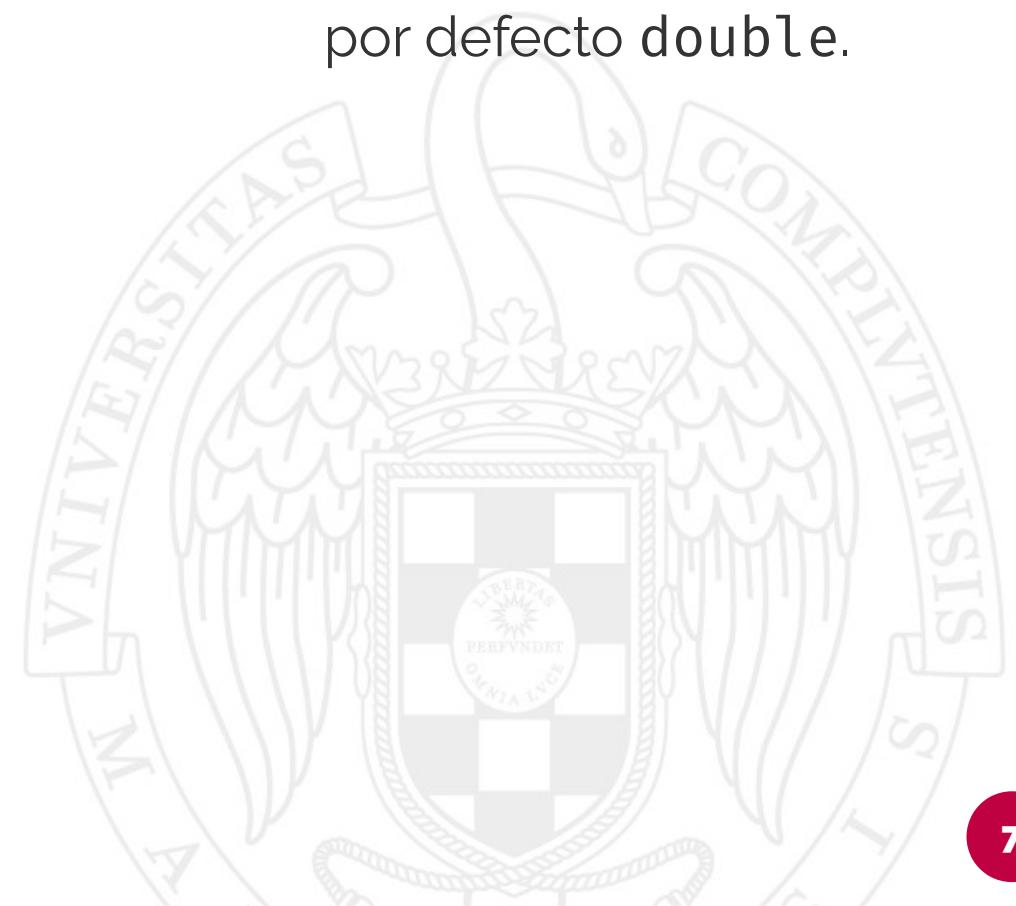
```
Complejo z4(2.0, -3.0); 
```

- ... aunque es posible indicar un tipo por defecto para la instancia.

# Plantillas: argumentos por defecto

```
template<typename T = double>
class Complejo {
    ...
};
```

- Si no se indica el tipo en la instancia, se utilizará por defecto double.



# Plantillas: argumentos por defecto

- Aún si queremos utilizar argumentos por defecto, es necesario indicar los delimitadores < y >, aunque no tengan nada en su interior.

```
Complejo<> z1(2.0, -3.0), z2(1.0, 0.0); ← Correcto (= Complejo<double>)  
Complejo z1(2.0, -3.0), z2(1.0, 0.0); ← Incorrecto
```

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# STL: Contenedores lineales

Manuel Montenegro Montes

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

# ¿Qué es la STL?

**STL = *Standard Template Library***

- Es una librería estándar de C++ que proporciona una serie de utilidades al programador/a.
  - Tipos abstractos de datos para almacenar colecciones de elementos: listas, pilas, colas, conjuntos, diccionarios, etc.
  - Iteradores.
  - Algoritmos sobre estos tipos abstractos de datos.

# Tipos de datos lineales en la STL

Clase	Fich. cabecera	Estructura
std::vector	<vector>	TAD Lista (arrays)
std::list	<list>	TAD Lista (listas doblemente enlazadas)
std::forward_list	<forward_list>	TAD Lista (listas enlazadas simples)
std::deque	<deque>	TAD doble cola
std::stack	<stack>	TAD pila
std::queue	<queue>	TAD cola

# Operaciones

- Tienen exactamente el mismo nombre que las que hemos visto a lo largo del curso:
  - `push_back()`
  - `push_front()`
  - `operator[]`
  - `begin()`
  - etc.



# Algunas excepciones

- `vector` no implementa `push_front()` o `pop_front()`.
- `list` no implementa `at()` ni el operador `[ ]`.
- No tienen ninguna función `display()`, ni sobrecargan el operador `<<`.



# Ejemplo

```
int main() {
    vector<int> v;
    for (int i = 0; i < 10; i++) {
        v.push_back(i * 3);
    }

    cout << v.size() << endl;
    int suma = 0;
    for (int x : v) {
        suma += x;
    }

    cout << "Suma total: " << suma << endl;
    return 0;
}
```



ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# STL: Iteradores

Manuel Montenegro Montes

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

# Tipos de iteradores

- Iteradores de entrada.
- Iteradores de salida.
- Iteradores hacia delante.
- Iteradores bidireccionales.
- Iteradores de acceso aleatorio.



# Iteradores de entrada

Entrada

`... = *it`

*Acceso*

`it++`

*Avance*

`it1 == it2`

*Comparación*



# Iteradores de salida

Entrada

`... = *it`

*Acceso*

`it1 == it2`

*Comparación*

`it++`

*Avance*

`*it = ...`

*Escritura*

Salida

# Iteradores hacia delante

Entrada

`... = *it`

*Acceso*

`it1 == it2`

*Comparación*

`it++`

*Avance*

`*it = ...`

*Escritura*

Salida

Hacia delante

# Iteradores bidireccionales

`... = *it`

*Acceso*

`it++`

*Avance*

`*it = ...`

*Escritura*

`it1 == it2`

*Comparación*

`it--`

*Retroceso*

Hacia delante

Bidireccionales

# Iteradores de acceso aleatorio

`... = *it`

*Acceso*

`it++`

*Avance*

`*it = ...`

*Escritura*

`it1 == it2`

*Comparación*

Hacia delante

`it--`

*Retroceso*

Bidireccionales

`it = it ± n`

*Avance/retroceso  
por saltos*

Acceso aleatorio

# Tipos de iteradores

- Cada implementación de TAD soporta un tipo de iterador determinado.

Expresión	Tipo de iterador
<code>vector :: begin()</code>	Acceso aleatorio
<code>list :: begin()</code>	Bidireccional
<code>deque :: begin()</code>	Acceso aleatorio
<code>forward_list :: begin()</code>	Hacia delante
<code>ostream_iterator</code>	Salida
<code>istream_iterator</code>	Entrada
<i>Punteros</i>	Acceso aleatorio

# Iterador de salida: ostream\_iterator

- Es un iterador asociado a un flujo de salida (fichero, salida estándar, etc.)
- Cada vez que se modifica el valor apuntado por el iterador, se realiza una operación de salida.
- Cada vez que se incrementa el iterador, no se hace nada.
- Es útil para la función `copy()`

# Ejemplo

```
int main() {  
    std::ostream_iterator<int> it(std::cout, " ");  
  
    *it = 10;  
    it++; // Opcional. No hace nada.  
    *it = 20;  
    it++; // Opcional. No hace nada.  
  
    std::cout << std::endl;  
  
    return 0;  
}
```

Flujo de salida

Separador

std::cout

it

# Ejemplo

```
int main() {  
    std::ostream_iterator<int> it(std::cout, " ");  
  
    *it = 10;  
    it++; // Opcional. No hace nada.  
    *it = 20;  
    it++; // Opcional. No hace nada.  
  
    std::cout << std::endl;  
  
    return 0;  
}
```

Flujo de salida

Separador

std::cout

10\_

it

# Ejemplo

```
int main() {  
    std::ostream_iterator<int> it(std::cout, " ");  
  
    *it = 10;  
    it++; // Opcional. No hace nada.  
    *it = 20;  
    it++; // Opcional. No hace nada.  
  
    std::cout << std::endl;  
  
    return 0;  
}
```

Flujo de salida

Separador

std::cout

10\_20\_

it

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# STL: Algoritmos (1)

Manuel Montenegro Montes

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

# La función copy()

# La función `copy()`

- Definida en `<algorithm>`

*copy(source\_begin, source\_end, destination\_begin)*

donde:

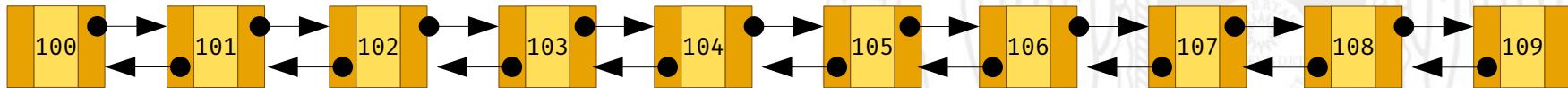
- `source_begin`, `source_end` son iteradores de entrada.
  - `destination_begin` es iterador de salida.
- Copia el intervalo de elementos delimitado por `source_begin` y `source_end` (excluyendo este último), a la posición apuntada por el iterador `destination_begin`.

# Ejemplo

```
int main() {
    vector<int> origen;
    list<int> destino;

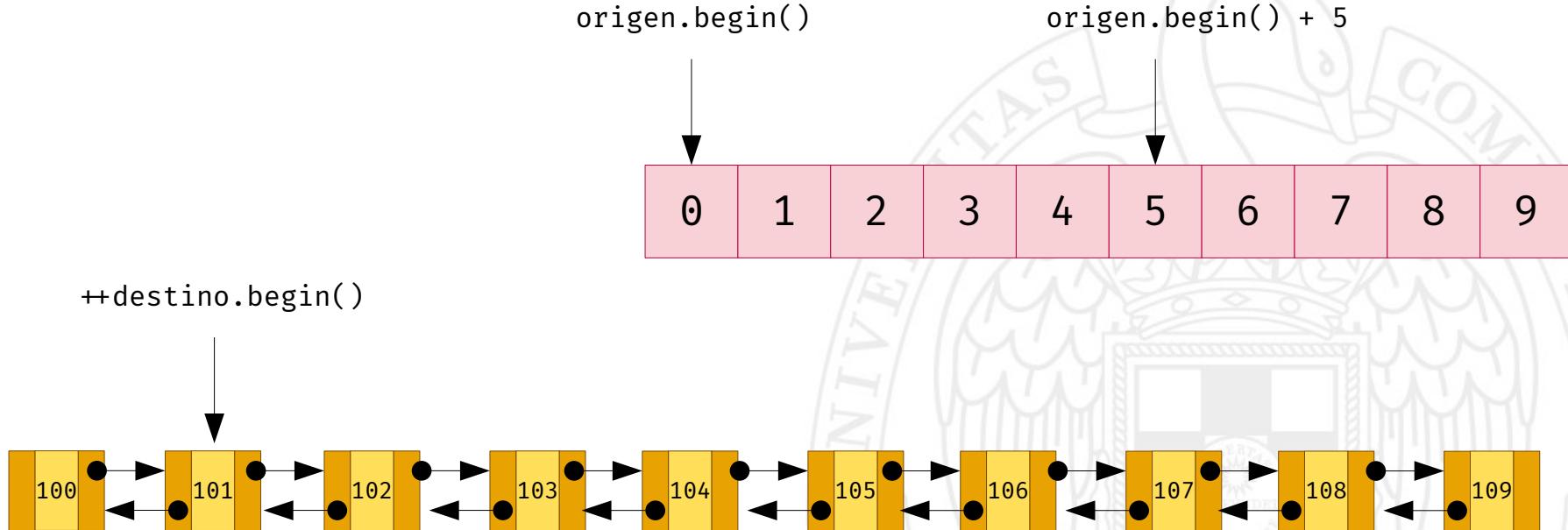
    for (int i = 0; i < 10; i++) {
        origen.push_back(i);
        destino.push_back(100 + i);
    }
    ...
}
```

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---



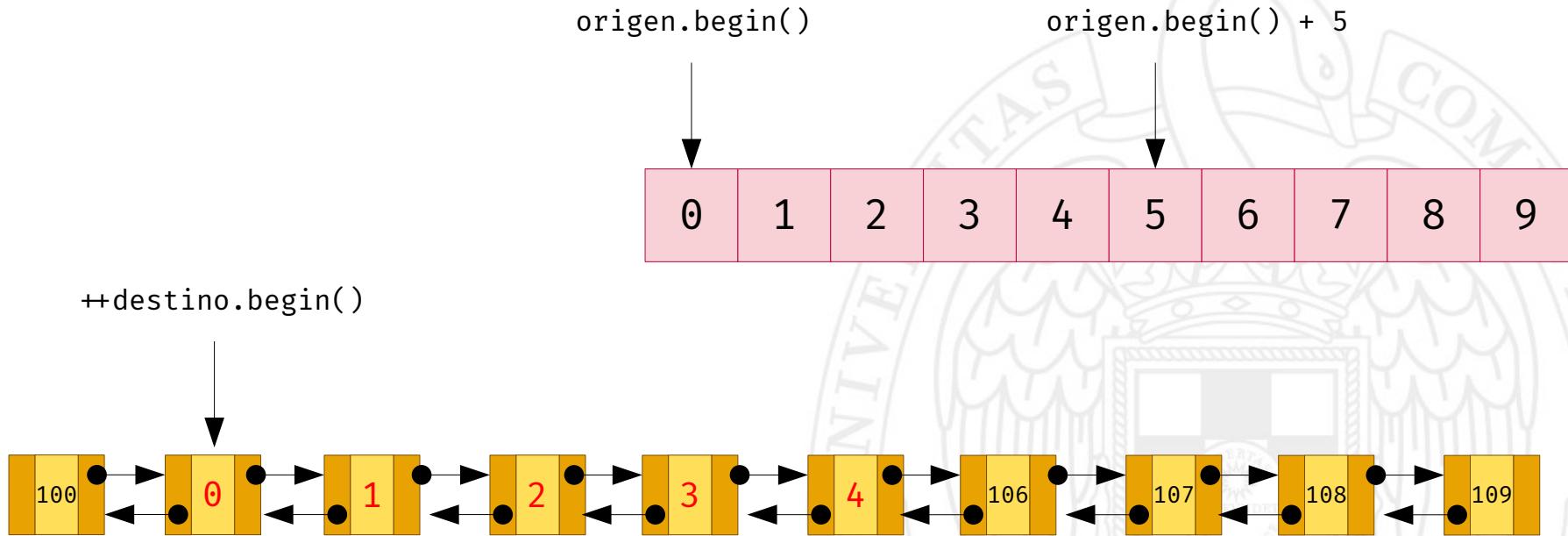
# Ejemplo

```
copy(origen.begin(), origen.begin() + 5, ++destino.begin());
```



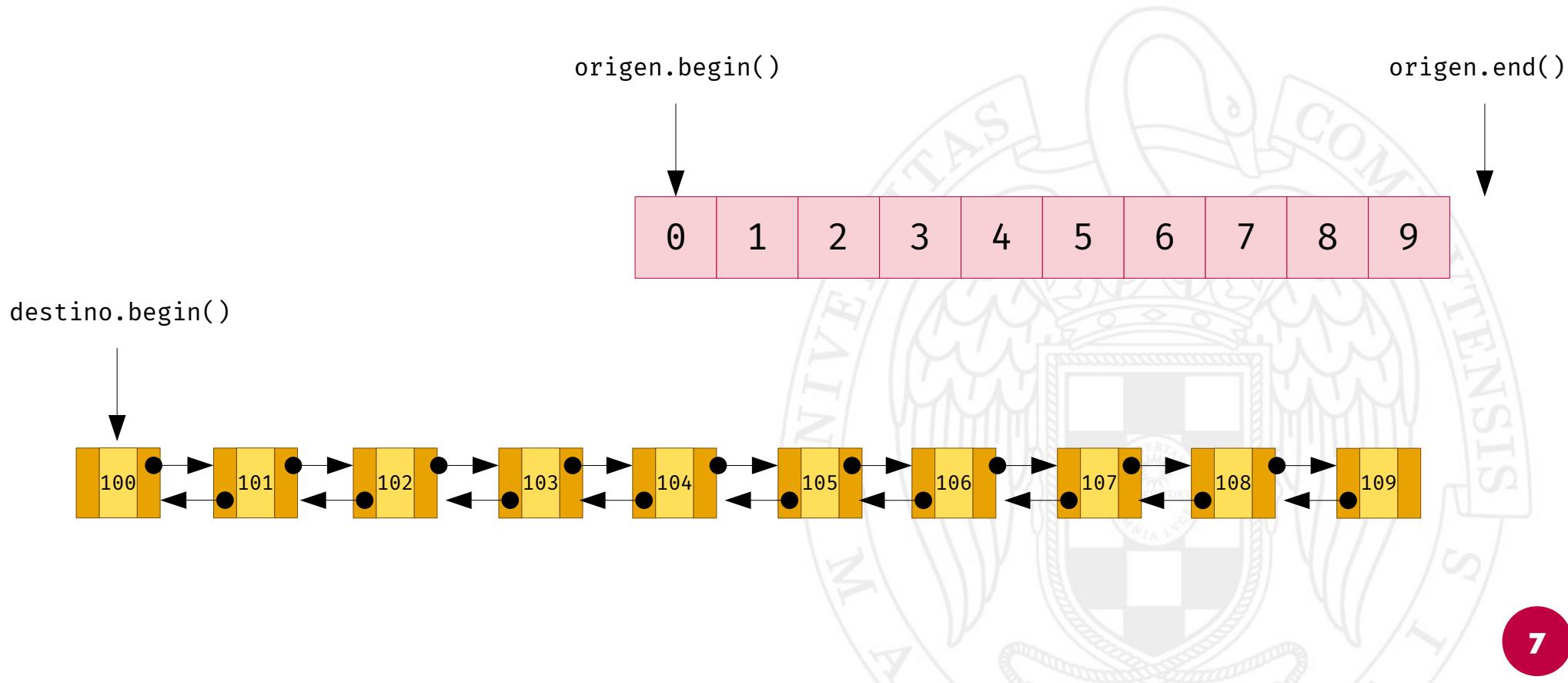
# Ejemplo

```
copy(origen.begin(), origen.begin() + 5, ++destino.begin());
```



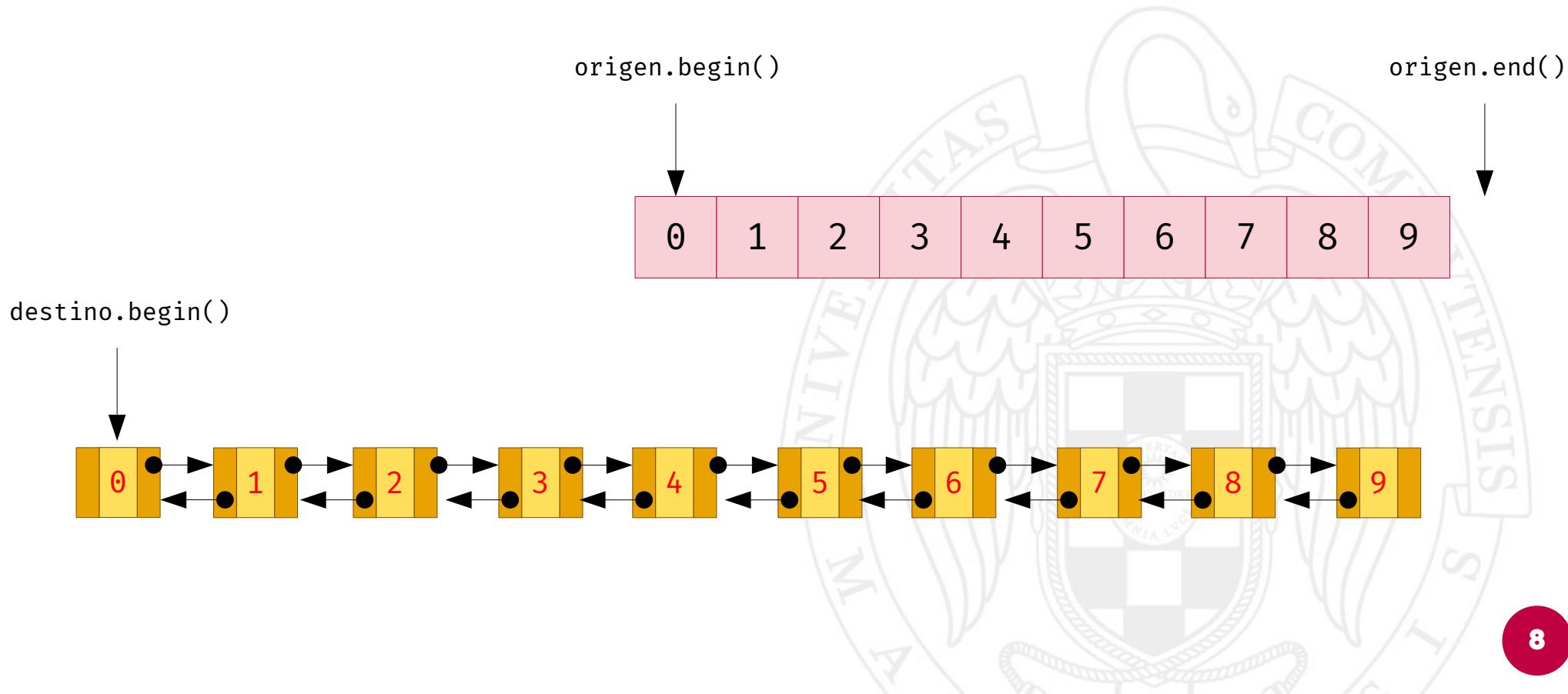
# Ejemplo

```
copy(origen.begin(), origen.end(), destino.begin());
```



# Ejemplo

```
copy(origen.begin(), origen.end(), destino.begin());
```

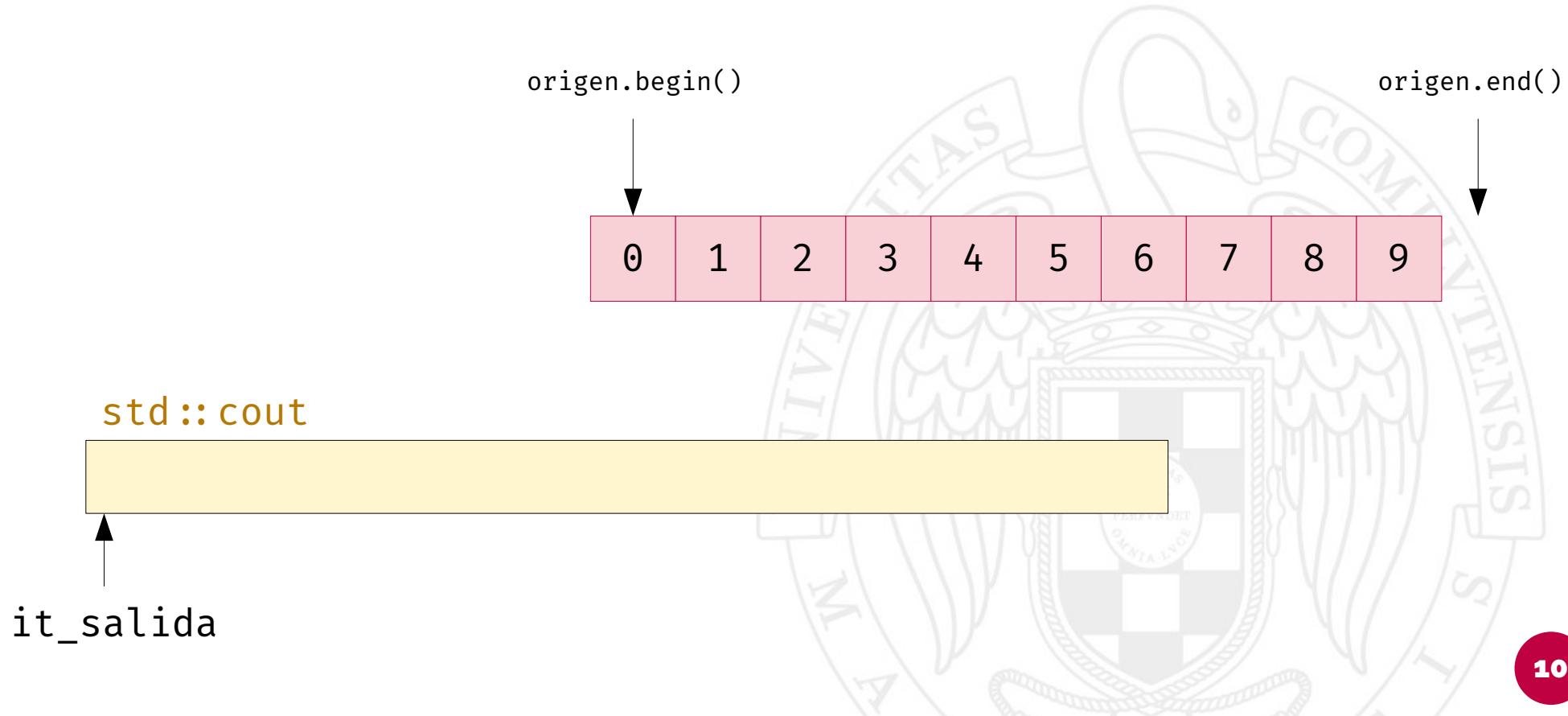


# Utilidad de función `copy()`

- Se puede utilizar para multitud de casos:
  - De un `vector` a un `list` y viceversa.
  - De `vector` a `vector`.
  - De `list` a `deque`.
  - De un `array` a `vector` y viceversa.
  - De un `array` a `list` y viceversa.
  - De un `vector/list/array` a un `ostream_iterator`.

# Otro ejemplo

```
ostream_iterator<int> it_salida(cout, " ");
copy(origen.begin(), origen.end(), it_salida);
```

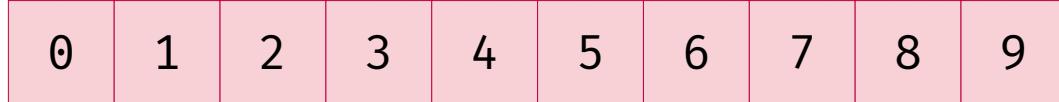


# Otro ejemplo

```
ostream_iterator<int> it_salida(cout, " ");
copy(origen.begin(), origen.end(), it_salida);
```

origen.begin()

origen.end()



std::cout

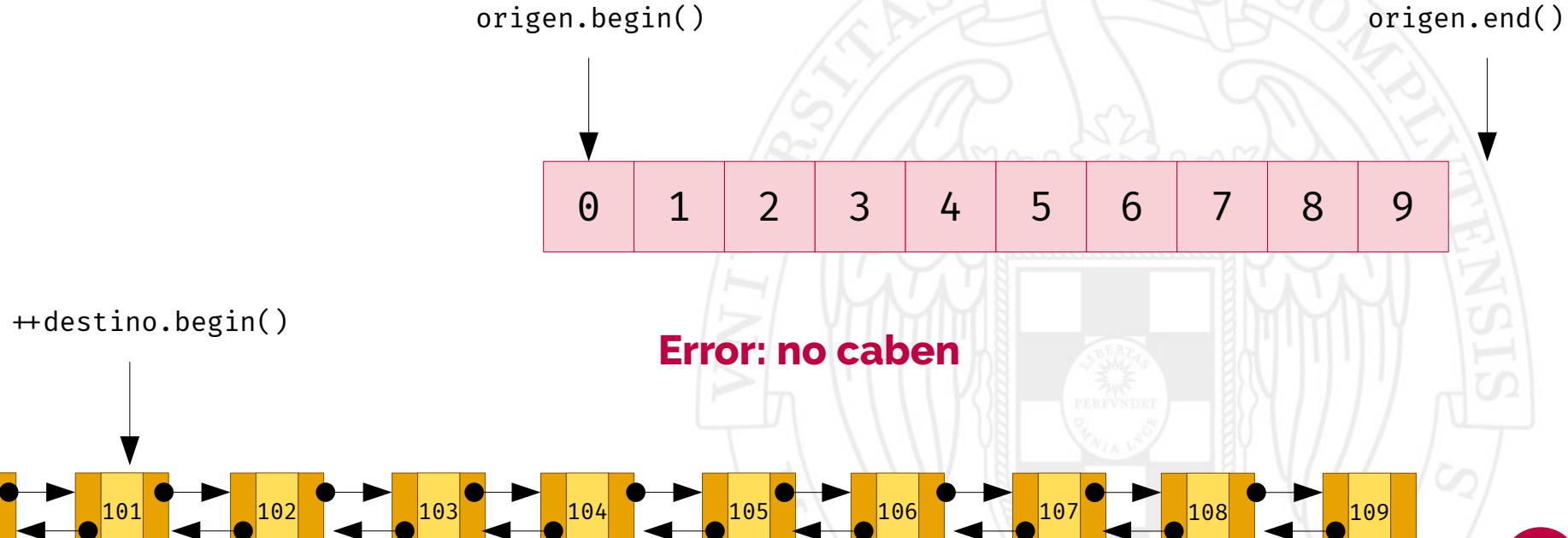
0\_1\_2\_3\_4\_5\_6\_7\_8\_9\_

it\_salida

# Cuidado!

- Para que la copia tenga éxito, el iterador de destino debe poderse incrementar tantas veces como elementos deseen copiarse.

```
copy(origen.begin(), origen.end(), +destino.begin());
```



# Los iteradores back\_insert\_iterator

- Son iteradores de salida que van asociados a un contenedor secuencial (list, vector, deque, etc).
- Cuando se escribe en el iterador, se añade un elemento al contenedor.
- Cuando se incrementa el iterador, no se hace nada.

# Ejemplo

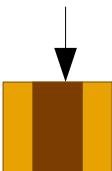
```
int main() {
    vector<int> origen;
    list<int> lista_destino;

    // inicializar origen
    ...
    // suponemos que lista_destino queda vacía

    back_insert_iterator<list<int>> it_dest(lista_destino);
    copy(origen.begin(), origen.end(), it_dest);

    imprimir(cout, lista_destino);
}
```

it\_dest



0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

# Ejemplo

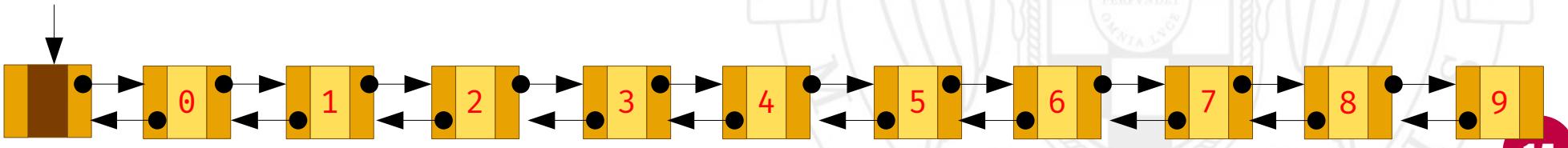
```
int main() {
    vector<int> origen;
    list<int> lista_destino;

    // inicializar origen
    ...
    // suponemos que lista_destino queda vacía

    back_insert_iterator<list<int>> it_dest(lista_destino);
    copy(origen.begin(), origen.end(), it_dest);

    imprimir(cout, lista_destino);
}
```

it\_dest



# La función sort()

# La función sort()

- También definida en <algorithm>.

`sort(begin, end)`

donde:

- `begin`, `end` son iteradores con acceso aleatorio.
- Ordena ascendente los elementos contenidos entre los iteradores `begin` y `end` (excluyendo este último).
- Utiliza el operador `<` para comparar los elementos.

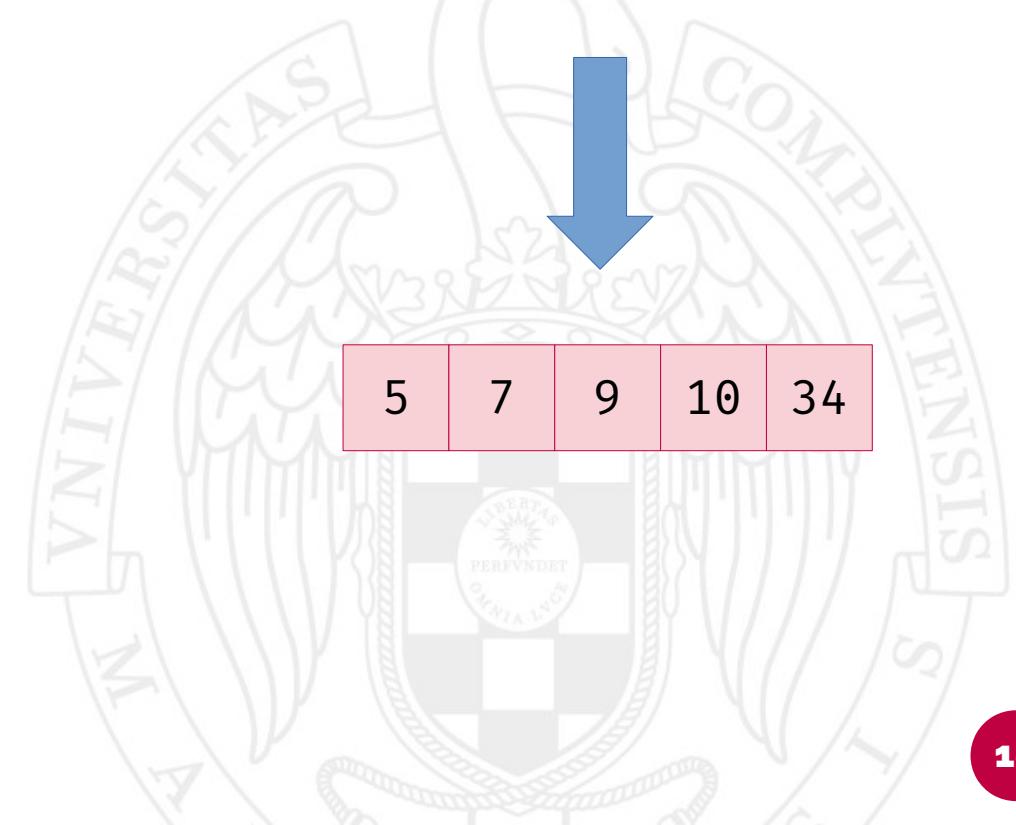
# Ejemplo

```
int main() {  
    vector<int> v;  
    v.push_back(10);  
    v.push_back(34);  
    v.push_back(5);  
    v.push_back(7);  
    v.push_back(9);  
  
    sort(v.begin(), v.end());  
}
```

10	34	5	7	9
----	----	---	---	---



5	7	9	10	34
---	---	---	----	----



# Otro ejemplo

```
int main() {  
    int elems[] = {14, 5, 1, 20, 4, 7};  
    sort(elems, elems + 6);  
}
```

14	5	1	20	4	7
----	---	---	----	---	---



1	4	5	7	14	20
---	---	---	---	----	----

# Más funciones en <algorithm>

- `find(begin, end, value)`
- `fill(begin, end, value)`
- `unique(begin, end)`
- `binary_search(begin, end, value)`
- `max(begin, end)`

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Punteros inteligentes

Manuel Montenegro Montes

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

# ¿Qué es un puntero inteligente?

- Es un TAD que permite las mismas operaciones que un puntero, pero añadiendo nuevas características.
- En particular se encarga de liberar automáticamente el objeto apuntado por él, sin que tengamos que hacerlo nosotros mediante `delete`.
- Las librerías de C++ definen dos tipos de punteros inteligentes en el fichero de cabecera `<memory>`:
  - `std :: unique_ptr<T>` - Puntero exclusivo a un dato de tipo T.  
No puede haber otros punteros apuntando al mismo dato.
  - `std :: shared_ptr<T>` - Puntero compartido a un dato de tipo T.  
Se permiten otros punteros apuntando al mismo dato.

# Recordatorio: clase Fecha

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo);  
    Fecha(int anyo);  
    Fecha();  
  
    int get_dia() const;  
    void set_dia(int dia);  
    int get_mes() const;  
    void set_mes(int mes);  
    int get_anyo() const;  
    void set_anyo(int anyo);  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};  
  
std::ostream & operator<<(std::ostream &out, const Fecha &f);
```



# Punteros exclusivos – std :: unique\_ptr

# Puntero normal vs unique\_ptr

- Ejemplo: crear un objeto en el heap mediante un puntero normal:

```
new Fecha(25, 12, 2019)
```

Esto devuelve un valor de tipo `Fecha *`.

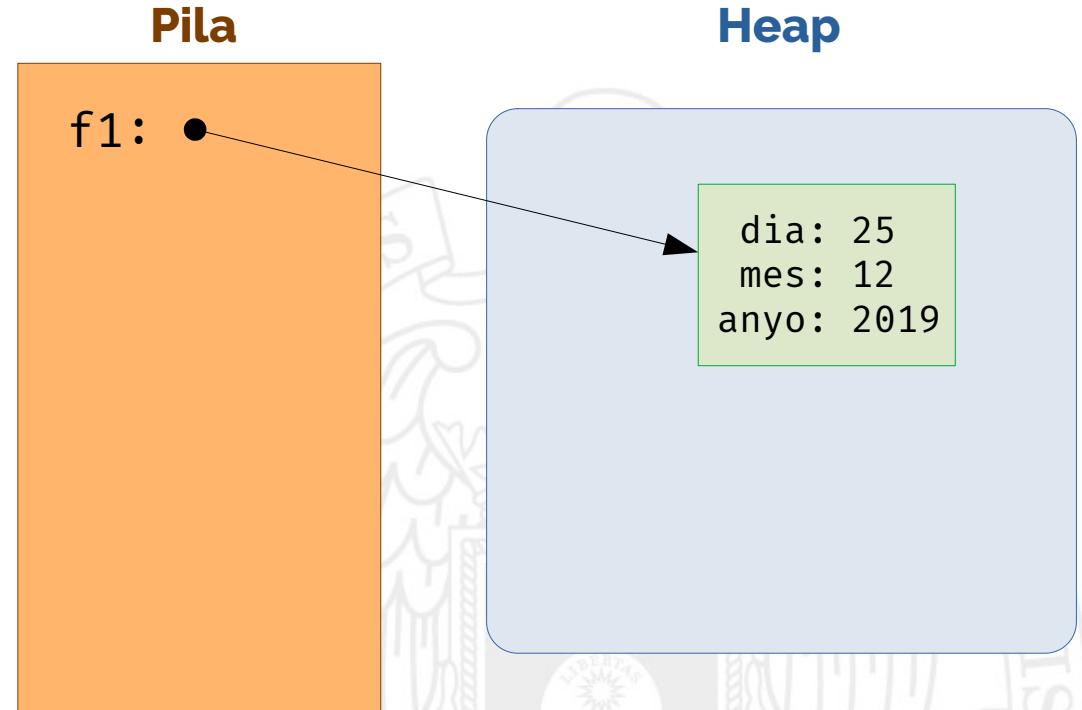
- Ejemplo: crear un objeto en el heap mediante un puntero exclusivo:

```
std::make_unique<Fecha>(25, 12, 2019)
```

Esto devuelve un objeto de tipo `std::unique_ptr<Fecha>`.

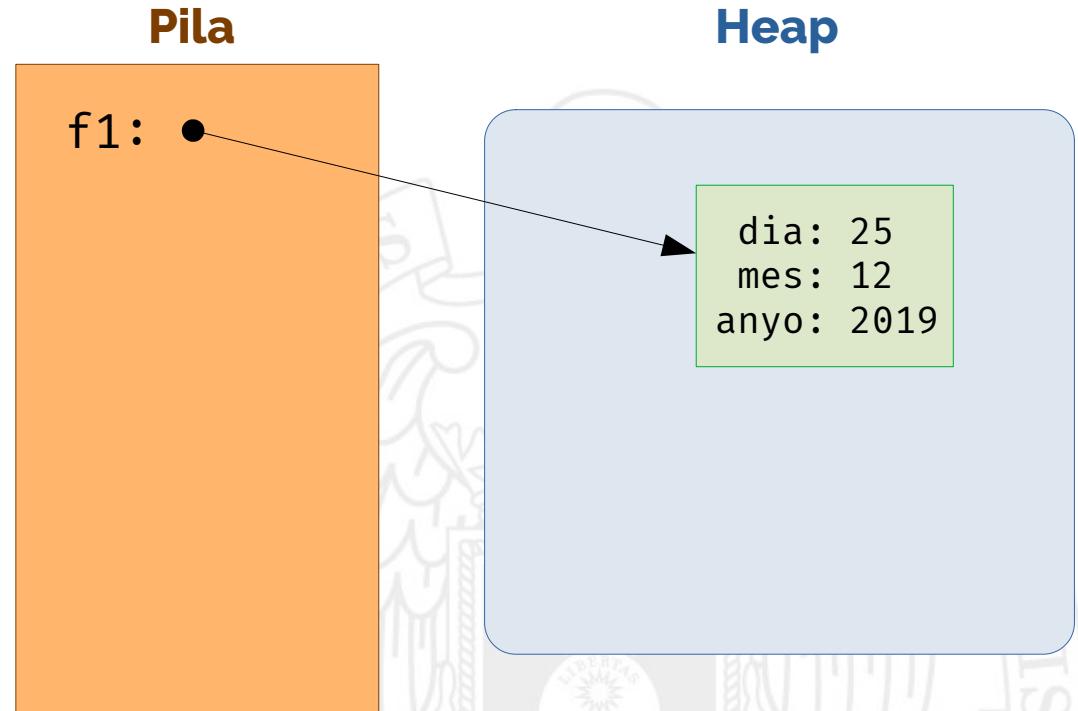
# Ejemplo

```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);
```



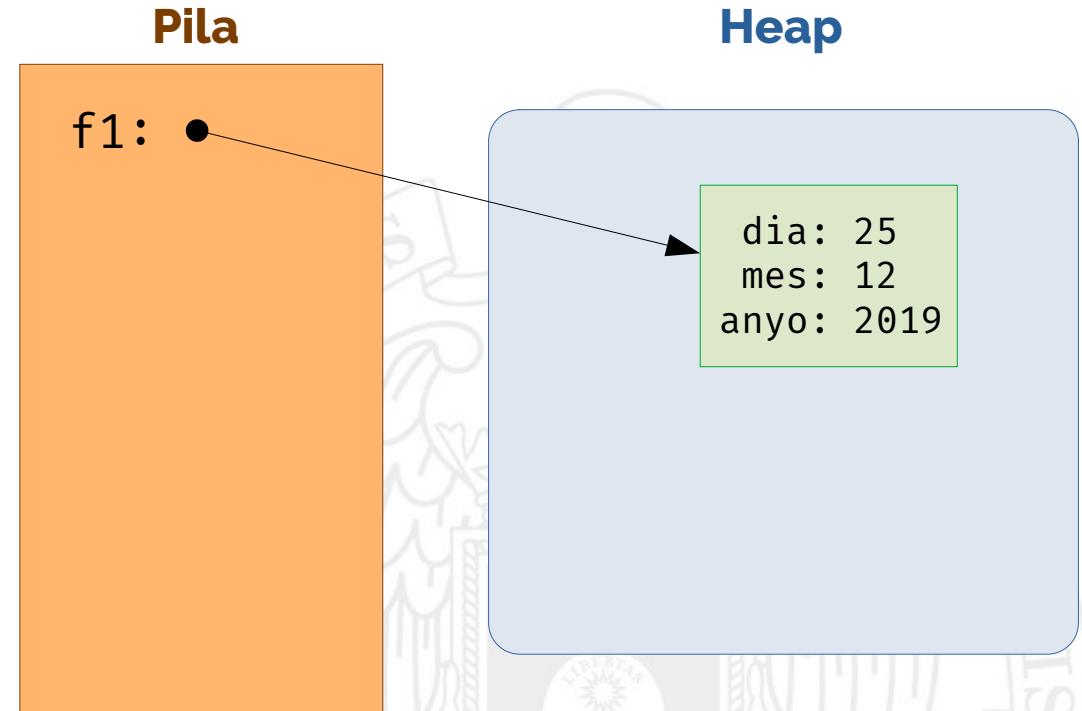
# Ejemplo

```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);  
  
if (f1 != nullptr) {  
    std::cout << f1->get_anyo() << std::endl;  
    std::cout << *f1 << std::endl;  
}
```



# Un unique\_ptr no puede ser copiado

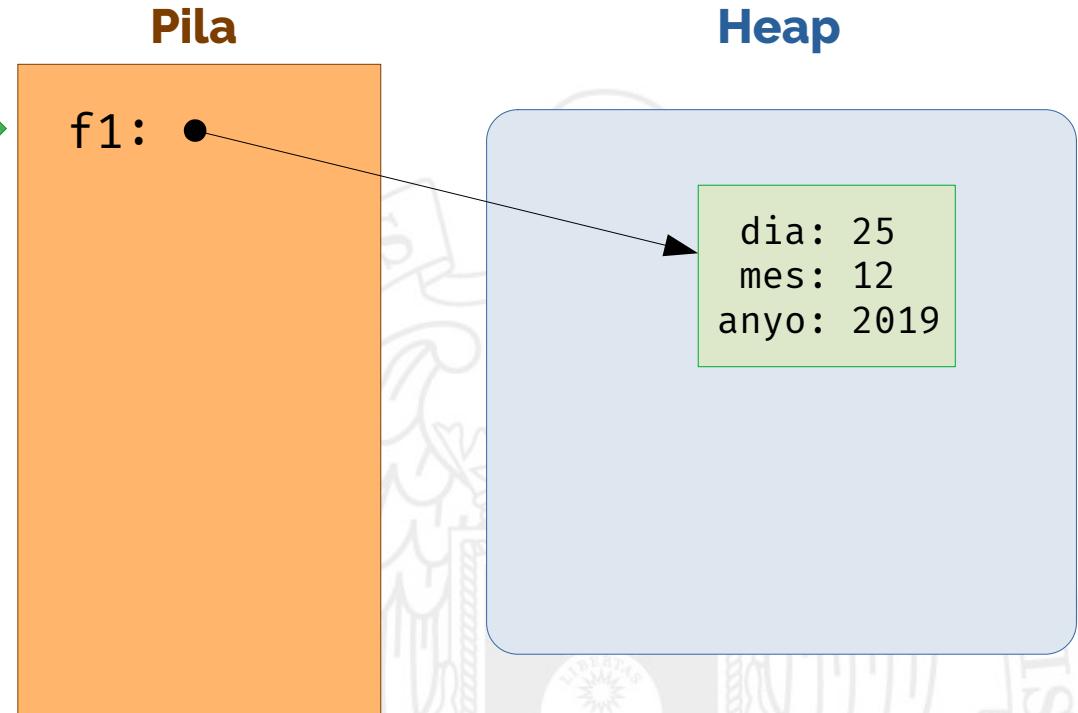
```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);  
  
std::unique_ptr<Fecha> f2 = f1; 
```



# Un unique\_ptr puede ser transferido

```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);
```

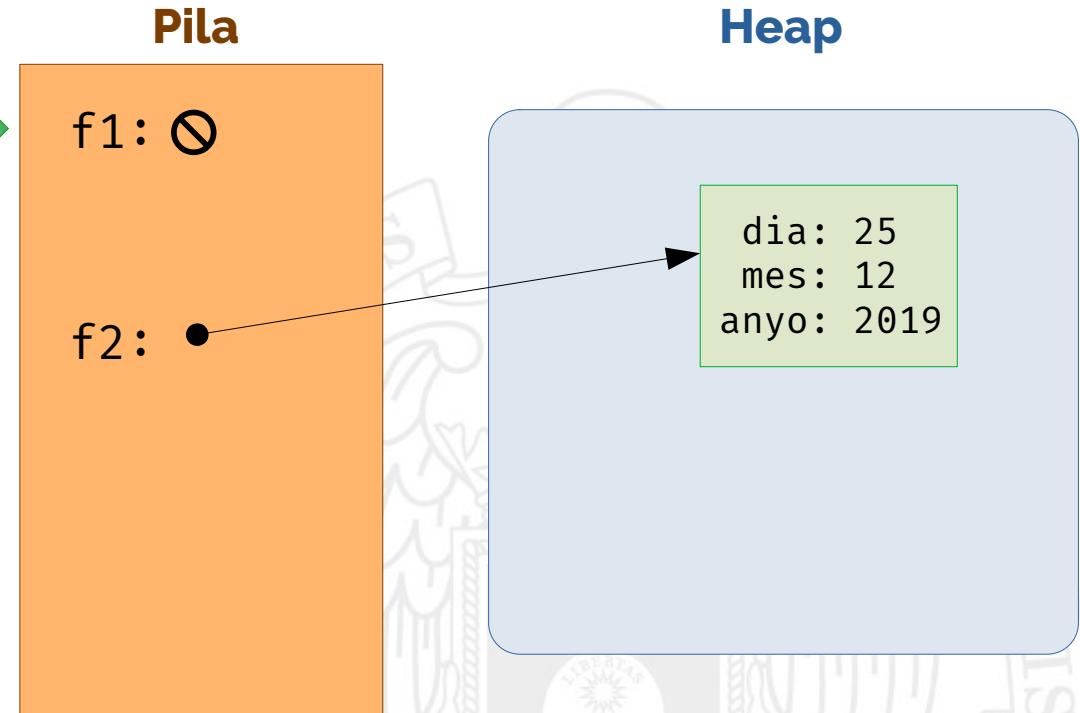
```
std::unique_ptr<Fecha> f2 = std::move(f1); ✓
```



# Un unique\_ptr puede ser transferido

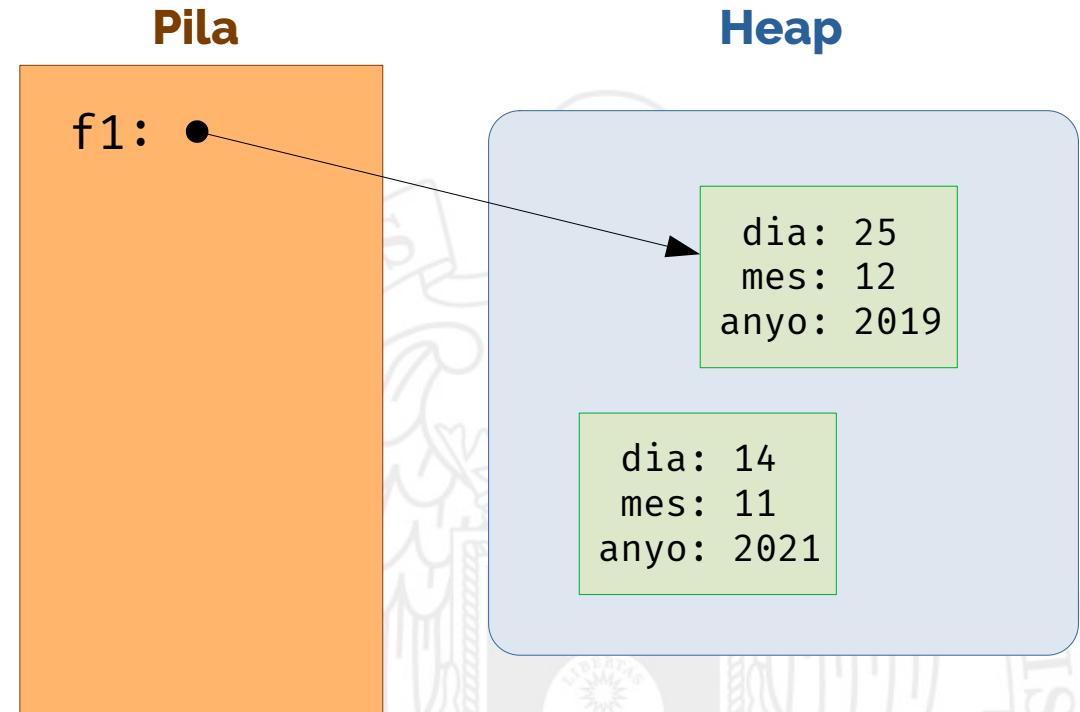
```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);
```

```
std::unique_ptr<Fecha> f2 = std::move(f1); ✓
```



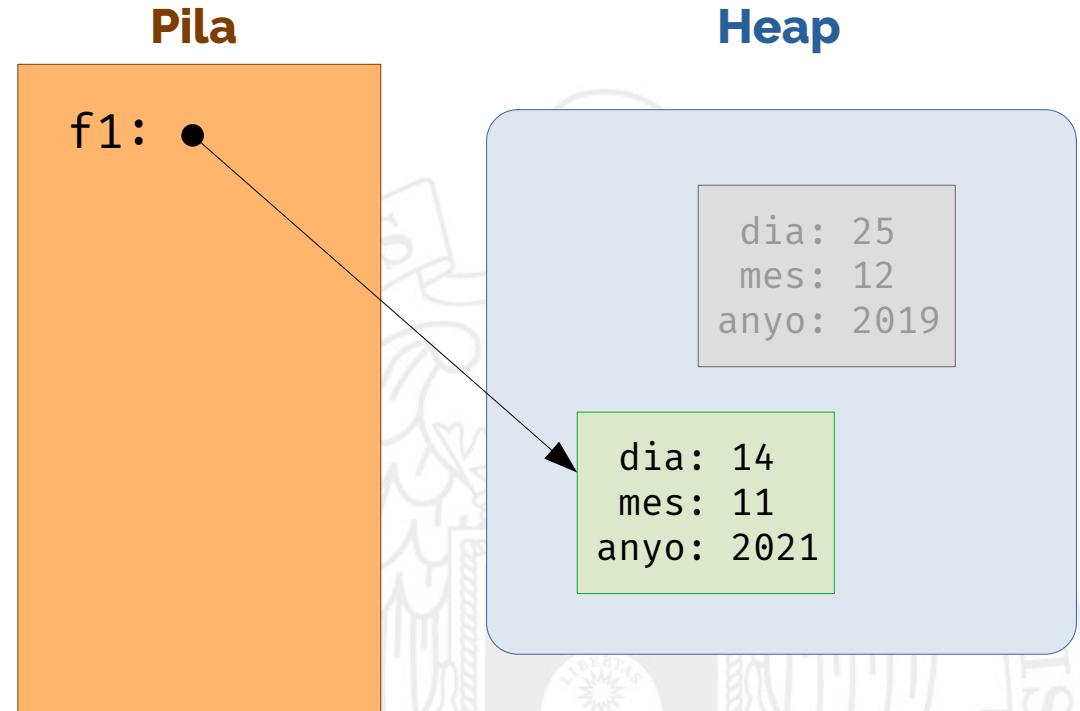
# Reasignando un unique\_ptr

```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);  
  
f1 = std::make_unique<Fecha>(14, 11, 2021);
```



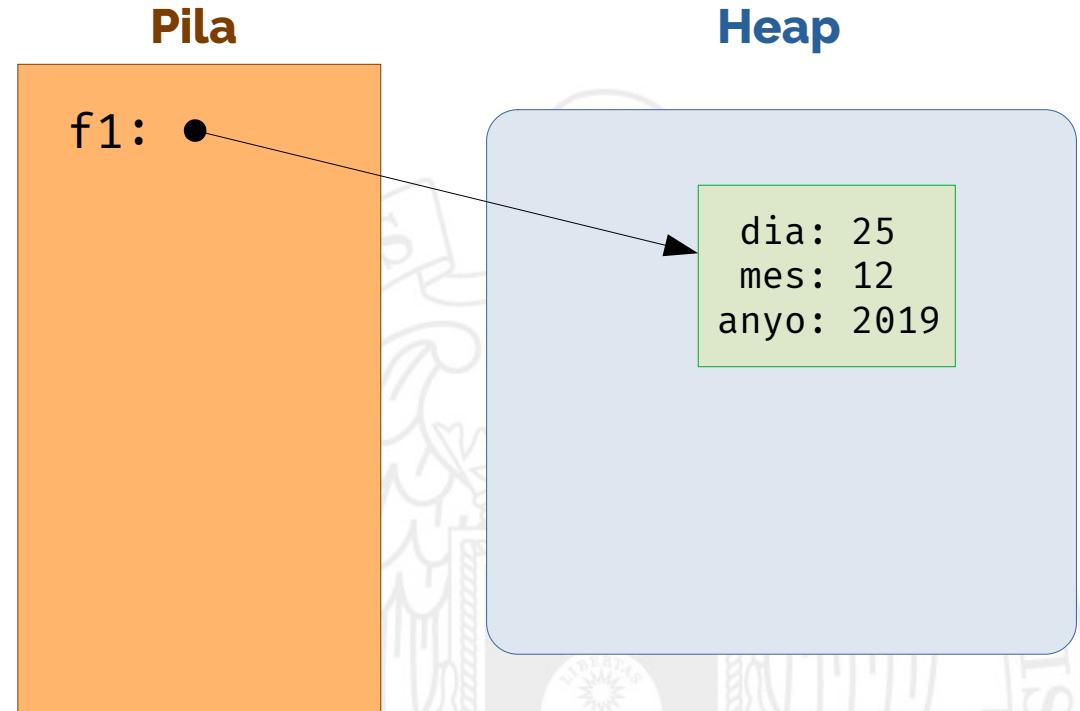
# Reasignando un unique\_ptr

```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);  
  
f1 = std::make_unique<Fecha>(14, 11, 2021);
```



# Reasignando un unique\_ptr

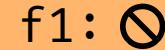
```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);  
  
f1 = nullptr;
```



# Reasignando un unique\_ptr

```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);  
  
f1 = nullptr;
```

Pila

f1: 

Heap

dia: 25  
mes: 12  
anyo: 2019

# Punteros compartidos – std :: shared\_ptr

# Crear un `shared_ptr`

- Para crear un objeto en el heap mediante un puntero compartido:

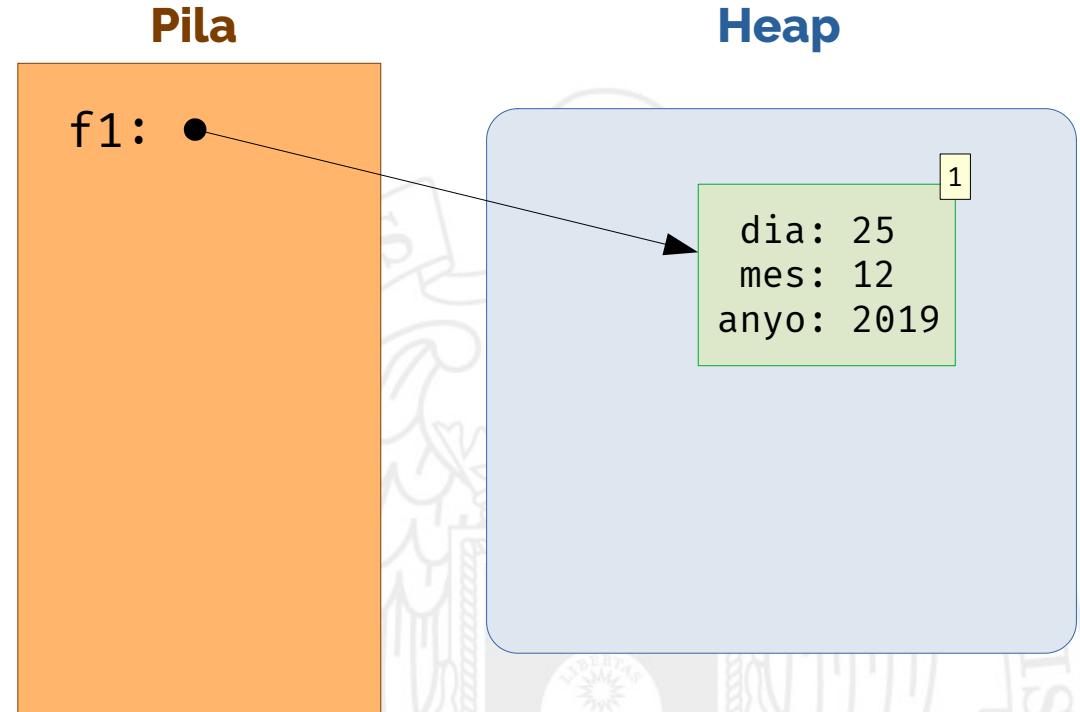
```
std :: make_shared<Fecha>(25, 12, 2019)
```

Esto devuelve un objeto de tipo `std :: shared_ptr<Fecha>`.

- Los objetos del *heap* apuntados por un puntero compartido llevan un **contador de referencias** que indica el número de punteros compartidos que apuntan hacia él.
  - Cuando este contador llega a 0, el objeto se libera.

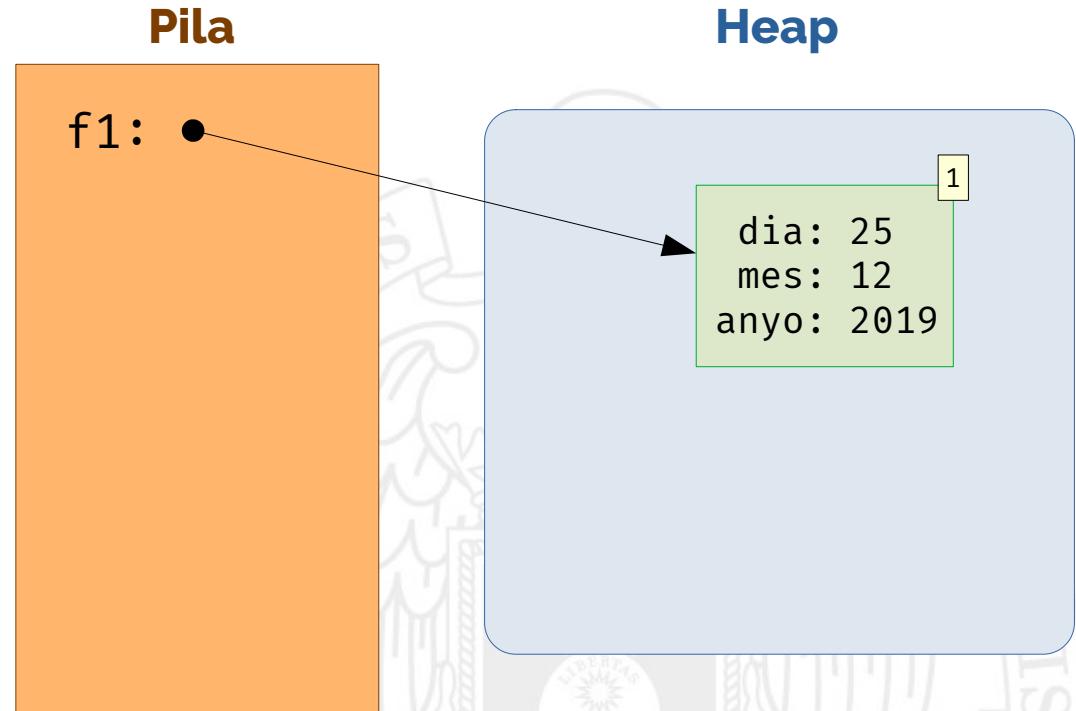
# Ejemplo

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);
```



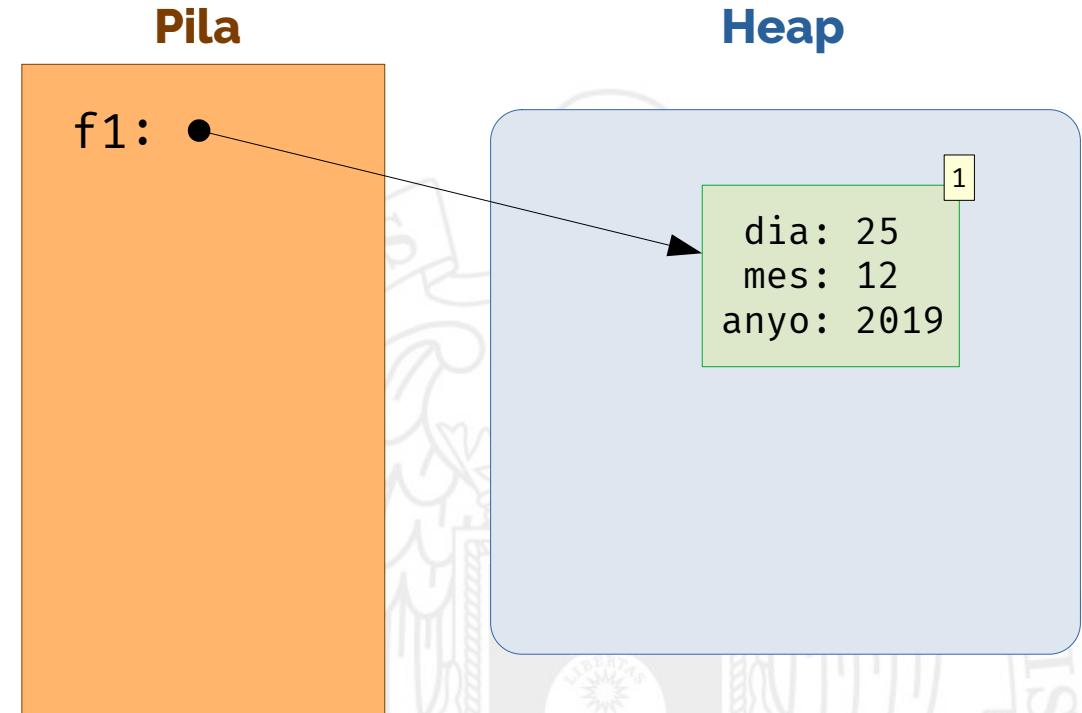
# Ejemplo

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
if (f1 != nullptr) {  
    std::cout << f1->get_anyo() << std::endl;  
    std::cout << *f1 << std::endl;  
}
```



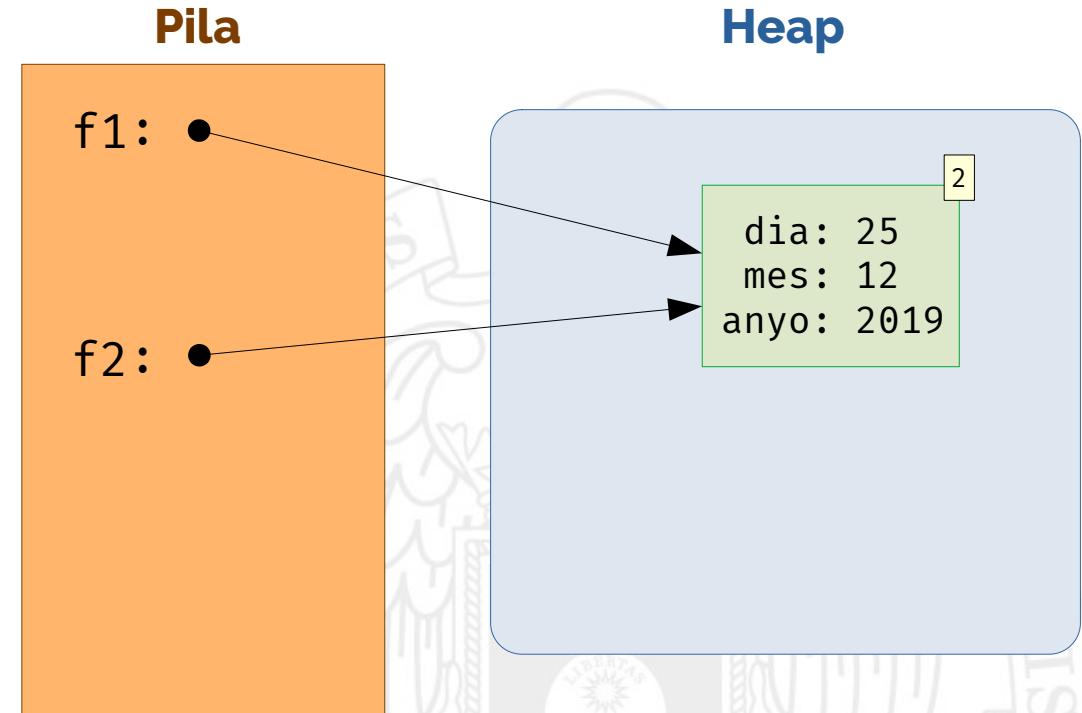
# Copia de un shared\_ptr

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
std::shared_ptr<Fecha> f2 = f1;
```



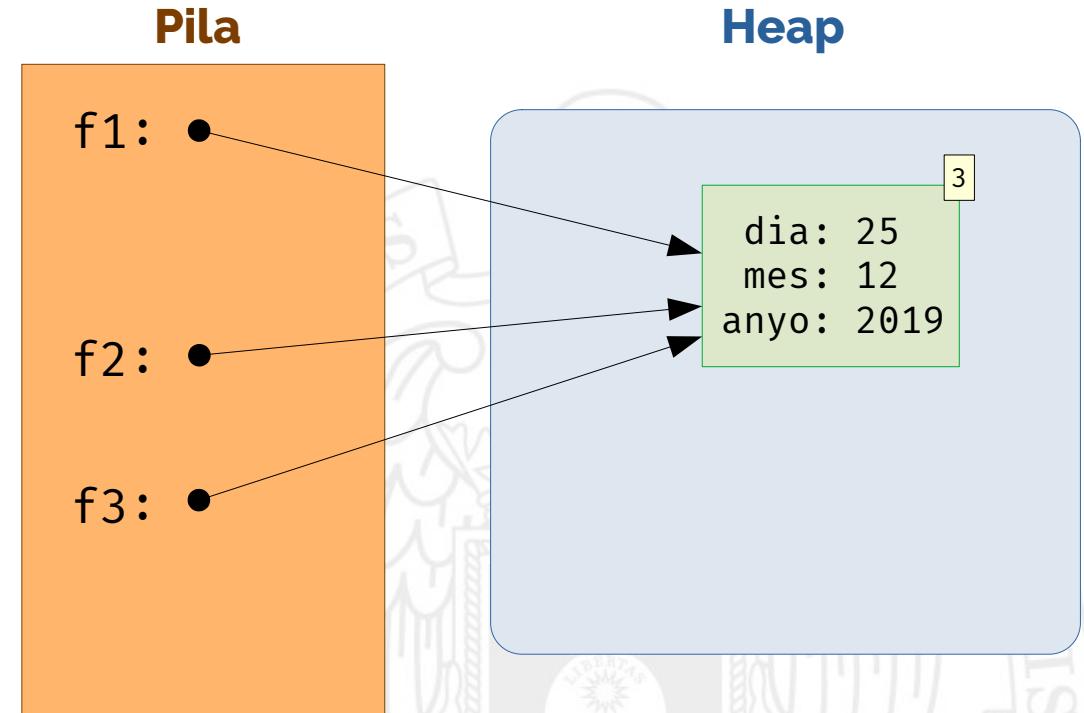
# Copia de un shared\_ptr

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
std::shared_ptr<Fecha> f2 = f1;
```



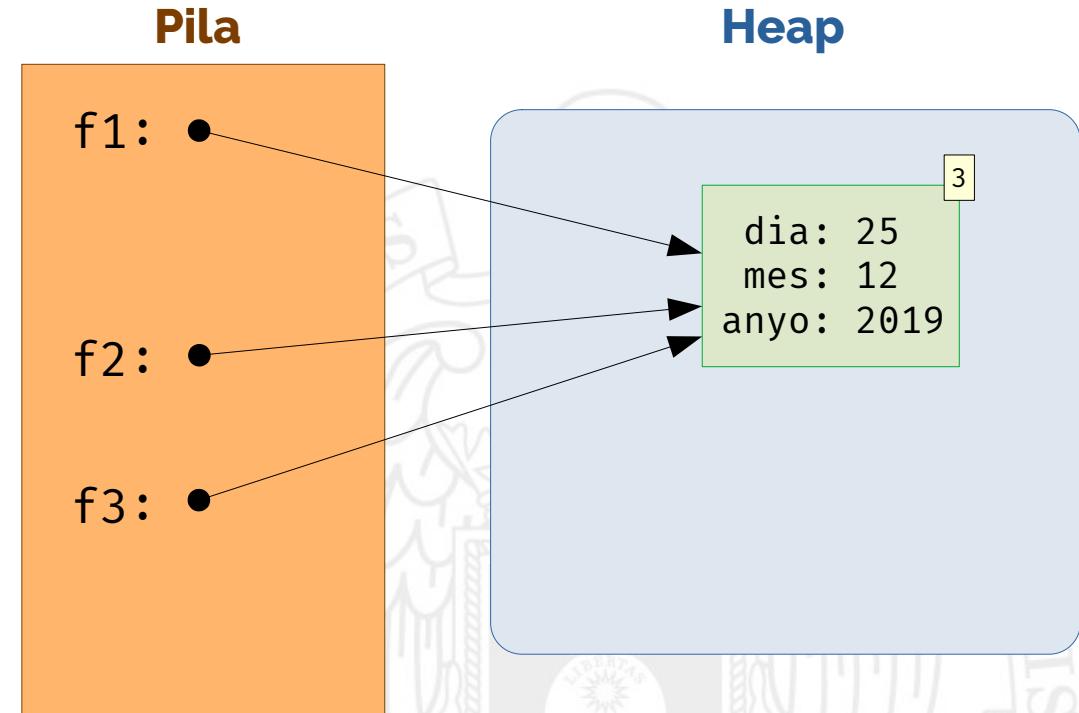
# Copia de un shared\_ptr

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
std::shared_ptr<Fecha> f2 = f1;  
std::shared_ptr<Fecha> f3 = f2;
```



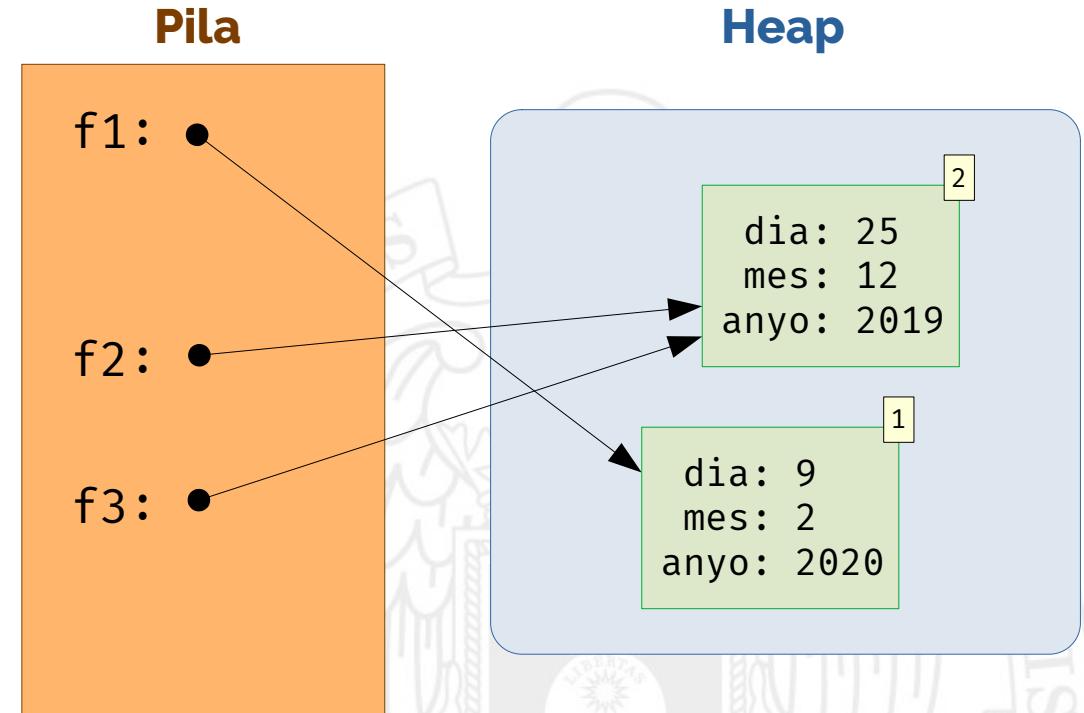
# Copia de un shared\_ptr

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
std::shared_ptr<Fecha> f2 = f1;  
std::shared_ptr<Fecha> f3 = f2;  
  
f1 = std::make_shared<Fecha>(9, 2, 2020);
```



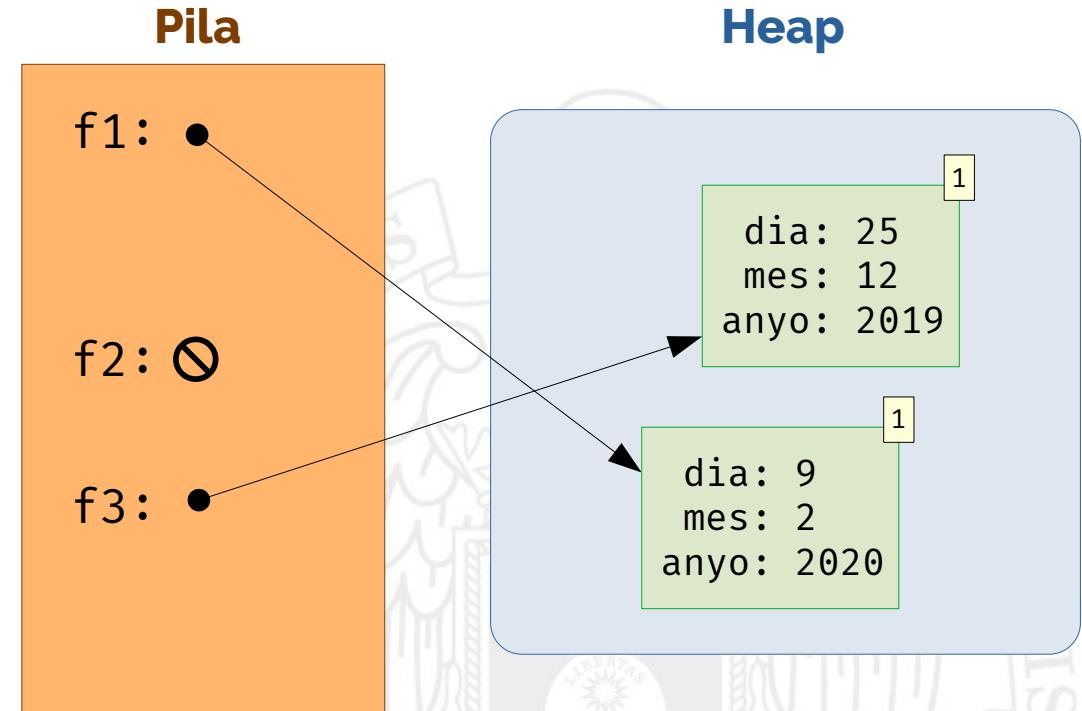
# Copia de un shared\_ptr

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
std::shared_ptr<Fecha> f2 = f1;  
std::shared_ptr<Fecha> f3 = f2;  
  
f1 = std::make_shared<Fecha>(9, 2, 2020);
```



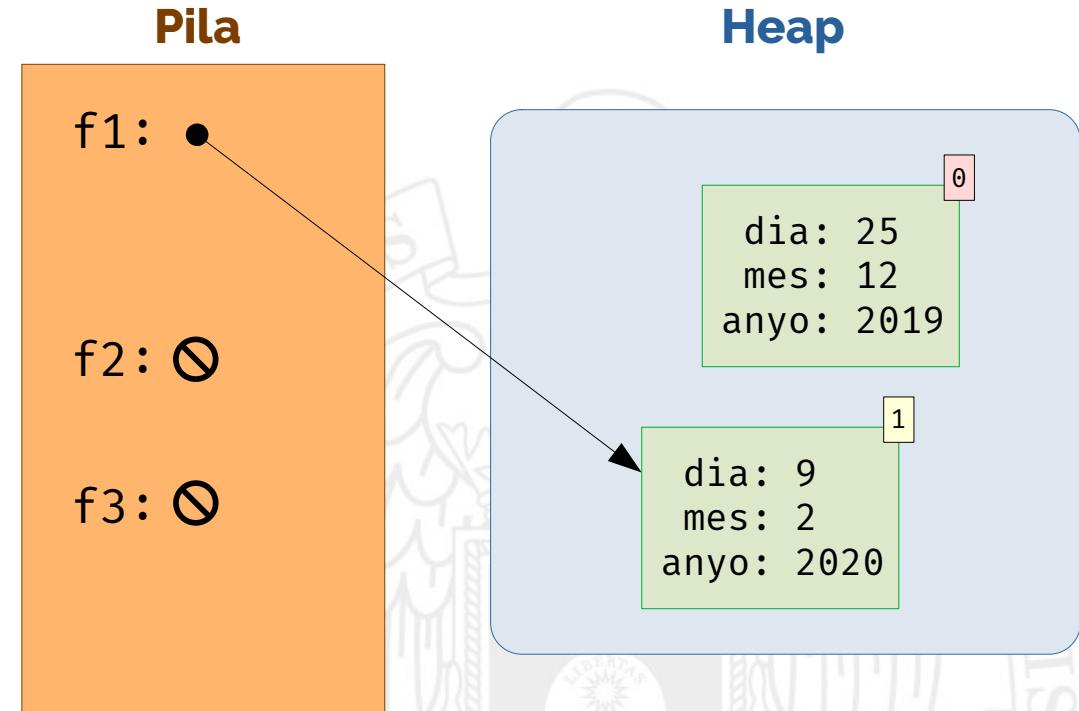
# Copia de un shared\_ptr

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
std::shared_ptr<Fecha> f2 = f1;  
std::shared_ptr<Fecha> f3 = f2;  
  
f1 = std::make_shared<Fecha>(9, 2, 2020);  
f2 = nullptr;
```



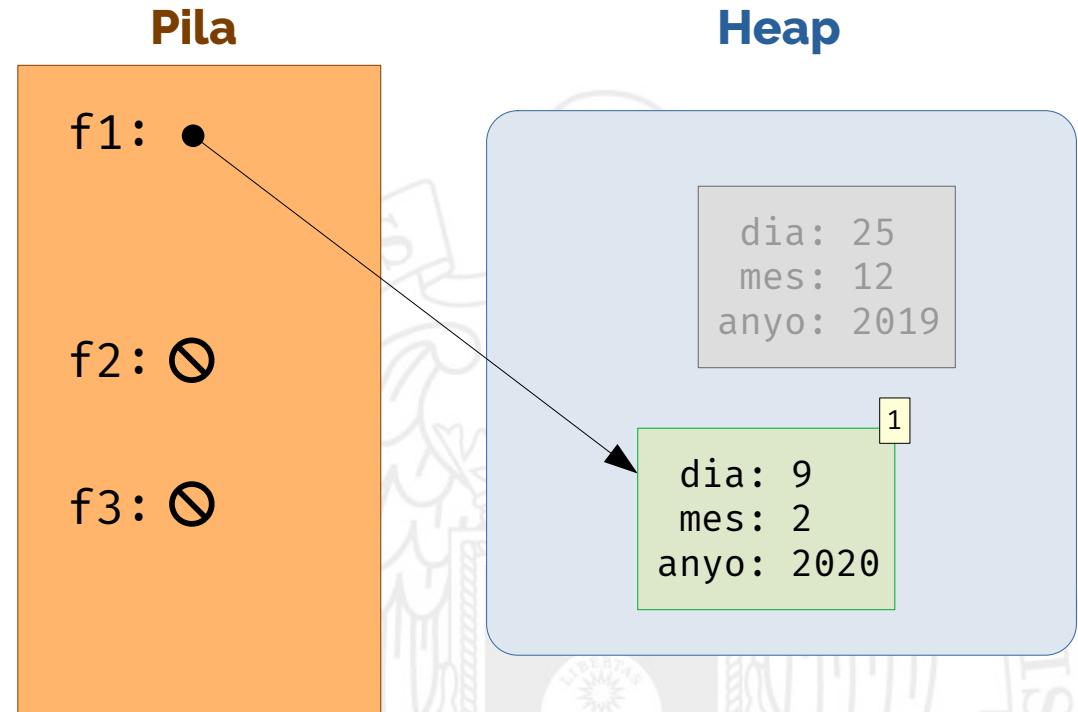
# Copia de un shared\_ptr

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
std::shared_ptr<Fecha> f2 = f1;  
std::shared_ptr<Fecha> f3 = f2;  
  
f1 = std::make_shared<Fecha>(9, 2, 2020);  
f2 = nullptr;  
f3 = nullptr;
```

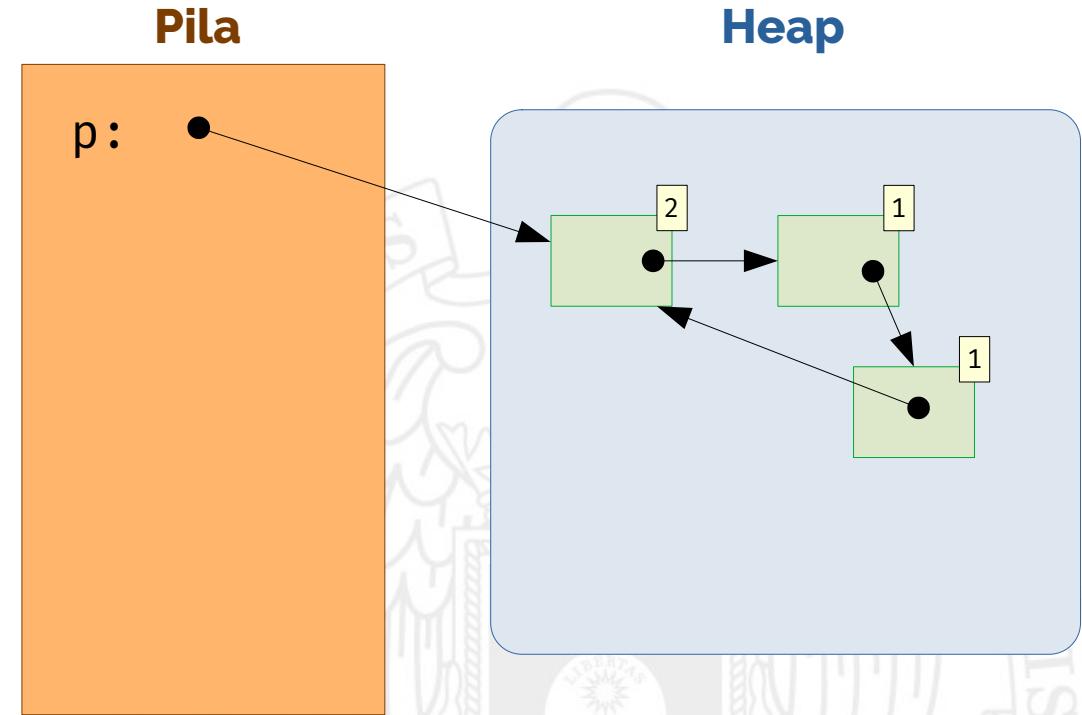


# Copia de un shared\_ptr

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
std::shared_ptr<Fecha> f2 = f1;  
std::shared_ptr<Fecha> f3 = f2;  
  
f1 = std::make_shared<Fecha>(9, 2, 2020);  
f2 = nullptr;  
f3 = nullptr;
```



# ¡Cuidado con las referencias circulares!



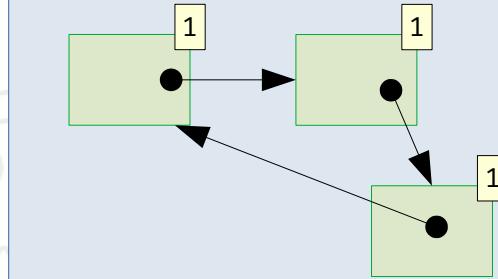
# ¡Cuidado con las referencias circulares!

```
p = nullptr;
```

Pila

p:  $\text{\textcircled{0}}$

Heap



ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Funciones de orden superior

Manuel Montenegro Montes

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

# Ejercicio

- Función que recibe una lista de enteros y elimina los números pares de la misma.

```
bool es_par(int x) { return x % 2 == 0; }

void eliminar_pares(std::list<int> &elems) {
    auto it = elems.begin();
    while (it != elems.end()) {
        if (es_par(*it)) {
            it = elems.erase(it);
        } else {
            ++it;
        }
    }
}
```

# Ejemplo

```
std::list<int> v1 = {1, 5, 6, 9, 10, 20};  
eliminar_pares(v1);  
std::cout << v1 << std::endl;
```

[1, 5, 9]

# Ejercicio

- Función que recibe una lista de enteros y elimina los números **impares** de la misma.

```
bool es_impar(int x) { return x % 2 == 1; }

void eliminar_impares(std::list<int> &elems) {
    auto it = elems.begin();
    while (it != elems.end()) {
        if (es_impar(*it)) {
            it = elems.erase(it);
        } else {
            ++it;
        }
    }
}
```

# Ejemplo

```
std::list<int> v1 = {1, 5, 6, 9, 10, 20};  
std::list<int> v2 = v1;  
eliminar_pares(v1);  
std::cout << v1 << std::endl;  
  
eliminar_impares(v2);  
std::cout << v2 << std::endl;
```

[1, 5, 9]

[6, 10, 20]

# Ejercicio

- Función que recibe una lista de enteros y elimina los números **positivos** de la misma.

```
bool es_positivo(int x) { return x > 0; }

void eliminar_positivos(std::list<int> &elems) {
    auto it = elems.begin();
    while (it != elems.end()) {
        if (es_positivo(*it)) {
            it = elems.erase(it);
        } else {
            ++it;
        }
    }
}
```

# Ejemplo

```
std::list<int> v1 = {1, 5, 6, 9, 10, 20};  
std::list<int> v2 = v1;  
eliminar_pares(v1);  
std::cout << v1 << std::endl;  
  
eliminar_impares(v2);  
std::cout << v2 << std::endl;  
  
std::list<int> v3 = {-2, 3, 10, -6, 20};  
eliminar_positivos(v3);  
std::cout << v3 << std::endl;
```

[1, 5, 9]

[6, 10, 20]

[-2, -6]

# ¡Cuánta duplicación!

```
void eliminar_positivos(std::list<int> &elems) {
    auto it = elems.begin();
    while (it != elems.end()) {
        if (es_positivo(*it)) {
            it = elems.erase(it);
        } else {
            ++it;
        }
    }
}
```

```
void eliminar_pares(std::list<int> &elems) {
    auto it = elems.begin();
    while (it != elems.end()) {
        if (es_par(*it)) {
            it = elems.erase(it);
        } else {
            ++it;
        }
    }
}
```

```
void eliminar_impar(std::list<int> &elems) {
    auto it = elems.begin();
    while (it != elems.end()) {
        if (es_impar(*it)) {
            it = elems.erase(it);
        } else {
            ++it;
        }
    }
}
```

- La solución para unificar estas tres funciones es **parametrizarlas** en aquello en lo que se diferencian.
- ¡Pero aquí se diferencian en una **función**!
- ¿Es posible pasar funciones como parámetros en C++?

# Sí, es posible, pero...

¿Qué tipo tiene ese parámetro?

```
void eliminar_positivos(std::list<int> &elems) {  
    auto it = elems.begin();  
    while (it != elems.end()) {  
        if (es_positivo(*it)) {  
            it = elems.erase(it);  
        } else {  
            ++it;  
        }  
    }  
}
```



```
void eliminar(std::list<int> &elems, ??? func) {  
    auto it = elems.begin();  
    while (it != elems.end()) {  
        if (func(*it)) {  
            it = elems.erase(it);  
        } else {  
            ++it;  
        }  
    }  
}
```

# Sí, es posible, pero...

¿Qué tipo tiene ese parámetro?

- **Puntero a función**
  - Mecanismo heredado de C.
- **Variable plantilla**
  - Utiliza el mecanismo de plantillas de C++.
  - Dejamos que el compilador infiera el tipo.
  - Compatible con objetos función.

**Siguiente video**

# Uso de variable de plantilla

```
template <typename T>
void eliminar(std::list<int> &elems, T func) {
    auto it = elems.begin();
    while (it != elems.end()) {
        if (func(*it)) {
            it = elems.erase(it);
        } else {
            ++it;
        }
    }
}
```

# Ejemplo

```
bool es_par(int x) { return x % 2 == 0; }
bool es_impar(int x) { return x % 2 == 1; }
bool es_positivo(int x) { return x > 0; }

std::list<int> v1 = {1, 5, 6, 9, 10, 20};
std::list<int> v2 = v1;
eliminar_pares(v1);
std::cout << v1 << std::endl;

eliminar_impares(v2);
std::cout << v2 << std::endl;

std::list<int> v3 = {-2, 3, 10, -6, 20};
eliminar_positivos(v3);
std::cout << v3 << std::endl;
```



# Ejemplo

```
bool es_par(int x) { return x % 2 == 0; }
bool es_impar(int x) { return x % 2 == 1; }
bool es_positivo(int x) { return x > 0; }
```

```
std::list<int> v1 = {1, 5, 6, 9, 10, 20};
std::list<int> v2 = v1;
eliminar_pares(v1); → eliminar(v1, es_par);
std::cout << v1 << std::endl;

eliminar_impares(v2); → eliminar(v2, es_impar);
std::cout << v2 << std::endl;

std::list<int> v3 = {-2, 3, 10, -6, 20};
eliminar_positivos(v3); → eliminar(v3, es_positivo);
std::cout << v3 << std::endl;
```

# Ejemplo

```
bool es_par(int x) { return x % 2 == 0; }
bool es_impar(int x) { return x % 2 == 1; }
bool es_positivo(int x) { return x > 0; }

std::list<int> v1 = {1, 5, 6, 9, 10, 20};
std::list<int> v2 = v1;
eliminar(v1, es_par);
std::cout << v1 << std::endl;

eliminar(v2, es_impar);
std::cout << v2 << std::endl;

std::list<int> v3 = {-2, 3, 10, -6, 20};
eliminar(v3, es_positivo);
std::cout << v3 << std::endl;
```

[1, 5, 9]

[6, 10, 20]

[-2, -6]

# Orden superior

- Cuando una función o método  $f$  recibe otras funciones como parámetros, o devuelve una función como resultado, decimos que  $f$  es una función o método de **orden superior**.
- La función `eliminar` es de orden superior.



# Una pequeña generalización

- Podemos hacer que eliminar funcione sobre listas de cualquier tipo; no solo sobre listas de `int`.

```
template <typename T, typename U>
void eliminar(std::list<U> &elems, T func) {
    auto it = elems.begin();
    while (it != elems.end()) {
        if (func(*it)) {
            it = elems.erase(it);
        } else {
            ++it;
        }
    }
}
```

# Ejemplo

```
std::list<Fecha> v4 = { {25, 12, 2010}, {10, 21, 2020}, {25, 12, 1900}, {1, 1, 2000} };  
eliminar(v4, es_navidad);  
std::cout << v4 << std::endl;
```

[10/21/2020, 01/01/2000]



ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Los tipos pair y tuple

Manuel Montenegro Montes

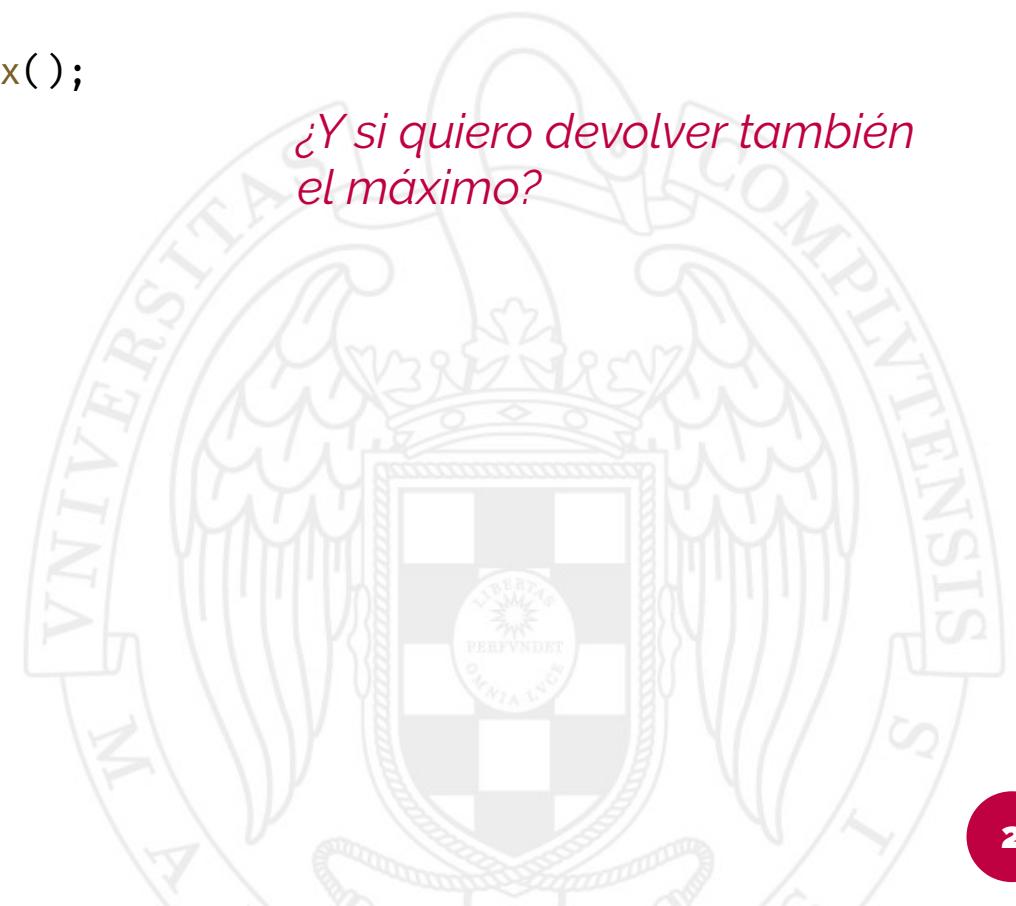
Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Ejemplo

- Calcular el elemento mínimo de un array.

```
int min(int *array, int longitud) {  
    int min = std::numeric_limits<int>::max();  
  
    for (int i = 0; i < longitud; i++) {  
        min = std::min(min, array[i]);  
    }  
  
    return min;  
}
```

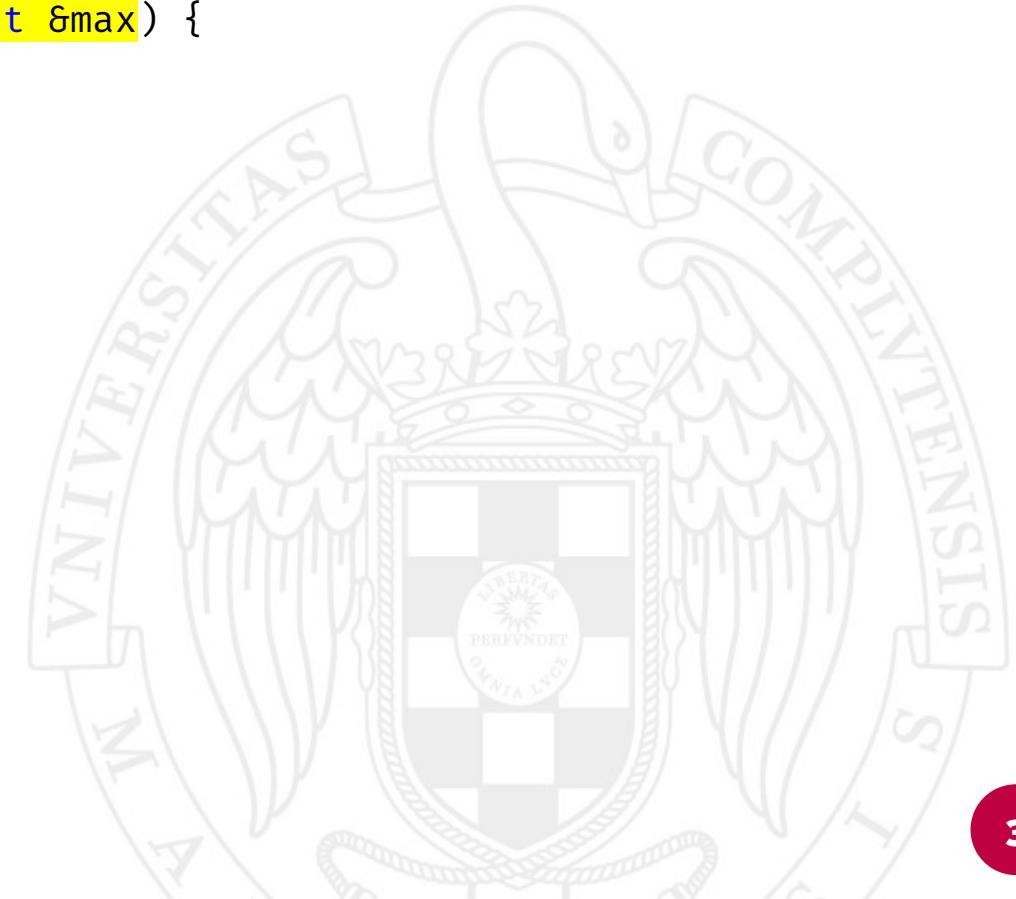
*¿Y si quiero devolver también el máximo?*



# Ejemplo

- Calcular el elemento mínimo y máximo de un array.

```
int min_max(int *array, int longitud, int &max) {  
}  
}
```



# Ejemplo

- Calcular el elemento mínimo y máximo de un array.

```
void min_max(int *array, int longitud, int &min, int &max) {  
    min = std::numeric_limits<int>::max();  
    max = std::numeric_limits<int>::min();  
  
    for (int i = 0; i < longitud; i++) {  
        min = std::min(min, array[i]);  
        max = std::max(max, array[i]);  
    }  
}
```

- ¿Son parámetros de salida o de E/S?
- Llamadas a función:

```
int min, max;  
min_max(arr, longitud, min, max);
```

# Múltiples resultados

- ¿Cómo podemos especificar varios valores de retorno para una función, sin tener que recurrir a parámetros de salida?



# Tipo específico

```
struct MinMaxResult {  
    int min;  
    int max;  
};  
  
MinMaxResult min_max(int *array, int longitud) {  
    MinMaxResult res;  
    res.min = std::numeric_limits<int>::max();  
    res.max = std::numeric_limits<int>::min();  
  
    for (int i = 0; i < longitud; i++) {  
        res.min = std::min(res.min, array[i]);  
        res.max = std::max(res.max, array[i]);  
    }  
  
    return res;  
}
```

# Tipo específico

- Llamada a la función:

```
MinMaxResult r = min_max(arr, longitud);
std::cout << "Min = " << r.min << " | Max = " << r.max;
```

- Problema: tener que definir un tipo específico.

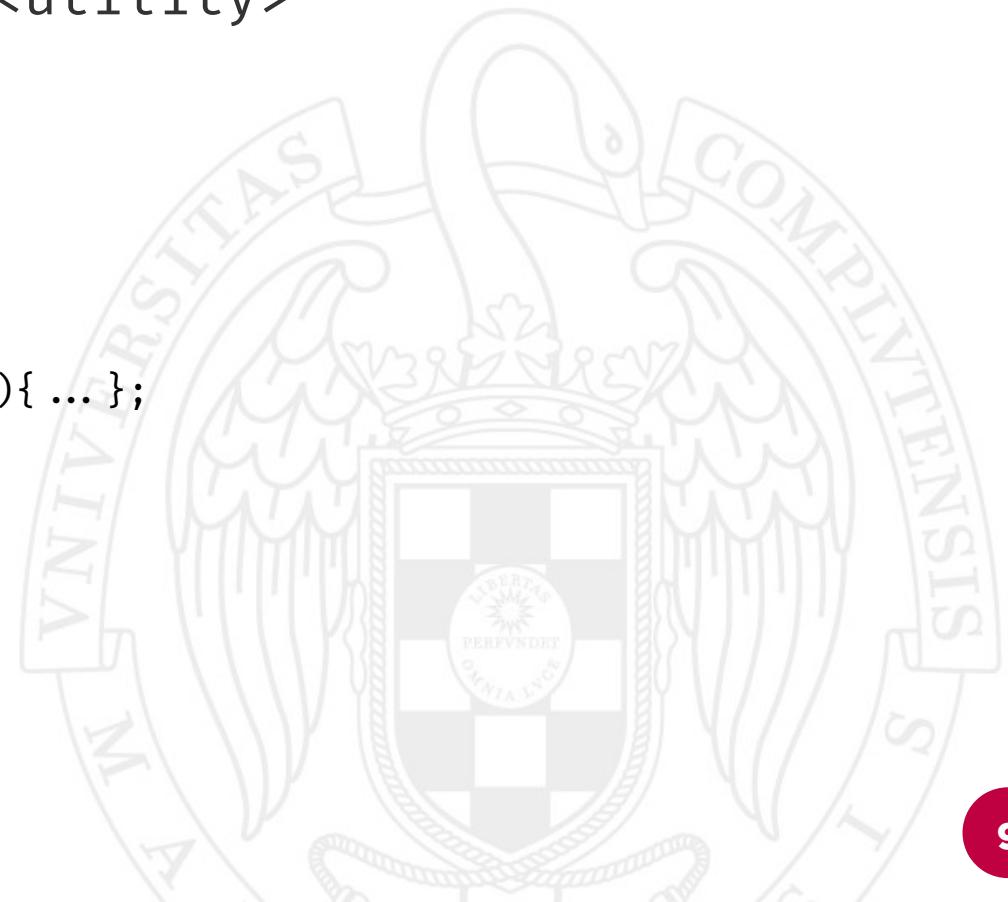
# Pares – std :: pair

# La clase pair

- Denota un par de elementos ( $x, y$ ), que pueden ser de distinto tipo.
- Definida en el fichero de cabecera `<utility>`

```
template <typename T1, typename T2>
class pair {
public:
    T1 first;
    T2 second;

    pair(const T1 &first, const T2 &second){ ... };
    ...
};
```



# La clase pair

```
std::pair<int, int> min_max(int *array, int longitud) {
    int min = std::numeric_limits<int>::max();
    int max = std::numeric_limits<int>::min();

    for (int i = 0; i < longitud; i++) {
        min = std::min(min, array[i]);
        max = std::max(max, array[i]);
    }

    return std::pair<int, int>(min, max);
}
```



# La clase pair

```
std::pair<int, int> min_max(int *array, int longitud) {
    int min = std::numeric_limits<int>::max();
    int max = std::numeric_limits<int>::min();

    for (int i = 0; i < longitud; i++) {
        min = std::min(min, array[i]);
        max = std::max(max, array[i]);
    }

    return {min, max};
}
```



# La clase pair

- Llamada a la función:

```
std::pair<int, int> p = min_max(arr, longitud);
std::cout << "Min = " << p.first << " | Max = " << p.second;
```

- Sintaxis abreviada (*structured binding declaration*) de C++17.

```
auto [min, max] = min_max(arr, longitud);
std::cout << "Min = " << min << " | Max = " << max << std::endl;
```

- En Visual Studio 2019 es necesario activar la opción /std:c++17 o /std:c++latest.

# La clase pair

- Hace explícitos los valores de salida.
- No requiere declarar ninguna clase.
- Pero... conviene documentar el significado de las componentes:

```
// Devuelve un par de enteros.  
// - La primera componente es el valor mínimo del array  
// - La segunda componente es el valor máximo del array  
  
std::pair<int, int> min_max(int *array, int longitud) {  
    ...  
}
```

*¿Y si la función devuelve más de dos valores?*

## Tuplas – std :: tuple

# La clase tuple

- Definida en el fichero de cabecera <tuple>

```
// Devuelve una tupla con tres componentes:  
// - La primera componente es el valor mínimo del array  
// - La segunda componente es el valor máximo del array  
// - La tercera componente es la suma de los valores del array  
  
std::tuple<int, int, int> min_max_sum(int *array, int longitud) {  
    int min = std::numeric_limits<int>::max();  
    int max = std::numeric_limits<int>::min();  
    int sum = 0;  
  
    for (int i = 0; i < longitud; i++) {  
        min = std::min(min, array[i]);  
        max = std::max(max, array[i]);  
        sum += array[i];  
    }  
  
    return {min, max, sum};  
}
```

# La clase tuple

- Llamada:

```
auto [min, max, sum] = min_max_sum(arr, longitud);
std::cout << "Min = " << min << " | Max = " << max << " | Sum = " << sum;
```



ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Objetos función

Manuel Montenegro Montes

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

# Recordatorio

- Función que recibe una lista, una función booleana y elimina aquellos elementos para los que la función devuelve `true`.

```
template <typename T, typename U>
void eliminar(std::list<U> &elems, T func) {
    auto it = elems.begin();
    while (it != elems.end()) {
        if (func(*it)) {
            it = elems.erase(it);
        } else {
            ++it;
        }
    }
}
```

# ¿Qué puedo pasar como parámetro func?

```
template <typename T, typename U>
void eliminar(std::list<U> &elems, T func) {
    ...
    if (func(*it)) { ... }
    ...
}
```

- Cualquier cosa sobre la que se pueda realizar una llamada.
  - En particular, cualquier función que acepte un solo parámetro.
  - ...¿algo más?

# ¿Qué operadores pueden sobrecargarse?

+ - \* / % ^ & | << >>  
== <= >= != < > && || !  
= += -= \*= /=  
++ --  
[] () →  
new delete  
etc.

# Sobrecarga del operador ()

- C++ permite sobrecargar el operador () .

```
class Prueba {  
public:  
    void operator()(parametros) { ... }  
};
```

- Este operador es invocado cuando se evalúa una expresión de la forma x( args ), donde x es una instancia de la clase Prueba.

# Ejemplo

```
class SumaUno {  
public:  
    int operator()(int x) { return x + 1; }  
};
```

- Supongamos que declaro una instancia de la clase SumaUno:  
`SumaUno s;`
- La expresión `s(3)` equivale a `s.operator()(3)` y se evaluará al valor 4.
- ¡Ojo! `s` no es una función; es un objeto que se comporta como una función.

# Objetos función

- Un **objeto función** es una instancia de una clase que sobrecarga el operador ( ).
- En nuestro ejemplo:

SumaUno s;

s es un objeto función.



# Uso de los objetos función

- Los objetos función pueden ser utilizados en cualquier contexto en el que se requiera una función.

```
template <typename T, typename U>
void eliminar(std::list<U> &elems, T func) {
    ...
    if (func(*it)) { ... }
    ...
}
```

# Ejemplo

```
class EsPar {  
public:  
    bool operator()(int x) { return x % 2 == 0; }  
};  
  
int main() {  
    ...  
    EsPar obj_fun;  
    eliminar(v1, obj_fun);  
    ...  
}
```



# ¿Para qué sirven los objetos función?

# ¿Cuál es la diferencia?

Entre esto...

```
class EsPar {  
public:  
    bool operator()(int x) { return x % 2 == 0; }  
};
```

...y esto...

```
bool es_par(int x) { return x % 2 == 0; }
```

# Ejemplo: criba de Eratóstenes

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30

# Ejemplo: criba de Eratóstenes

- Supongamos que tenemos una lista con los números 2, 3, 4, 5, ..., 100.
  - Eliminamos los múltiplos de 2.
  - Eliminamos los múltiplos de 3.
  - Eliminamos los múltiplos de 5.
  - etc.



# Ejemplo: criba de Eratóstenes

```
bool es_multiplo_de_dos(int x) {
    return x % 2 == 0;
}

bool es_multiplo_de_tres(int x) {
    return x % 3 == 0;
}

bool es_multiplo_de_cinco(int x) { ... }

int main() {
    std::list<int> lista;
    for (int i = 2; i <= 100; i++) { lista.push_back(i); }

    std::list<int> primos;
    primos.push_back(lista.front());
    eliminar(lista, es_multiplo_de_dos);
    primos.push_back(lista.front());
    eliminar(lista, es_multiplo_de_tres);
    primos.push_back(lista.front());
    eliminar(lista, es_multiplo_de_cinco);
    ...
}
```

# Ejemplo: criba de Eratóstenes

```
bool es_multiplo_de_y(int x, int y) {
    return x % y == 0;
}

int main() {
    std::list<int> lista;
    for (int i = 2; i <= 100; i++) { lista.push_back(i); }

    std::list<int> primos;
    primos.push_back(lista.front());
    eliminar(lista, es_multiplo_de_y);
    primos.push_back(lista.front());
    eliminar(lista, es_multiplo_de_y);
    primos.push_back(lista.front());
    eliminar(lista, es_multiplo_de_y);
    ...
}
```



# Ejemplo: criba de Eratóstenes

```
class EsMultiploDeY {
private:
    int y;
public:
    EsMultiploDeY(int y): y(y) { }
    bool operator()(int x) { return x % y == 0; }
};

int main() {
    std::list<int> lista;
    for (int i = 2; i <= 100; i++) { lista.push_back(i); }

    EsMultiploDeY mult_dos(2), mult_tres(3), mult_cinco(5);
    std::list<int> primos;
    primos.push_back(lista.front());
    eliminar(lista, mult_dos);
    primos.push_back(lista.front());
    eliminar(lista, mult_tres);
    primos.push_back(lista.front());
    eliminar(lista, mult_cinco);
    ...
}
```

# Ejemplo: criba de Eratóstenes

```
class EsMultiploDeY {
private:
    int y;
public:
    EsMultiploDeY(int y): y(y) { }
    bool operator()(int x) { return x % y == 0; }
};

int main() {
    std::list<int> lista;
    for (int i = 2; i <= 100; i++) { lista.push_back(i); }

    std::list<int> primos;
    while (!lista.empty()) {
        primos.push_back(lista.front());
        EsMultiploDeY multiplos_de_front(lista.front());
        eliminar(lista, multiplos_de_front);
    }
}
```

# ¿Para qué sirve un objeto función?

- Cuando queremos pasar una función como parámetro, pero esa función, además de sus argumentos, depende de otros valores.

```
class EsMultiploDeY {  
private:  
    int y;  
public:  
    EsMultiploDeY(int y): y(y) { }  
    bool operator()(int x) { return x % y == 0; }  
};
```



ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Expresiones lambda (C++11)

Manuel Montenegro Montes

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

# Recordatorio

- Función que recibe una lista, una función booleana y elimina aquellos elementos para los que la función devuelve `true`.

```
template <typename T, typename U>
void eliminar(std::list<U> &elems, T func) {
    auto it = elems.begin();
    while (it != elems.end()) {
        if (func(*it)) {
            it = elems.erase(it);
        } else {
            ++it;
        }
    }
}
```

# Recordatorio

Hasta ahora hemos pasado como argumento func:

- **Funciones:**

```
bool es_par(int x) { return x % 2 == 0; }  
...  
eliminar(v1, es_par);
```

- **Objetos función:**

```
class EsMultiploDeY { ... }  
...  
EsMultiploDeY multiplo_de_dos(2);  
eliminar(v1, multiplo_de_dos);
```

- En cualquier caso, tenemos que definir una función o una clase aparte.
  - y es posible que solamente se utilice una vez.

# Expresiones lambda

- Nos permiten declarar un objeto función en el sitio en el que se utiliza, con una sintaxis más breve.
- Sintaxis:

```
[capturas] (parámetros) { cuerpo }
```

# Ejemplo

- En lugar de

```
bool es_par(int x) { return x % 2 == 0; }
...
eliminar(v1, es_par);
```

- Puede escribirse

```
eliminar(v1, [](int x) { return x % 2 == 0; });
```

# Más ejemplos

```
std::list<int> v1 = {1, 5, 6, 9, 10, 20};
```

```
std::list<int> v2 = v1;
```

```
eliminar(v1, [](int x) { return x % 2 == 0; });
```

```
std::cout << v1 << std::endl;
```

```
eliminar(v2, [](int x) { return x % 2 == 1; });
```

```
std::cout << v2 << std::endl;
```

```
std::list<int> v3 = {-2, 3, 10, -6, 20};
```

```
eliminar(v3, [](int x) { return x > 0; });
```

```
std::cout << v3 << std::endl;
```

```
std::list<Fecha> v4 = { {25, 12, 2010}, {10, 21, 2020}, {25, 12, 1900}, {1, 1, 2000} };
```

```
eliminar(v4, [](const Fecha &f) { return f.get_dia() == 25 && f.get_mes() == 12; });
```

```
std::cout << v4 << std::endl;
```

# Capturas

- Las expresiones lambda pueden tener, en su cuerpo, referencias a variables *externas* (esto es, variables distintas a los parámetros).

```
int y = 3;  
eliminar(v, [](int x) { return x % y = 0; });
```

- Cuando esto ocurre, decimos que la variable y está **capturada** por la expresión lambda.
- C++ nos obliga a declarar las variables capturadas dentro de [ ].

```
int y = 3;  
eliminar(v, [y](int x) { return x % y = 0; });
```

# Capturas

Hay dos maneras de capturar variables:

- **Por valor**

```
[y](int x) { /* ... */ }
```

Dentro de la lambda expresión no se pueden realizar cambios sobre la variable y.

- **Por referencia**

```
[&y](int x) { /* ... */ }
```

La lambda expresión trabaja con una **referencia** a la variable y.

Cualquier cambio que se haga sobre la variable y dentro de la lambda expresión afectará a la variable y externa.

# Ejemplo

```
int y = 3;  
auto f = [&y]() { y++; };  
f();  
std::cout << y << std::endl;
```



# Criba de eratóstenes: el retorno

```
std::list<int> lista;
std::list<int> primos;
...
while (!lista.empty()) {
    int primero = lista.front();
    primos.push_back(primer);
    eliminar(lista, [primer](int x) { return x % primero == 0; });
}
std::cout << primos << std::endl;
```

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# STL: Algoritmos (2)

Manuel Montenegro Montes

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

# Funciones de orden superior

# La función transform

`transform(ini, fin, dest, fun)`

- Definida en `<algorithm>`
- Aplica la función `fun` al conjunto de elementos contenido entre los iteradores `[ini, fin]`.
- Los resultados devueltos por `fun` son copiados a partir del iterador `dest`.
- Si se desea modificar la lista original, utilizar `dest = ini`.

# Ejemplos

```
vector<int> v = { 3, 10, 9, 3, 15 };
transform(v.begin(), v.end(), v.begin(), [](int x) { return x * 2; });
```

v = [6, 20, 18, 6, 30]

```
vector<string> nombres = {"Juan", "Rosario", "Amalia"};
vector<int> longitudes;
```

```
transform(nombres.begin(), nombres.end(),
         back_insert_iterator<vector<int>>(longitudes),
         [](const string &x) { return x.length(); });
```

longitudes = [4, 7, 6]

# La función `remove_if`

`remove_if(ini, fin, fun)`

- Definida en `<algorithm>`
- Elimina del rango de elementos `[ini, fin]` aquellos para los que `fun` devuelve `true`.
- Devuelve un iterador tras el último elemento de la colección resultante.

# Ejemplo

```
vector<int> v2 = { 3, 10, 8, 7, 4 };
auto it_end = remove_if(v2.begin(), v2.end(), [](int x) { return x % 2 == 0; });

copy(v2.begin(), it_end, ostream_iterator<int>(cout, " "));
```

3 7

# Las funciones `find_if` y `count_if`

`find_if(ini, fin, fun)`

- Devuelve un iterador al primer elemento del rango `[ini, fin]` para el que `fun` devuelve `true`.

`count_if(ini, fin, fun)`

- Devuelve el número de elementos del rango `[ini, fin]` para los que `fun` devuelve `true`.

# Ejemplo

```
vector<Fecha> fechas = {{10, 3, 2010}, {1, 6, 2019}, {28, 8, 1985}, {19, 3, 2001}};  
  
auto it_marzo = find_if(fechas.begin(), fechas.end(),  
                       [](const Fecha &f) { return f.get_mes() = 3; });  
  
cout << *it_marzo << endl;    10/03/2010  
  
int num_fechas_verano =  
    count_if(fechas.begin(), fechas.end(),  
             [](const Fecha &f) { return f.get_mes() ≥ 6 && f.get_mes() ≤ 8; });  
  
cout << num_fechas_verano << endl; 2
```

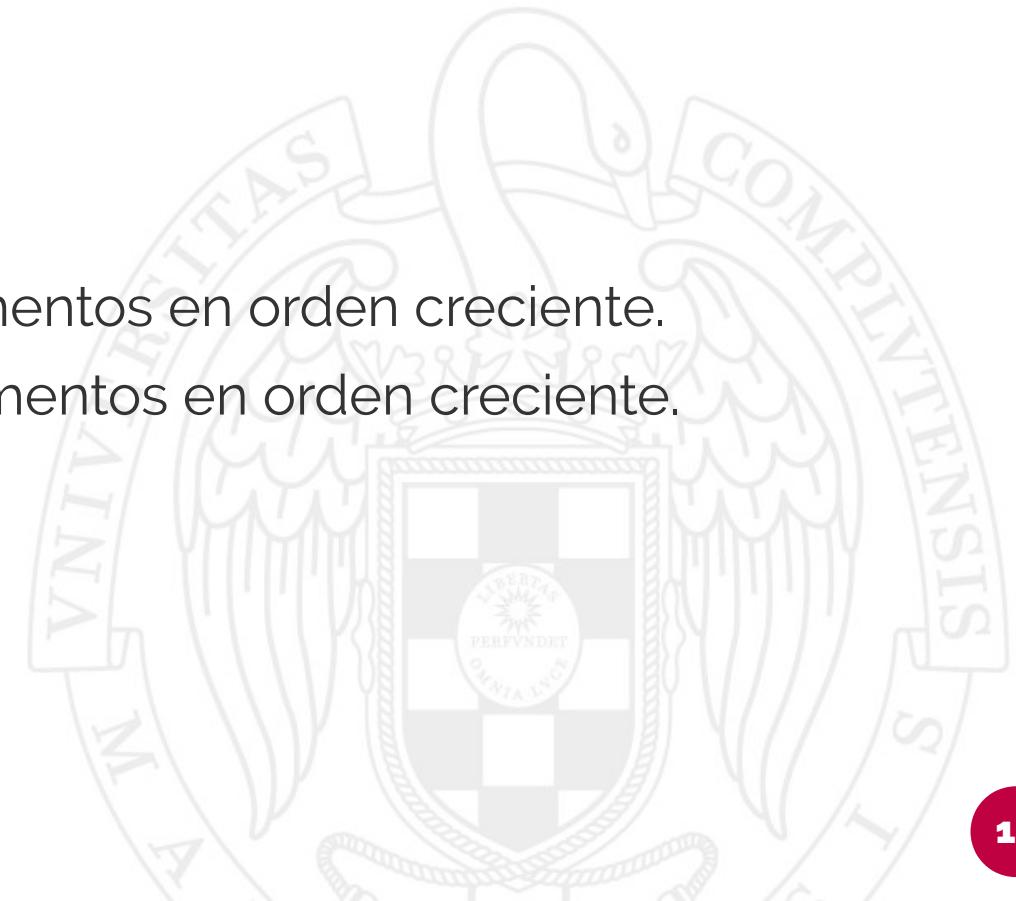
# Otras funciones de orden superior

- `all_of(ini, fin, fun)`
  - `any_of(ini, fin, fun)`
  - `none_of(ini, fin, fun)`
- 
- `accumulate(ini, fin, valor_inicial)`
  - `accumulate(ini, fin, valor_inicial, fun)`

# Funciones sobre conjuntos

# Funciones sobre conjuntos

- Las siguientes funciones pueden aplicarse sobre colecciones tales que, al ser iteradas, produzcan secuencias de elementos en orden ascendente. Esto incluye:
  - set (pero no unordered\_set)
  - map (pero no unordered\_map)
  - Listas que almacenen sus elementos en orden creciente.
  - Arrays que almacenen sus elementos en orden creciente.



# Funciones sobre conjuntos

- `includes(ini1, fin1, ini2, fin2)`
- `set_union(ini1, fin1, ini2, fin2, dest)`
- `set_intersection(ini1, fin1, ini2, fin2, dest)`
- `set_difference(ini1, fin1, ini2, fin2, dest)`

# Ejemplos

```
set<int> elems1 = { 6, 1, 9, 4, 3, 10 };
set<int> elems2 = { 10, 1, 4, 6 };

cout << includes(elems1.begin(), elems1.end(), elems2.begin(), elems2.end()) << endl; true
```

```
set<string> chicos = {"Ricardo", "Jaime", "Rafa", "Enrique", "Adrián", "Jose"};
set<string> chicas = {"Clara", "Susana", "Jose", "Natalia", "Elvira"};
list<string> result;
```

```
set_union(chicos.begin(), chicos.end(),
          chicas.begin(), chicas.end(),
          back_insert_iterator<list<string>>(result));
```

```
result = [Adrián, Clara, Elvira, Enrique, Jaime, Jose, Natalia, Rafa, Ricardo, Susana]
```

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# STL: Algoritmos (2)

Manuel Montenegro Montes

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

# Funciones de orden superior

# La función transform

`transform(ini, fin, dest, fun)`

- Definida en `<algorithm>`
- Aplica la función `fun` al conjunto de elementos contenido entre los iteradores `[ini, fin]`.
- Los resultados devueltos por `fun` son copiados a partir del iterador `dest`.
- Si se desea modificar la lista original, utilizar `dest = ini`.

# Ejemplos

```
vector<int> v = { 3, 10, 9, 3, 15 };
transform(v.begin(), v.end(), v.begin(), [](int x) { return x * 2; });
```

v = [6, 20, 18, 6, 30]

```
vector<string> nombres = {"Juan", "Rosario", "Amalia"};
vector<int> longitudes;
```

```
transform(nombres.begin(), nombres.end(),
         back_insert_iterator<vector<int>>(longitudes),
         [](const string &x) { return x.length(); });
```

longitudes = [4, 7, 6]

# La función `remove_if`

`remove_if(ini, fin, fun)`

- Definida en `<algorithm>`
- Elimina del rango de elementos `[ini, fin]` aquellos para los que `fun` devuelve `true`.
- Devuelve un iterador tras el último elemento de la colección resultante.

# Ejemplo

```
vector<int> v2 = { 3, 10, 8, 7, 4 };
auto it_end = remove_if(v2.begin(), v2.end(), [](int x) { return x % 2 == 0; });

copy(v2.begin(), it_end, ostream_iterator<int>(cout, " "));
```

3 7

# Las funciones `find_if` y `count_if`

`find_if(ini, fin, fun)`

- Devuelve un iterador al primer elemento del rango `[ini, fin]` para el que `fun` devuelve `true`.

`count_if(ini, fin, fun)`

- Devuelve el número de elementos del rango `[ini, fin]` para los que `fun` devuelve `true`.

# Ejemplo

```
vector<Fecha> fechas = {{10, 3, 2010}, {1, 6, 2019}, {28, 8, 1985}, {19, 3, 2001}};  
  
auto it_marzo = find_if(fechas.begin(), fechas.end(),  
                       [](const Fecha &f) { return f.get_mes() = 3; });  
  
cout << *it_marzo << endl;    10/03/2010  
  
int num_fechas_verano =  
    count_if(fechas.begin(), fechas.end(),  
             [](const Fecha &f) { return f.get_mes() ≥ 6 && f.get_mes() ≤ 8; });  
  
cout << num_fechas_verano << endl; 2
```

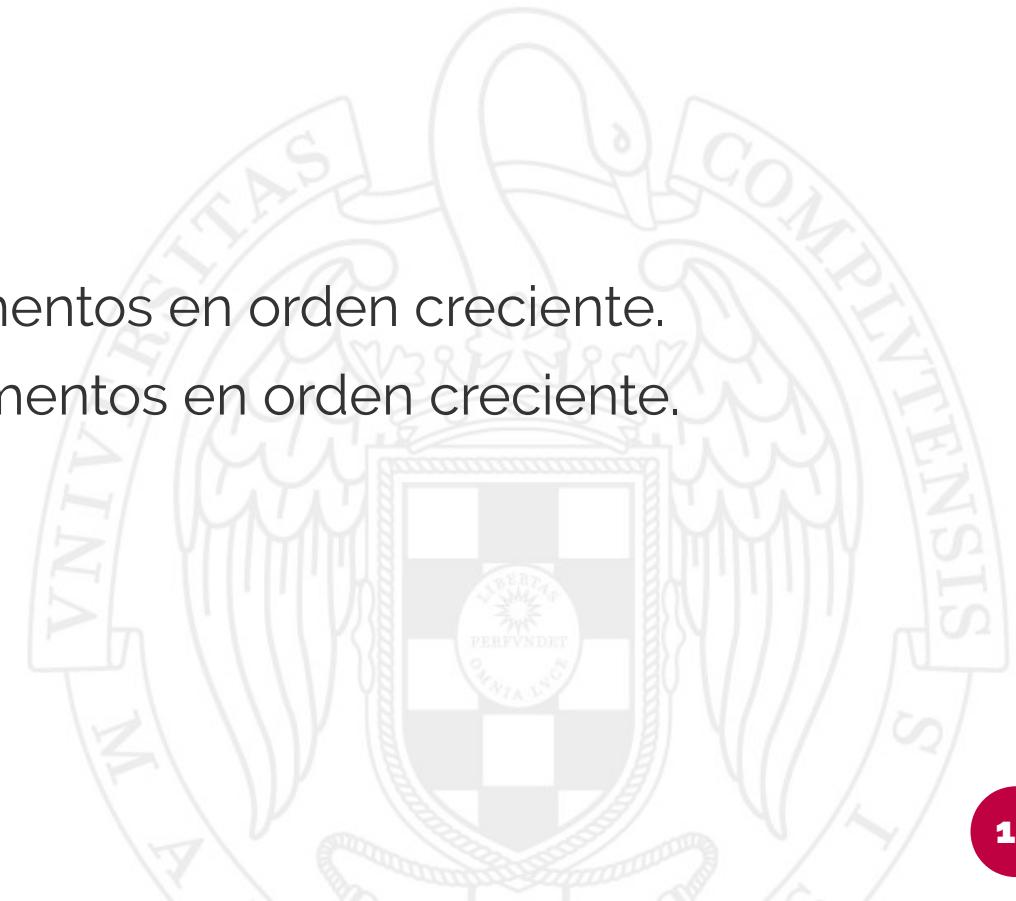
# Otras funciones de orden superior

- `all_of(ini, fin, fun)`
  - `any_of(ini, fin, fun)`
  - `none_of(ini, fin, fun)`
- 
- `accumulate(ini, fin, valor_inicial)`
  - `accumulate(ini, fin, valor_inicial, fun)`

# Funciones sobre conjuntos

# Funciones sobre conjuntos

- Las siguientes funciones pueden aplicarse sobre colecciones tales que, al ser iteradas, produzcan secuencias de elementos en orden ascendente. Esto incluye:
  - `set` (pero no `unordered_set`)
  - `map` (pero no `unordered_map`)
  - Listas que almacenen sus elementos en orden creciente.
  - Arrays que almacenen sus elementos en orden creciente.



# Funciones sobre conjuntos

- `includes(ini1, fin1, ini2, fin2)`
- `set_union(ini1, fin1, ini2, fin2, dest)`
- `set_intersection(ini1, fin1, ini2, fin2, dest)`
- `set_difference(ini1, fin1, ini2, fin2, dest)`

# Ejemplos

```
set<int> elems1 = { 6, 1, 9, 4, 3, 10 };
set<int> elems2 = { 10, 1, 4, 6 };

cout << includes(elems1.begin(), elems1.end(), elems2.begin(), elems2.end()) << endl; true
```

```
set<string> chicos = {"Ricardo", "Jaime", "Rafa", "Enrique", "Adrián", "Jose"};
set<string> chicas = {"Clara", "Susana", "Jose", "Natalia", "Elvira"};
list<string> result;
```

```
set_union(chicos.begin(), chicos.end(),
          chicas.begin(), chicas.end(),
          back_insert_iterator<list<string>>(result));
```

```
result = [Adrián, Clara, Elvira, Enrique, Jaime, Jose, Natalia, Rafa, Ricardo, Susana]
```

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Manejo de excepciones

Manuel Montenegro Montes

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

# Lanzar y capturar excepciones

# Lanzamiento de excepciones

- Se utiliza la palabra clave **throw**.
- Recibe como argumento la excepción a lanzar.
  - Puede ser un objeto (*recomendado*) o un valor básico.
- No es necesario declarar los tipos de excepciones lanzadas.

```
class division_por_cero { };

double dividir(double x, double y) {
    if (y == 0) {
        throw division_por_cero();
    } else {
        return x / y;
    }
}
```

# Captura de excepciones

- Se utilizan bloques **try/catch**, con sintaxis similar a la de Java.
- Se permiten varios bloques catch, cada uno capturando un tipo distinto.
- No existe bloque **finally**.

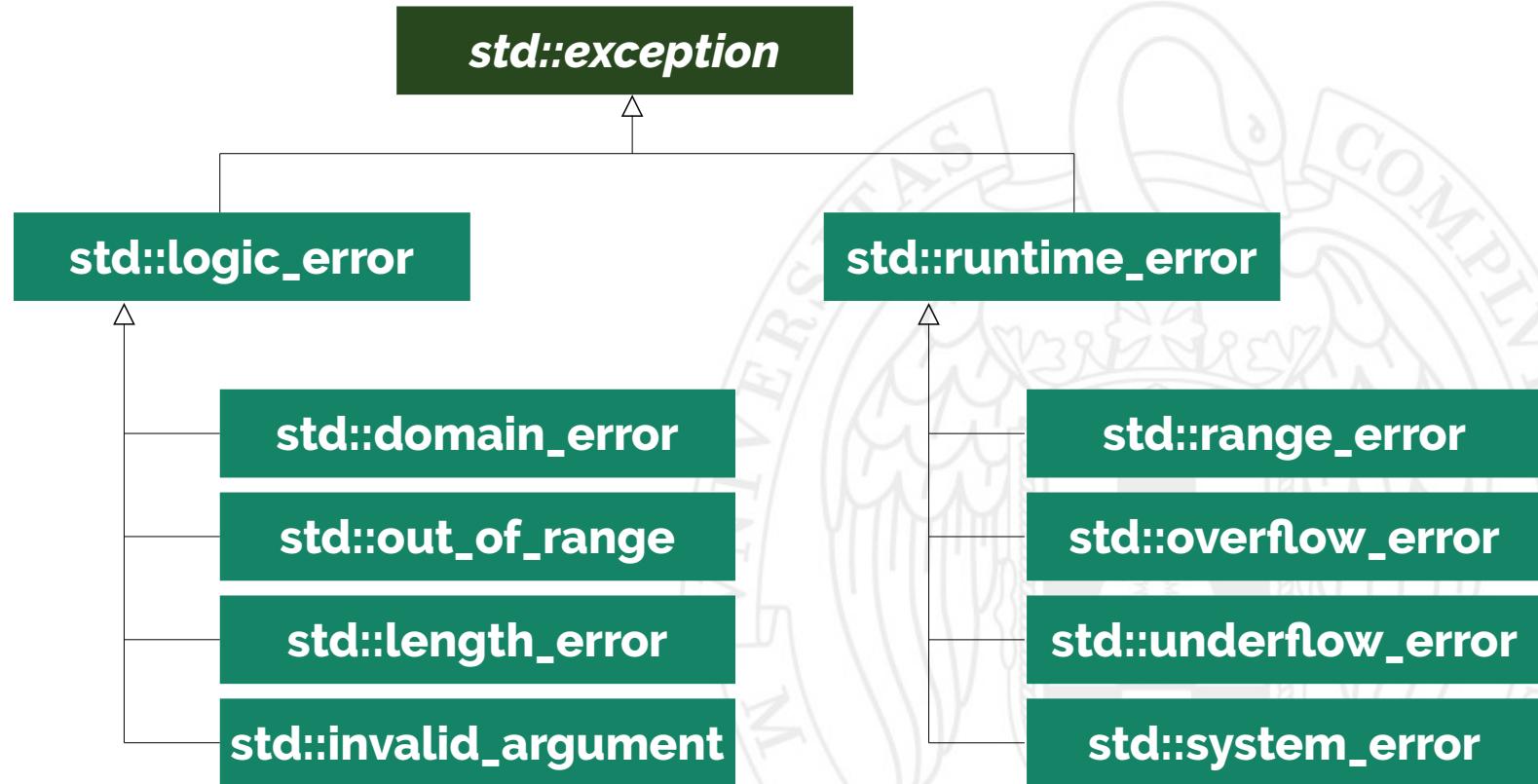
```
try {  
    dividir(1, 0);  
} catch (division_por_cero &e) {  
    std::cout << "División por cero!" << std::endl;  
}
```

La excepciones  
se capturan por  
referencia

# Jerarquía de excepciones estándar

# Excepciones estándar

- Ficheros de cabecera <exception> y <stdexcept>.



# Excepciones estándar

- Ficheros de cabecera <exception> y <stdexcept>.

***std::exception***

const char \*what()

Descripción de la excepción

# Ejemplo

```
double dividir(double x, double y) {
    if (y == 0) {
        throw std::domain_error("división por cero");
    } else {
        return x / y;
    }
}

int main() {
    try {
        dividir(1, 0);
    } catch (std::exception &e) {
        std::cout << e.what() << std::endl;
    }
}
```



# Heredar de excepciones estándar

```
class division_por_cero: public std::logic_error {  
public:  
    division_por_cero(): std::logic_error("división por cero") {}  
};  
  
double dividir(double x, double y) {  
    if (y == 0) {  
        throw division_por_cero();  
    } else {  
        return x / y;  
    }  
}  
  
int main() {  
    try {  
        dividir(1, 0);  
    } catch (division_por_cero &e) {  
        std::cout << e.what() << std::endl;  
    }  
}
```

**std::logic\_error**

**division\_por\_cero**

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Herencia y polimorfismo

Manuel Montenegro Montes

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

# Herencia



# Heredar de una clase

```
class Rectangulo {  
public:  
    Rectangulo(double ancho, double alto): ancho(ancho), alto(alto) {}  
  
    double area() { return ancho * alto; }  
    double perimetro() { return 2 * ancho + 2 * alto; }  
  
protected:  
    double ancho, alto;  
};  
  
class Cuadrado: public Rectangulo {  
public:  
    Cuadrado(double lado): Rectangulo(lado, lado) {}  
};
```

# Ejemplo

```
Rectangulo *r;  
double ancho, alto;  
cin >> ancho >> alto;  
  
if (ancho == alto) {  
    r = new Cuadrado(ancho);  
} else {  
    r = new Rectangulo(ancho, alto);  
}  
  
double area = r->area();  
double perimetro = r->perimetro();  
cout << "Area: " << area << endl;  
cout << "Perímetro: " << perimetro << endl;  
  
delete r;
```



# Polimorfismo y métodos virtuales

# Nuevo método: dibujar()

```
class Rectangulo {  
public:  
    ...  
    void dibujar() {  
        std::cout << "Rectángulo de ancho " << ancho << " y alto " << alto << std::endl;  
    }  
protected:  
    double ancho, alto;  
};  
  
class Cuadrado: public Rectangulo {  
public:  
    Cuadrado(double lado): Rectangulo(lado, lado) { }  
  
    void dibujar() {  
        std::cout << "Cuadrado de lado " << ancho << std::endl;  
    }  
};
```

# Ejemplo

```
Rectangulo *r;  
double ancho, alto;  
cin >> ancho >> alto;  
  
if (ancho == alto) {  
    r = new Cuadrado(ancho);  
} else {  
    r = new Rectangulo(ancho, alto);  
}  
  
r->dibujar();  
  
delete r;
```

1.2 4.5

Rectángulo de ancho 1.2 y alto 4.5

1.2 1.2

Rectángulo de ancho 1.2 y alto 1.2



# Vinculación estática vs dinámica

- C++ determina a qué método llamar en base al tipo del objeto sobre el que se realiza la llamada.

```
Rectangulo *r;  
double ancho, alto;  
cin >> ancho >> alto;  
  
if (ancho == alto) {  
    r = new Cuadrado(ancho);  
} else {  
    r = new Rectangulo(ancho, alto);  
}  
  
r->dibujar();  
  
delete r;
```

r es de tipo puntero a Rectangulo  
Por tanto el compilador determina que  
r->dibujar() llama al método  
dibujar de Rectangulo.

# Vinculación estática vs dinámica

- Si se realiza **vinculación dinámica**, decimos al compilador que se compruebe, en tiempo de ejecución, la clase a la que pertenece el objeto, y se llame al método correspondiente a esa clase, independientemente del tipo.
- Por defecto, en C++ se utiliza vinculación estática.
- Por defecto, en Java se utiliza vinculación dinámica.

# Habilitar la vinculación dinámica

```
class Rectangulo {  
public:  
    ...  
    virtual void dibujar() {  
        std::cout << "Rectángulo de ancho " << ancho << " y alto " << alto << std::endl;  
    }  
protected:  
    double ancho, alto;  
};  
  
class Cuadrado: public Rectangulo {  
public:  
    Cuadrado(double lado): Rectangulo(lado, lado) {}  
  
    virtual void dibujar() {  
        std::cout << "Cuadrado de lado " << ancho << std::endl;  
    }  
};
```

# Ejemplo

```
Rectangulo *r;  
double ancho, alto;  
cin >> ancho >> alto;  
  
if (ancho == alto) {  
    r = new Cuadrado(ancho);  
} else {  
    r = new Rectangulo(ancho, alto);  
}  
  
r->dibujar();  
  
delete r;
```

1.2 4.5

Rectángulo de ancho 1.2 y alto 4.5

1.2 1.2

Cuadrado de lado 1.2



# Reglas generales

- Cualquier método que sea susceptible de ser reescrito debe declararse como `virtual`.
- Si una clase tiene un método `virtual`, es muy aconsejable declarar su destructor como `virtual`, aunque no haga nada.



# Métodos abstractos

```
class Figura {  
public:  
    virtual double area() = 0;  
    virtual double perimetro() = 0;  
    virtual void dibujar() = 0;  
  
    virtual ~Figura() {}  
  
};  
  
class Rectangulo: public Figura { ... }
```

- Los métodos abstractos han de ser virtuales.
- Si una clase tiene un método abstracto, la clase es abstracta.
  - No pueden crearse instancias de Figura.

# Otras diferencias con Java

- En C++ no existe la noción de interfaz (**interface**).
- Se permite herencia múltiple.



ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Atributos y Métodos

Manuel Montenegro Montes

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

# Definición de una clase



# Definición de una clase: atributos

```
class Fecha {  
    int dia;  
    int mes;  
    int anyo;  
};
```



# Definición de una clase: métodos

```
class Fecha {  
    int dia;  
    int mes;  
    int anyo;  
  
    int get_dia();  
    void set_dia(int dia);  
    int get_mes();  
    void set_mes(int mes);  
    int get_anyo();  
    void set_anyo(int anyo);  
};
```

- Aquí se están **declarando** los métodos, pero no aparecen sus **implementaciones**.
- Por defecto, todos los atributos y métodos son privados.

# Modificadores de acceso

```
class Fecha {  
    int dia;  
    int mes;  
    int anyo;  
  
public:  
    int get_dia();  
    void set_dia(int dia);  
    int get_mes();  
    void set_mes(int mes);  
    int get_anyo();  
    void set_anyo(int anyo);  
};
```

- Hay tres tipos de modificadores:
  - **public**:
  - **private**:
  - **protected**:
- Afectan a los métodos y atributos situados a continuación del modificador.

# Modificadores de acceso

```
class Fecha {  
public:  
    int get_dia();  
    void set_dia(int dia);  
    int get_mes();  
    void set_mes(int mes);  
    int get_anyo();  
    void set_anyo(int anyo);  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```

- Hay tres tipos de modificadores:
  - public:
  - private:
  - protected:
- Afectan a los métodos y atributos situados a continuación del modificador.

# Implementación de métodos

```
class Fecha {  
public:  
    int get_dia() {  
        return dia;  
    }  
  
    void set_dia(int dia) {  
        this→dia = dia;  
    }  
  
    // Igualmente para mes y año  
    // ...  
  
private:  
    // ...  
};
```

- Posibilidad 1: Implementación dentro de la definición de clase.
- “Estilo Java”



# Implementación de métodos

```
class Fecha {  
public:  
    int get_dia();  
    void set_dia(int dia);  
    //  
    // ...  
private:  
    // ...  
};  
  
int Fecha::get_dia() {  
    return dia;  
}  
  
void Fecha::set_dia(int dia) {  
    this->dia = dia;  
}
```

- Posibilidad 2: Implementación fuera de la definición de clase.



# ¡No son equivalentes!

- Implementaciones dentro de la clase: se consideran métodos **inline**.

<https://www.geeksforgeeks.org/inline-functions-cpp/>

- Son más eficientes, pero incrementan el tamaño del código.

- **Consejo:**

- Métodos cortos (p.ej. acceso, modificación) pueden definirse dentro de la clase.
  - Métodos largos deben definirse fuera de la clase.

# **Uso de una clase: instancias**



# Creación de instancias

```
int main() {  
    Fecha f;  
    f.set_dia(28);  
    f.set_mes(8);  
    f.set_anyo(2019);  
  
    std::cout << "Día: " << f.get_dia() << std::endl;  
    std::cout << "Mes: " << f.get_mes() << std::endl;  
    std::cout << "Año: " << f.get_anyo() << std::endl;  
}
```

Día: 28  
Mes: 8  
Año: 2019

# **Salida con formato**



# Un nuevo método: imprimir

```
class Fecha {  
public:  
    int get_dia();  
    void set_dia(int dia);  
    int get_mes();  
    void set_mes(int mes);  
    int get_anyo();  
    void set_anyo(int anyo);  
  
    void imprimir();  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```



# Un nuevo método: imprimir

```
void Fecha::imprimir() {  
    std::cout << dia << "/" << mes << "/" << anyo;  
}
```

# Un nuevo método: imprimir

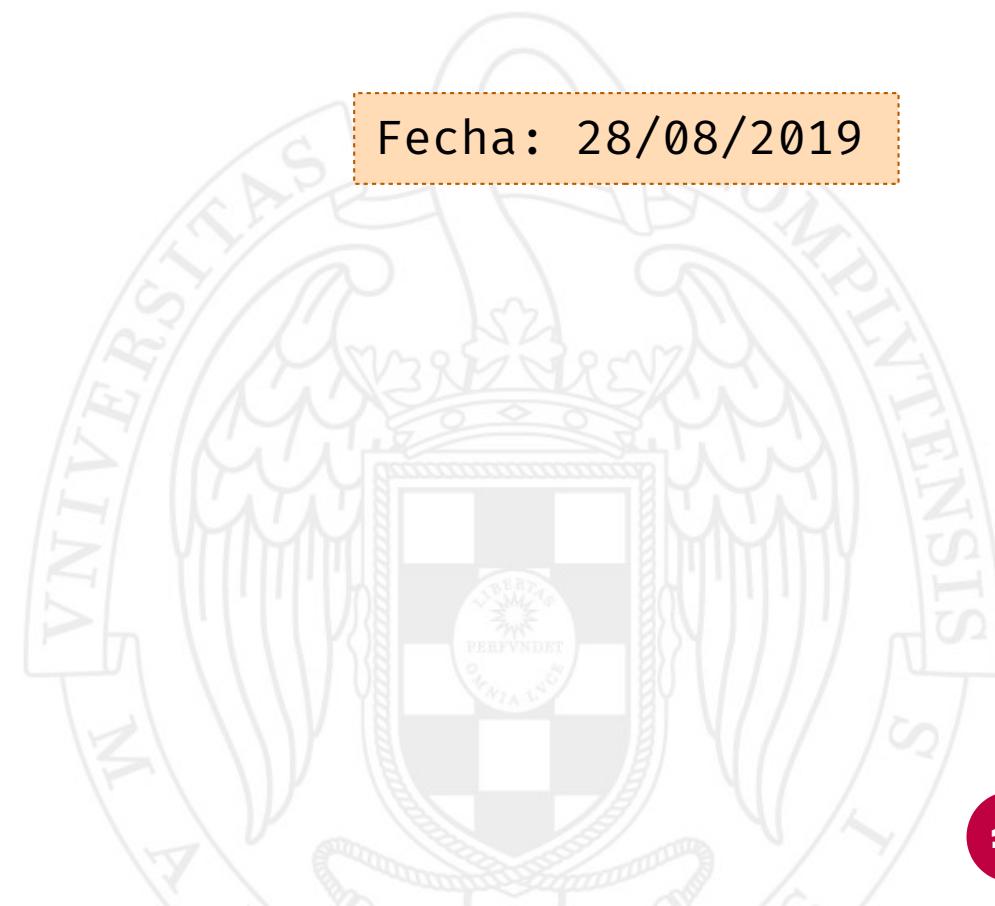
```
void Fecha::imprimir() {  
    std::cout << std::setfill('0') << std::setw(2) << dia << "/"  
        << std::setw(2) << mes << "/"  
        << std::setw(4) << anyo;  
}
```

```
#include <iostream>  
#include <iomanip>
```

# Uso del método imprimir

```
int main() {  
    Fecha f;  
    f.set_dia(28);  
    f.set_mes(8);  
    f.set_anyo(2019);  
  
    std::cout << "Fecha: ";  
    f.imprimir();  
    std::cout << std::endl;  
}
```

Fecha: 28/08/2019



ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Constructores Listas de Inicialización

Manuel Montenegro Montes

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

# Recordatorio: clase Fecha

```
class Fecha {  
public:  
    int get_dia();  
    void set_dia(int dia);  
    int get_mes();  
    void set_mes(int mes);  
    int get_anyo();  
    void set_anyo(int anyo);  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```



# Recordatorio: clase Fecha

```
int main() {  
    Fecha f;  
    f.set_dia(28);  
    f.set_mes(8);  
    f.set_anyo(2019);  
  
    std::cout << "Fecha: ";  
    f.imprimir();  
    std::cout << std::endl;  
}
```

- Hemos inicializado los atributos del objeto tras su creación, mediante los métodos set.
- ¿Y si se me hubiera olvidado llamar a estos métodos?
- **¿Existe alguna manera de asegurarnos de que el objeto está inicializado tras su creación?**
- Sí: **constructores**

# Tipos de constructores

- **Constructor por defecto** (sin parámetros).
- **Constructor paramétrico.**
- **Constructor de copia.**
- **Constructor *move*.**
- **Constructor de conversión.**



# Constructor por defecto



# Constructor por defecto

```
class Fecha {  
public:  
    Fecha() {  
        dia = 1;  
        mes = 1;  
        anyo = 1900;  
    }  
  
    // ...  
private:  
    int dia;  
    int mes;  
    int anyo;  
}
```

- Todos los constructores tienen el mismo nombre que la clase.
- No tienen tipo de retorno.
- El **constructor por defecto** no tiene parámetros.

# Constructor por defecto

```
class Fecha {  
public:  
    Fecha();  
    // ...  
private:  
    int dia;  
    int mes;  
    int anyo;  
}  
  
Fecha::Fecha() {  
    dia = 1;  
    mes = 1;  
    anyo = 1900;  
}
```

- Otra posibilidad: definir la implementación fuera de la clase.



# Uso del constructor por defecto

```
int main() {  
    Fecha f;  
    f.imprimir();  
}
```

01/01/1900



# Constructor con parámetros

# Constructor con parámetros

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo) {  
        this->dia = dia;  
        this->mes = mes;  
        this->anyo = anyo;  
    }  
  
    // ...  
private:  
    int dia;  
    int mes;  
    int anyo;  
}
```



# Sobrecarga de constructores

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo) {  
        this->dia = dia;  
        this->mes = mes;  
        this->anyo = anyo;  
    }  
  
    Fecha(int anyo) {  
        this->dia = 1;  
        this->mes = 1;  
        this->anyo = anyo;  
    }  
    // ...  
private:  
    int dia;  
    int mes;  
    int anyo;  
}
```



# Delegación de constructores

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo) {  
        this->dia = dia;  
        this->mes = mes;  
        this->anyo = anyo;  
    }  
  
    Fecha(int anyo): Fecha(1, 1, anyo) {  
        // vacío  
    }  
  
    // ...  
private:  
    int dia;  
    int mes;  
    int anyo;  
}
```



# Uso del constructor con parámetros

```
int main() {  
    Fecha f;  
    f.imprimir();  
  
    return 0;  
}
```

Error: no hay constructor por defecto



# Uso del constructor con parámetros

```
int main() {  
    Fecha f1(28, 8, 2019);  
    Fecha f2(2019);  
  
    f1.imprimir();  
    std::cout << " ";  
    f2.imprimir();  
  
    return 0;  
}
```

28/08/2019 01/01/2019



# Uso del constructor con parámetros

```
int main() {  
    Fecha f1 = {28, 8, 2019};  
    Fecha f2 = {2019};  
  
    f1.imprimir();  
    std::cout << " "  
    f2.imprimir();  
  
    return 0;  
}
```

Sintaxis alternativa

# Paso de objetos a funciones

```
bool es_navidad(Fecha f) {
    return f.get_dia() == 25 && f.get_mes() == 12;
}

int main() {
    Fecha f = {25, 12, 2019};
    if (es_navidad(f)) {
        std::cout << "Feliz navidad!" << std::endl;
    }
    return 0;
}
```



# Paso de objetos a funciones

```
bool es_navidad(Fecha f) {  
    return f.get_dia() = 25 && f.get_mes() = 12;  
}  
  
int main() {  
    if (es_navidad({25, 12, 2019})) {  
        std::cout << "Feliz navidad!" << std::endl;  
    }  
    return 0;  
}
```

Creación de objeto en el argumento

# Listas de inicialización

# Una nueva clase: Persona

```
class Persona {  
  
private:  
    std::string nombre;  
    Fecha fecha_nacimiento;  
};
```

```
int main() {  
    Persona p;  
    ...  
}
```

El constructor por defecto no  
puede inicializar fecha\_nacimiento

# Añadiendo un constructor a Persona

```
class Persona {  
public:  
    Persona(std::string nombre, int dia, int mes, int anyo) {  
        this->nombre = nombre;  
        ... ???  
    }  
  
private:  
    std::string nombre;  
    Fecha fecha_nacimiento;  
};
```

- ¿Cómo indico que quiero llamar al constructor de Fecha pasándole dia, mes y anyo?

# Llamando al constructor de Fecha

```
class Persona {  
public:  
    Persona(std::string nombre, int dia, int mes, int anyo)  
        : fecha_nacimiento(dia, mes, anyo) {  
            this->nombre = nombre;  
    }  
  
private:  
    std::string nombre;  
    Fecha fecha_nacimiento;  
};
```

- Al crear el objeto Persona, se llamará al constructor de Fecha con los tres argumentos indicados.

# Llamando al constructor de Fecha

```
class Persona {  
public:  
    Persona(std::string nombre, int dia, int mes, int anyo)  
        : nombre(nombre), fecha_nacimiento(dia, mes, anyo) {}  
  
private:  
    std::string nombre;  
    Fecha fecha_nacimiento;  
};
```

- Podemos utilizar la misma sintaxis con el resto de los atributos.
- A esto se le llama **lista de inicialización**.

# Listas de inicialización

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo) {  
        this->dia = dia;  
        this->mes = mes;  
        this->anyo = anyo;  
    }  
  
    Fecha(int anyo): Fecha(1, 1, anyo) {}  
  
    // ...  
private:  
    int dia;  
    int mes;  
    int anyo;  
}
```



# Listas de inicialización

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo): dia(dia), mes(mes), anyo(anyo) {}  
  
    Fecha(int anyo): Fecha(1, 1, anyo) {}  
  
    // ...  
private:  
    int dia;  
    int mes;  
    int anyo;  
}
```



ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Métodos constantes

Manuel Montenegro Montes

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

# Recordatorio: clase Fecha

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo);  
    Fecha(int anyo);  
  
    int get_dia();  
    void set_dia(int dia);  
    int get_mes();  
    void set_mes(int mes);  
    int get_anyo();  
    void set_anyo(int anyo);  
    void imprimir();  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```



# Paso de objetos por valor

```
bool es_navidad(Fecha f) {  
    return f.get_dia() == 25  
        && f.get_mes() == 12;  
}  
  
int main() {  
    Fecha mi_fecha(25, 12, 2000);  
    if (es_navidad(mi_fecha)) {  
        std::cout << "Feliz navidad!"  
            << std::endl;  
    }  
    return 0;  
}
```

- La función `es_navidad` recibe su argumento **por valor**.
- Al pasar por valor una instancia de una clase se hace una copia del argumento.
  - ¿Cómo? Constructor de copia.
- Si queremos evitar eso, debemos pasar el parámetro **por referencia**.

# Paso de objetos por referencia



# Paso de objetos por referencia

```
bool es_navidad(Fecha &f) {  
    return f.get_dia() == 25  
        && f.get_mes() == 12;  
}  
  
int main() {  
    Fecha mi_fecha(25, 12, 2000);  
    if (es_navidad(mi_fecha)) {  
        std::cout << "Feliz navidad!"  
            << std::endl;  
    }  
    return 0;  
}
```

- Mediante el símbolo & indicamos que el parámetro f se recibe por referencia.
- Con esto se evita hacer una copia de mi\_fecha.
- ¡Ojo! Cualquier cambio que es\_navidad realice en f se reflejará también en mi\_fecha.
  - En este caso, podemos ver que es\_navidad no está alterando el objeto f.

# ¿Y si no conocemos la implementación?

- ¿Cuál de estas dos funciones te inspira más confianza?

```
bool compara(Fecha f1, Fecha f2);
```

```
bool compara(Fecha &f1, Fecha &f2);
```

- La primera garantiza que no va a alterar el estado de los objetos Fecha que reciba, ya que va a trabajar sobre copias de los mismos.
- La segunda no ofrece esa garantía, aunque se ahorra la copia de los argumentos.
- **¿Podemos conseguir los beneficios de ambas versiones?**

# Referencias constantes

```
bool compara(const Fecha &f1, const Fecha &f2);
```

- Una **referencia constante** no permite modificar el estado del objeto apuntado por la referencia.
- El compilador comprueba que `compara` no modifique los atributos de los objetos `f1` y `f2`.
- Con esto:
  - Nos ahorramos copias de los argumentos, porque se pasan por referencia.
  - El que llame a la función `compara` tiene la certeza de que sus objetos no se van a ver modificados.

# Paso de objetos por valor

```
bool es_navidad(const Fecha &f) {  
    return f.get_dia() == 25  
        && f.get_mes() == 12; X  
}
```

- Hacemos que la función `es_navidad` reciba su parámetro como referencia constante.
- ... pero el compilador protesta sobre nuestra definición.
- El compilador no sabe si los métodos `get_dia()` o `get_mes()` alteran el estado de `f`.

# Métodos constantes

# Métodos constantes

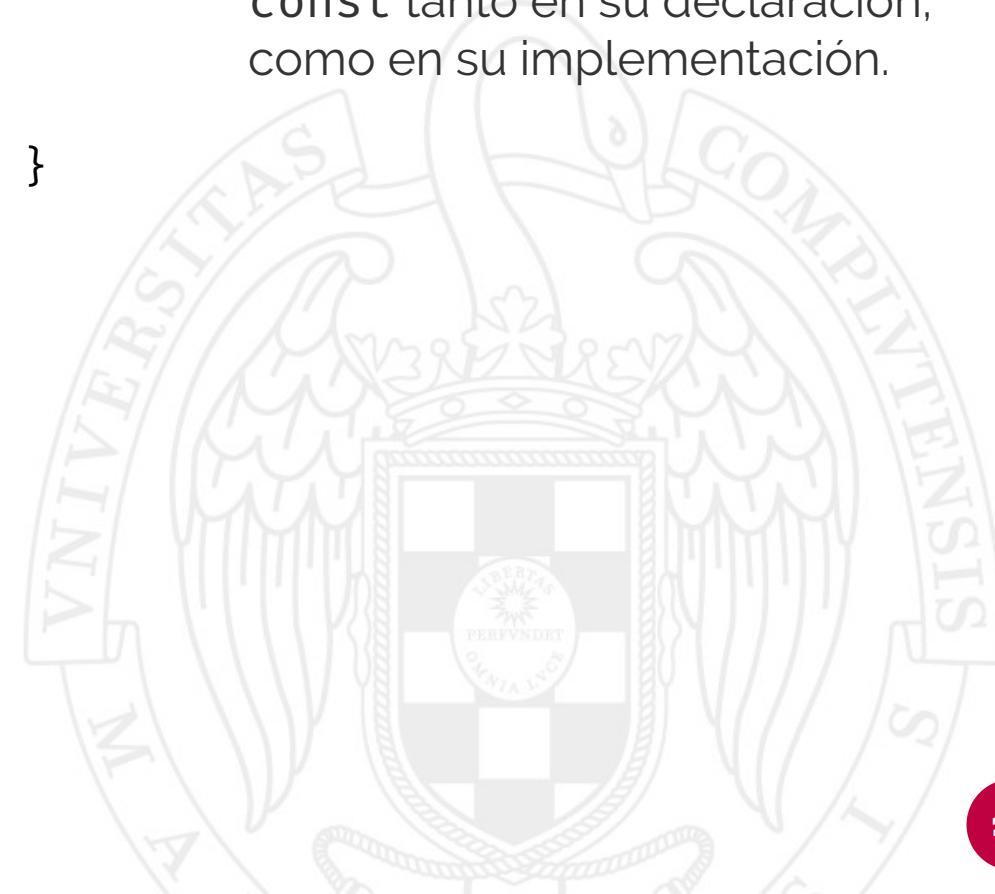
```
class Fecha {  
public:  
    ...  
    int get_dia() const { return dia; }  
    int get_mes() const { return mes; }  
    int get_anyo() const { return anyo; }  
    void imprimir() const;  
    ...  
  
private:  
    ...  
};
```

- Se declaran añadiendo la palabra **const** tras la lista de parámetros.
- Con esto se indica que el método no altera el estado del objeto.
- El compilador comprueba:
  - que el método no modifique los atributos del objeto.
  - que el método no llame a otros métodos de ese mismo objeto, salvo que también sean constantes.

# Métodos constantes

```
class Fecha {  
public:  
    ...  
    int get_dia() const { return dia; }  
    int get_mes() const { return mes; }  
    int get_anyo() const { return anyo; }  
    void imprimir() const;  
    ...  
  
private:  
    ...  
};  
  
void Fecha::imprimir() const {  
    ...  
}
```

- Si un método se implementa fuera de la clase, es necesario poner **const** tanto en su declaración, como en su implementación.



# Llamadas a métodos constantes

```
bool es_navidad(const Fecha &f) {  
    return f.get_dia() == 25  
        && f.get_mes() == 12; ✓  
}
```

- Si una referencia a un objeto es constante:
  - No podemos modificar sus atributos públicos a través de esa referencia.
  - Solamente podemos llamar a los métodos `const` de esa referencia.

# ¿Qué métodos deben ser const?

- Todos los que no modifiquen el estado del objeto que recibe la llamada al método (`this`).
- Este tipo de métodos reciben el nombre de **observadores**.
- Incluye, entre otros:
  - Métodos de acceso (`get`).
  - Métodos para imprimir el objeto por pantalla o a otro flujo de salida.
  - Métodos de conversión a otro objeto (por ejemplo, `to_string()`).

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Objetos y memoria dinámica

Manuel Montenegro Montes

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

# Regiones de memoria: pila y *heap*



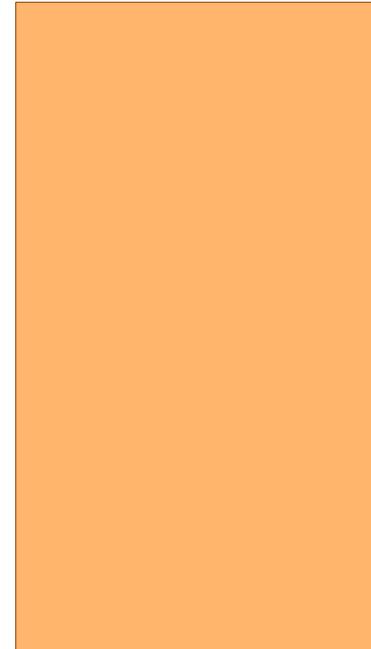
# Regiones de memoria

- **Memoria principal (global)**: variables globales.
  - Se reserva al iniciarse el programa, y se libera al finalizarse.
- **Pila**: variables locales, parámetros.
  - Se reserva y libera a medida que estas variables entran en ámbito y salen de ámbito, respectivamente.
- **Heap**: memoria dinámica.
  - Se reserva y libera manualmente mediante `new` y `delete`.
  - Solamente es accesible a través de punteros.

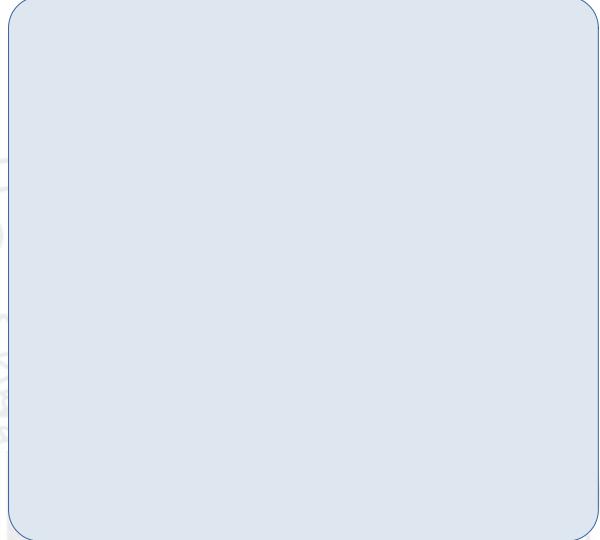
# Regiones de memoria

```
int main() {  
    int x = 3;  
    int *y = new int;  
    *y = 3;  
    int *z = &x;  
  
    delete y;  
    return 0;  
}
```

Pila



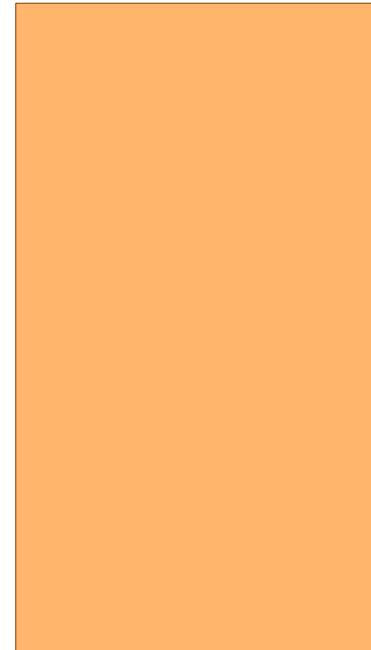
Heap



# Regiones de memoria

```
int main() {  
    int *xs = new int[4];  
    xs[0] = 3;  
    xs[1] = 7;  
  
    int ys[3];  
  
    delete[] xs;  
    return 0;  
}
```

Pila



Heap



# Creación de objetos en el *heap*

# Recordatorio: clase Fecha

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo);  
    Fecha(int anyo);  
  
    int get_dia() const;  
    void set_dia(int dia);  
    int get_mes() const;  
    void set_mes(int mes);  
    int get_anyo() const;  
    void set_anyo(int anyo);  
    void imprimir();  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```



# Creación de instancias en la pila y heap

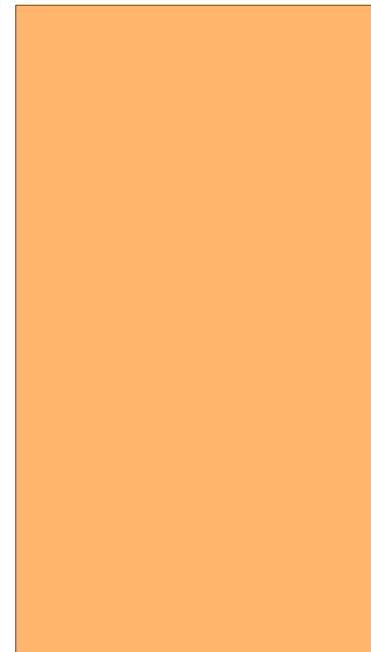
```
int main() {
    Fecha f1(28, 8, 2038);
    Fecha *f2 = new Fecha(10, 6, 2010);

    std::cout << "Fecha 1: ";
    f1.imprimir();
    std::cout << std::endl;

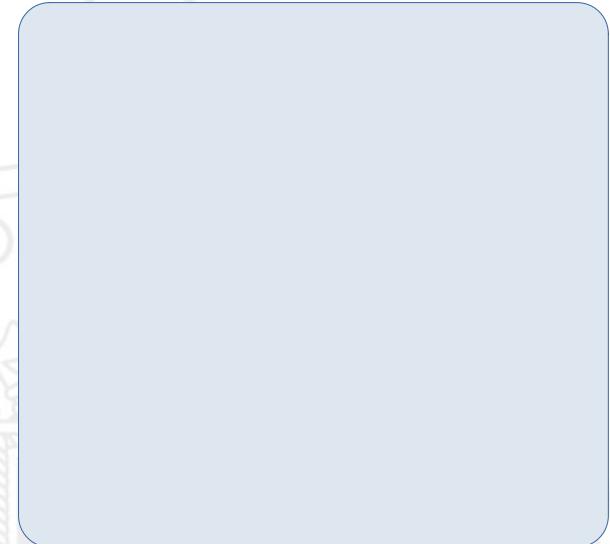
    std::cout << "Fecha 2: ";
    f2->imprimir();
    std::cout << std::endl;

    delete f2;
    return 0;
}
```

Pila



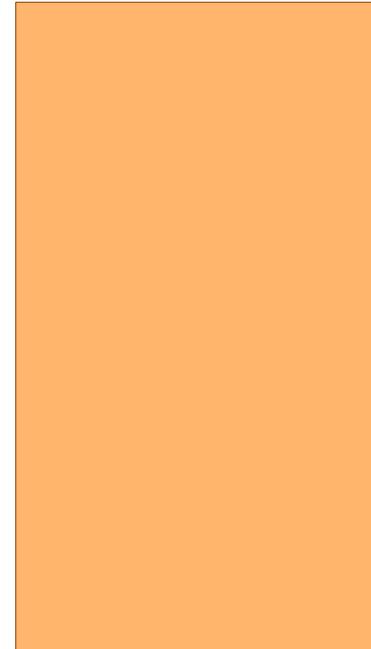
Heap



# Arrays de objetos

```
int main() {  
    Fecha fs[3] =  
        { {2010}, {2011}, {2012} };  
  
    return 0;  
}
```

Pila



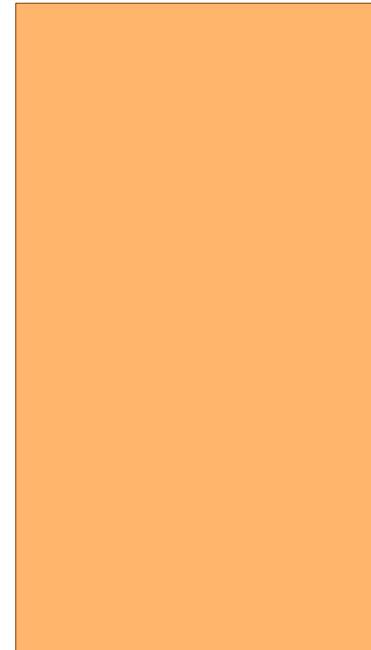
Heap



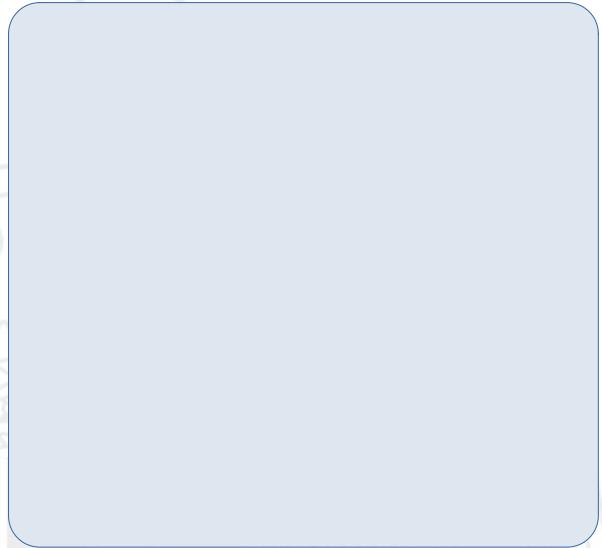
# Arrays de punteros a objetos

```
int main() {  
    Fecha *fs[3];  
    fs[0] = new Fecha(2010);  
    fs[1] = new Fecha(2011);  
    fs[2] = new Fecha(2012);  
  
    delete fs[0];  
    delete fs[1];  
    delete fs[2];  
  
    return 0;  
}
```

Pila



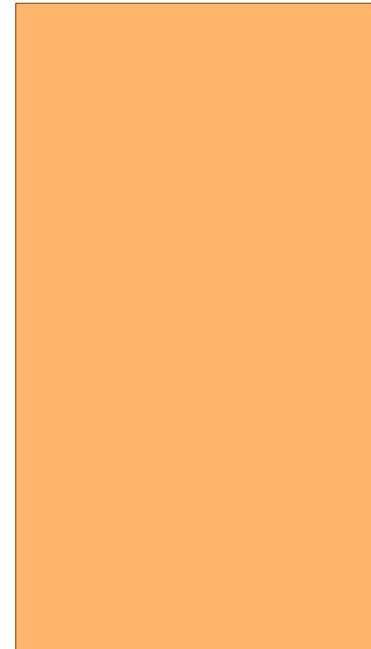
Heap



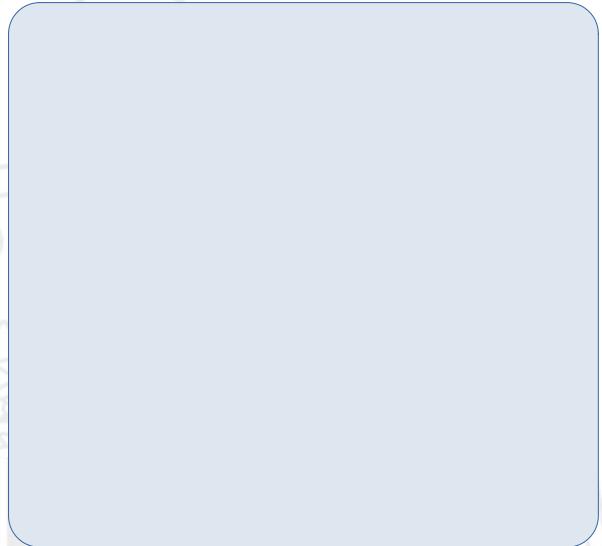
# Arrays dinámicos de punteros a objetos

```
int main() {  
    Fecha **fs = new Fecha*[3];  
    fs[0] = new Fecha(2010);  
    fs[1] = new Fecha(2011);  
    fs[2] = new Fecha(2012);  
  
    delete fs[0];  
    delete fs[1];  
    delete fs[2];  
    delete[] fs;  
  
    return 0;  
}
```

Pila



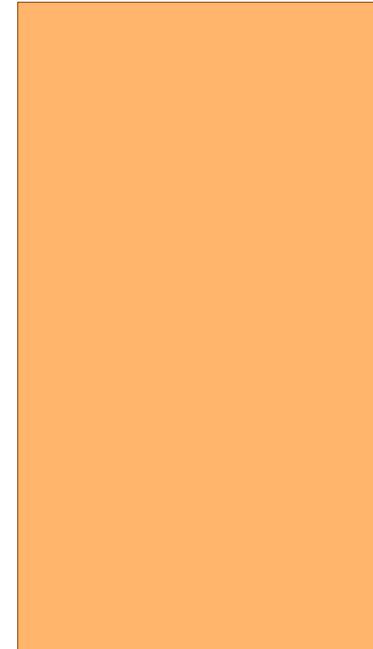
Heap



# Arrays dinámicos de objetos

```
int main() {  
    Fecha *fs = new Fecha[3];  
  
    delete[] fs;  
    return 0;  
}
```

Pila



Heap



# Añadiendo un constructor por defecto

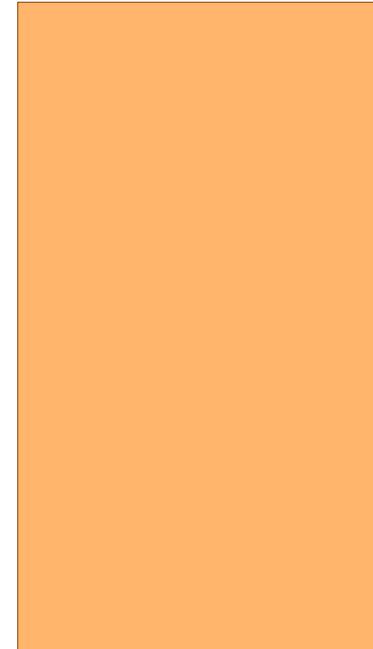
```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo);  
    Fecha(int anyo);  
    Fecha(): Fecha(1, 1, 1900) { }  
  
    int get_dia() const;  
    void set_dia(int dia);  
    int get_mes() const;  
    void set_mes(int mes);  
    int get_anyo() const;  
    void set_anyo(int anyo);  
    void imprimir();  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```



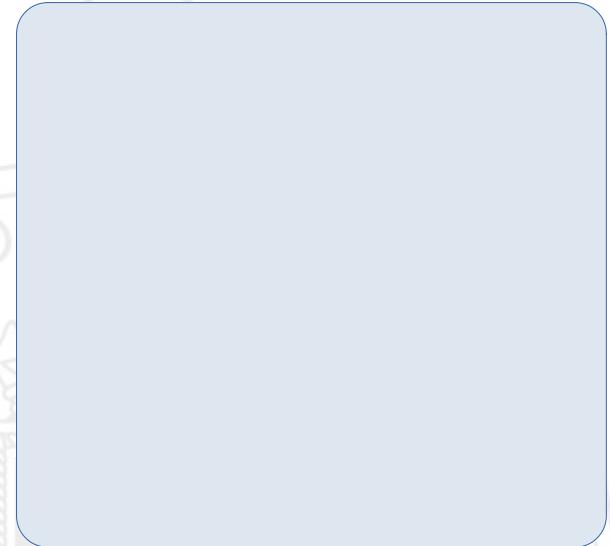
# Arrays dinámicos de objetos

```
int main() {  
    Fecha *fs = new Fecha[3];  
  
    delete[] fs;  
    return 0;  
}
```

Pila



Heap

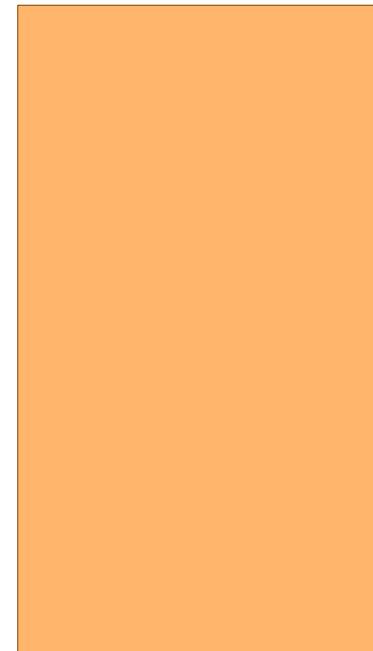


# Compartición de objetos

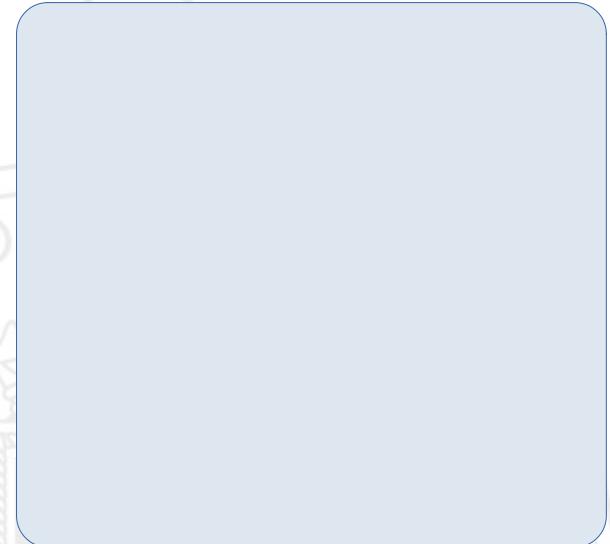
# Compartición de punteros

```
int main() {  
    Fecha *f1 = new Fecha(28, 8, 2019);  
    Fecha *f2 = f1;  
  
    f1→imprimir();  
    f2→imprimir();  
  
    f1→set_dia(1);  
  
    f1→imprimir();  
    f2→imprimir();  
  
    delete f1;  
    // delete f2  
    return 0;  
}
```

Pila



Heap



# Comparación con Java

# Java

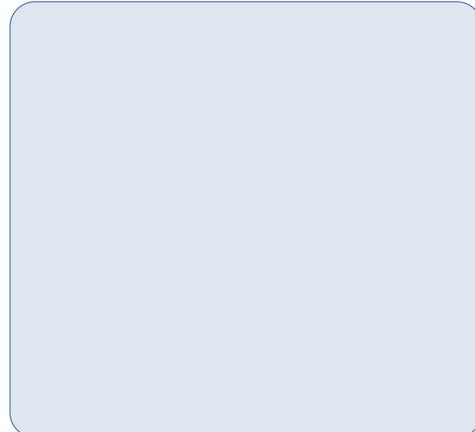
vs

# C++

Pila



Heap

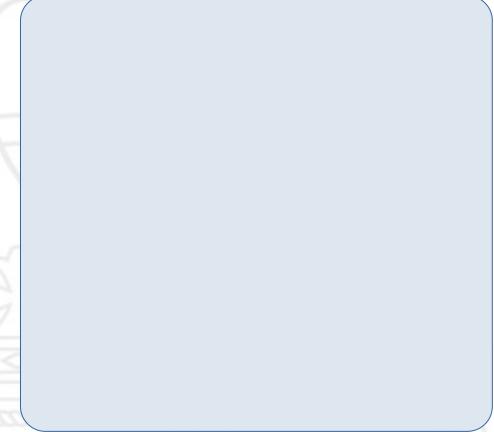


- **Todos los objetos viven en el heap.**
- La pila solo almacena valores básicos o punteros a objetos.

Pila



Heap



- Los objetos pueden almacenarse en el heap o en la pila.

# Manejo de objetos en Java y C++

```
public static void main(String[] args) {  
    Fecha f = new Fecha(20, 3, 2010);  
    f.imprimir();  
}
```

En Java tenemos que crear el objeto *f* en el *heap*, porque todos los objetos se crean allí.

```
int main() {  
    Fecha *f = new Fecha(20, 3, 2010);  
    f->imprimir();  
  
    delete f;  
    return 0;  
}
```

En C++ no es necesario crear el objeto en el *heap*.

# Manejo de objetos en Java y C++

```
public static void main(String[] args) {  
    Fecha f = new Fecha(20, 3, 2010);  
    f.imprimir();  
}
```

En Java tenemos que crear el objeto f en el *heap*, porque todos los objetos se crean allí.

```
int main() {  
    Fecha f(20, 3, 2010);  
    f.imprimir();  
  
    return 0;  
}
```

En C++ es más sencillo crear el objeto f en la pila.

# ¿Cuándo se utiliza el heap en C++?

Lo vamos a utilizar en estas situaciones:

- Cuando el tamaño de un array no es conocido en tiempo de compilación.
- Para estructuras de datos recursivas.
  - Por ejemplo, nodos de árboles y listas enlazadas.

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Destructores

Manuel Montenegro Montes

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

# Recordatorio: clase Fecha

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo);  
    Fecha(int anyo);  
    Fecha();  
  
    int get_dia() const;  
    void set_dia(int dia);  
    int get_mes() const;  
    void set_mes(int mes);  
    int get_anyo() const;  
    void set_anyo(int anyo);  
    void imprimir();  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```



# Recordatorio: clase Persona

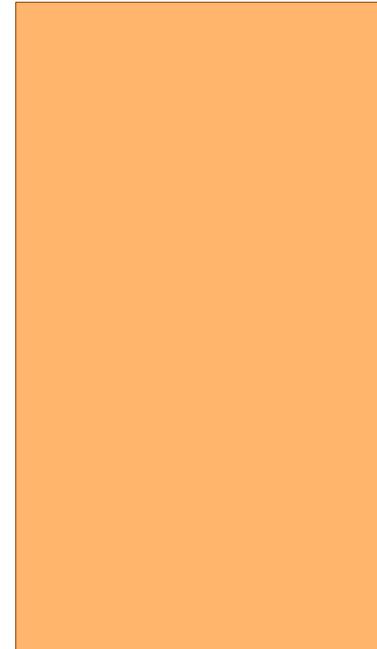
```
class Persona {  
public:  
    Persona(std::string nombre, int dia, int mes, int anyo)  
        : nombre(nombre), fecha_nacimiento(dia, mes, anyo) {}  
  
private:  
    std::string nombre;  
    Fecha fecha_nacimiento;  
};
```



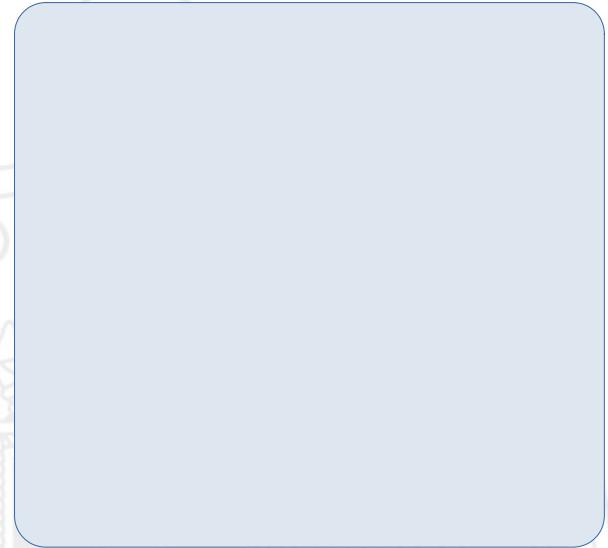
# Ejemplo de uso

```
void ejemplo() {  
    Persona p("David", 15, 3, 1979);  
}
```

Pila



Heap



# Cambio en la representación

```
class Persona {  
public:  
    Persona(std::string nombre, int dia, int mes, int anyo)  
        : nombre(nombre) {  
            this->fecha_nacimiento = new Fecha(dia, mes, anyo);  
    }  
  
private:  
    std::string nombre;  
    Fecha *fecha_nacimiento;  
};
```

# Cambio en la representación

```
class Persona {  
public:  
    Persona(std::string nombre, int dia, int mes, int anyo)  
        : nombre(nombre), fecha_nacimiento(new Fecha(dia, mes, anyo)) {}  
  
private:  
    std::string nombre;  
    Fecha *fecha_nacimiento;  
};
```

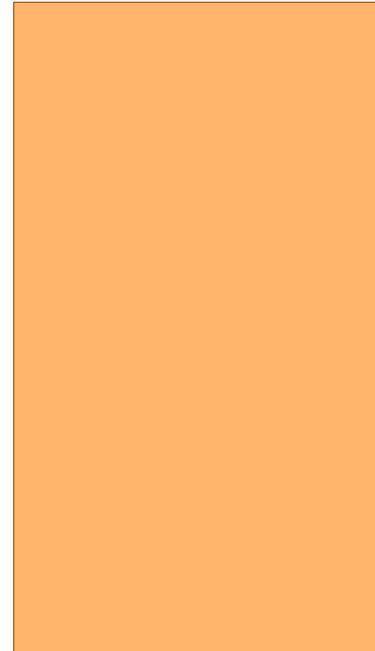


# Ejemplo de uso

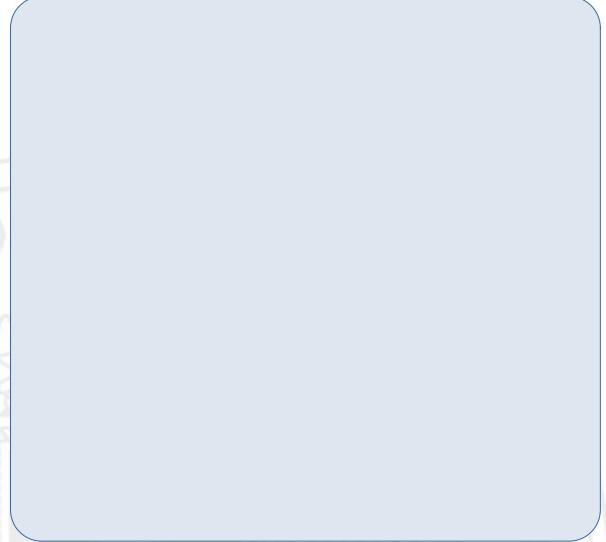
```
void ejemplo() {  
    Persona p("David", 15, 3, 1979);  
}
```

Necesitamos una manera de eliminar el objeto Fecha justo antes de que p salga de ámbito

Pila



Heap



# Destructores en C++

- Un **destructor** es un método especial que es invocado cada vez que el objeto correspondiente se libera.
  - Si el objeto está en la pila, el destructor es invocado cuando la variable que contiene dicho objeto sale de ámbito.
  - Si el objeto está en el *heap*, el destructor es invocado cuando se aplica `delete` sobre el objeto.
- El nombre del método destructor es el mismo que el de la clase en el que está definido, pero anteponiendo el símbolo `~`. 
- El método destructor no tiene ni parámetros, ni tipo de retorno.

# Añadiendo un destructor a Persona

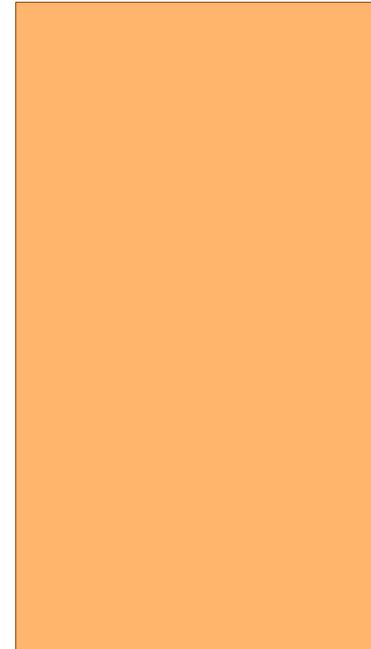
```
class Persona {  
public:  
    Persona(std::string nombre, int dia, int mes, int anyo)  
        : fecha_nacimiento(new Fecha(dia, mes, anyo)) {}  
  
    ~Persona() {  
        delete fecha_nacimiento;  
    }  
  
private:  
    std::string nombre;  
    Fecha *fecha_nacimiento;  
};
```

← Método destructor

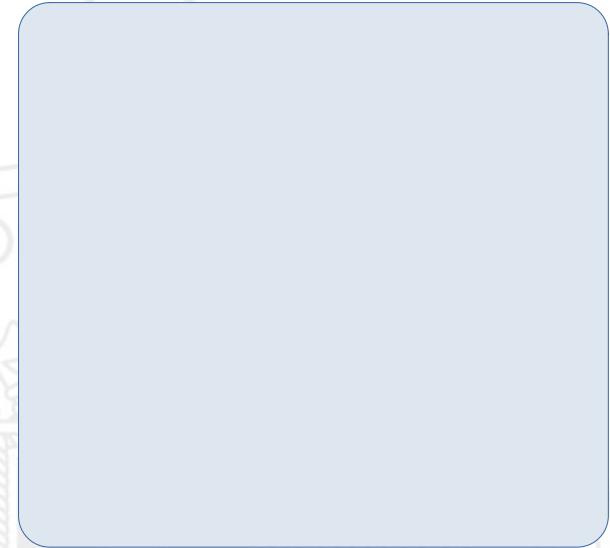
# Ejemplo de uso

```
void ejemplo() {  
    Persona p("David", 15, 3, 1979);  
}
```

Pila



Heap



# ***Resource acquisition is initialization (RAII)***

- En la gran mayoría de casos, la reserva de memoria (`new`) que se realice en el constructor debe tener asociada su liberación (`delete`) en el destructor.
- Excepciones:
  - La memoria reservada se ha liberado antes de invocar el destructor.
  - La memoria reservada está compartida entre varias instancias.
- El principio RAII no solo se aplica a memoria, sino también a otros recursos (apertura/cierre de ficheros, conexiones a bases de datos, etc.)

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Constructores de copia

Manuel Montenegro Montes

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

# Recordatorio: clases Fecha y Persona

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo);  
    Fecha(int anyo);  
    Fecha();  
  
    int get_dia() const;  
    void set_dia(int dia);  
    int get_mes() const;  
    void set_mes(int mes);  
    int get_anyo() const;  
    void set_anyo(int anyo);  
    void imprimir();  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```

```
class Persona {  
public:  
    Persona(std::string nombre,  
            int dia,  
            int mes,  
            int anyo);  
    ~Persona();  
  
    void set_nombre(const std::string &nombre);  
    void set_fecha_nacimiento(int dia,  
                             int mes,  
                             int anyo);  
    void imprimir();  
  
private:  
    std::string nombre;  
    Fecha *fecha_nacimiento;  
};
```

# Ejemplo

```
void modificar_copia(Persona p) {  
    p.set_nombre("Berta");  
    p.set_fecha_nacimiento(10, 10, 2010);  
}  
  
int main() {  
    Persona david("David", 15, 3, 1979);  
    david.imprimir();  
    modificar_copia(david);  
    david.imprimir();  
  
    return 0;  
}
```

Nombre: David  
Fecha de nacimiento: 15/03/1979

Nombre: David  
Fecha de nacimiento: 10/10/2010



# ¿Qué ha pasado?

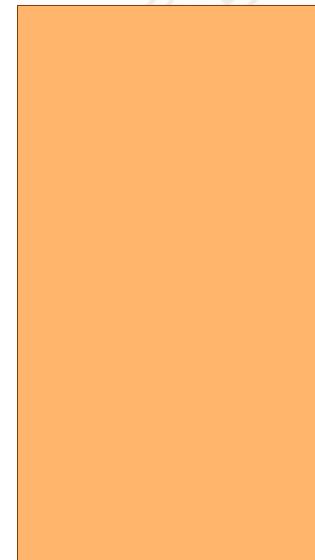
- Cuando se pasa una instancia a una función como parámetro **por valor**, se crea una copia de dicha instancia.
- **¿Cómo se realiza la copia?** Copiando el valor de cada uno de los atributos de la instancia “origen” a la instancia “destino”.

```
void modificar_copia(Persona p) { ... }

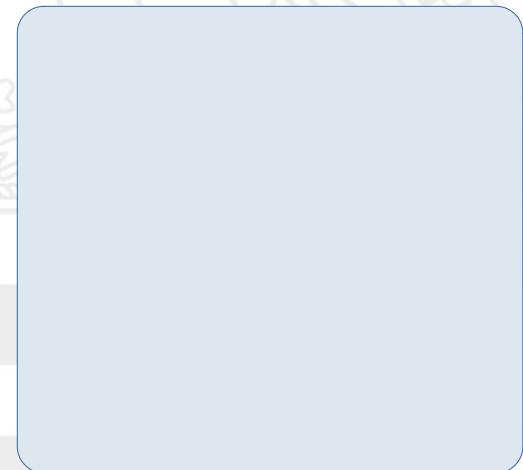
int main() {
    Persona david("David", 15, 3, 1979);
    david.imprimir();
    modificar_copia(david);
    david.imprimir();

    return 0;
}
```

Pila



Heap



# ¿Qué ha pasado?

- Copiar uno a uno los atributos funciona bien en la mayoría de los casos.
- Pero cuando los atributos son punteros a arrays u otras estructuras, solamente se hace una copia del **puntero**, de modo que tanto el objeto original como la copia, **apuntan a la misma estructura**.
- Aún peor: los **destructores** de sendas instancias pueden intentar liberar la estructura compartida **dos veces**.

¿Puede alterarse el modo en el que se realiza la copia en estos casos?

# Tipos de constructores

- **Constructor por defecto** (sin parámetros). ✓
- **Constructor paramétrico.** ✓
- **Constructor de copia.**
- **Constructor *move*.**
- **Constructor de conversión.**



# Constructor de copia

```
class Fecha {  
public:  
    ...  
    Fecha(const Fecha &f);  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```

- Es un método con el mismo nombre que la clase.
- Recibe un único parámetro: una referencia constante a un objeto de la misma clase.
- No devuelve nada.

# Constructor de copia

```
class Fecha {  
public:  
    ...  
    Fecha(const Fecha &f)  
        : dia(f.dia),  
          mes(f.mes),  
          anyo(f.anyo) { }  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```

- En el caso de Fecha, el constructor de copia inicializa los atributos del objeto con los atributos correspondientes del objeto f pasado como parámetro.
- Este es el comportamiento por defecto.

# Constructor de copia

```
class Persona {  
public:  
  
    Persona(const Persona &p)  
        : nombre(p.nombre) {  
        fecha_nacimiento =  
            new Fecha(  
                p.fecha_nacimiento->get_dia(),  
                p.fecha_nacimiento->get_mes(),  
                p.fecha_nacimiento->get_anyo()  
            );  
    }  
  
    ...  
}
```

- En el caso de Persona, el constructor de copia inicializa el atributo `fecha_nacimiento` creando un **nuevo** objeto `Fecha`, e inicializa los valores de este último con los de la fecha de `p`.

# Constructor de copia

```
class Persona {  
public:  
  
    Persona(const Persona &p)  
        : nombre(p.nombre) {  
        fecha_nacimiento =  
            new Fecha(*p.fecha_nacimiento);  
    }  
    ...  
}
```

- También podría haberse llamado explícitamente al constructor de copia de Fecha.



# Constructor de copia

```
class Persona {  
public:  
  
    Persona(const Persona &p)  
        : nombre(p.nombre),  
          fecha_nacimiento(  
              new Fecha(*p.fecha_nacimiento))  
    } { }  
  
}  
...
```

- También podría haberse llamado explícitamente al constructor de copia de Fecha.



# ¿Cuándo se llama al constructor de copia?

- Cuando se invoca explícitamente al crear un objeto.

```
Persona p1("David", 15, 3, 1979);  
Persona p2(p1);
```

- Cuando se declara una variable y se inicializa desde otro objeto.

```
Persona p1("David", 15, 3, 1979);  
Persona p2 = p1;
```

- Cuando se pasa un parámetro por valor.

```
bool es_navidad(Fecha f) { ... }  
...  
Fecha f1(15, 3, 1979);  
if(es_navidad(f1)) { ... }
```

- Cuando se devuelve un objeto como resultado.

```
Fecha nochevieja(int anyo) {  
    Fecha result(31, 12, anyo);  
    return result;  
}
```

# ¿Cuándo NO se llama?

- Cuando se asigna un objeto a una variable inicializada previamente.

```
Persona p1("David", 15, 3, 1979);  
Persona p2("Gerardo", 1, 2, 1983);  
p2 = p1;
```

No se llama al  
constructor de copia

```
Persona p1("David", 15, 3, 1979);  
Persona p2 = p1;
```

Sí se llama al  
constructor de copia

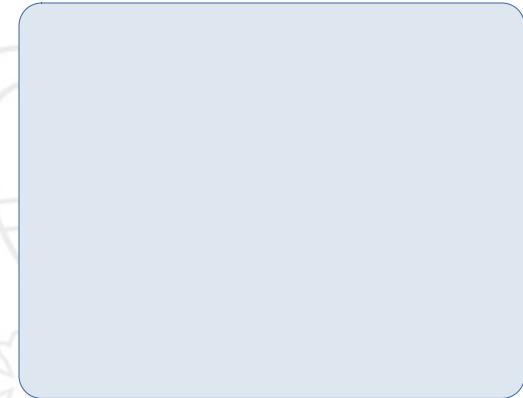
# Volviendo a nuestro ejemplo...

```
void modificar_copia(Persona p) {  
    p.set_nombre("Berta");  
    p.set_fecha_nacimiento(10, 10, 2010);  
}  
  
int main() {  
    Persona david("David", 15, 3, 1979);  
    david.imprimir();  
    modificar_copia(david);  
    david.imprimir();  
  
    return 0;  
}
```

Pila



Heap



Nombre: David  
Fecha de nacimiento: 15/03/1979

Nombre: David  
Fecha de nacimiento: 15/03/1979

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Sobrecarga de operadores

Manuel Montenegro Montes

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

# Ejemplo: números complejos

```
class Complejo {  
public:  
    Complejo(double real, double imag);  
  
    double get_real() const;  
    double get_imag() const;  
    void display() const;  
  
private:  
    double real, imag;  
};
```

Existe la clase `std :: complex`, definida en `<complex>`

# Aritmética con números complejos

```
Complejo suma(const Complejo &z1, const Complejo &z2) {
    return { z1.get_real() + z2.get_real(),
              z1.get_imag() + z2.get_imag() };
}

Complejo multiplica(const Complejo &z1, const Complejo &z2) {
    double z1_real = z1.get_real(), z1_imag = z1.get_imag();
    double z2_real = z2.get_real(), z2_imag = z2.get_imag();
    return { z1_real * z2_real - z1_imag * z2_imag,
              z1_real * z2_imag + z1_imag * z2_real };
}
```

# Ejemplo de uso

```
int main() {  
    Complejo z1(2.0, -3.0), z2(1.0, 0.0);  
    Complejo z3 = suma(z1, z2);  
    Complejo z4 = suma(multiplica(z1, z1), z2);  
  
    z3.display();  
    std::cout << std::endl;  
  
    z4.display();  
    std::cout << std::endl;  
  
    return 0;  
}
```

3-3i

-4-12i



# Uso de operadores

- Con los tipos numéricos básicos (`int`, `double`, etc.) podemos expresar operaciones aritméticas utilizando los operadores `+` y `*` en forma infija.
  - Ejemplo: `x + y * z`
- Con nuestra clase `Complejo` no tenemos la misma suerte:
  - `suma(z1, z2)`
  - `suma(multiplica(z1, z1), z2)`
- Sería más legible poder escribir:
  - `z1 + z2`
  - `z1 * z1 + z2`
- En C++ es posible definir implementaciones personalizadas de los operadores, es decir, **sobrecargarlos**.

# Sobrecargar operadores

# Aritmética con números complejos

```
Complejo suma(const Complejo &z1, const Complejo &z2) {
    return { z1.get_real() + z2.get_real(),
              z1.get_imag() + z2.get_imag() };
}

Complejo multiplica(const Complejo &z1, const Complejo &z2) {
    double z1_real = z1.get_real(), z1_imag = z1.get_imag();
    double z2_real = z2.get_real(), z2_imag = z2.get_imag();
    return { z1_real * z2_real - z1_imag * z2_imag,
              z1_real * z2_imag + z1_imag * z2_real };
}
```

# Aritmética con números complejos

```
Complejo operator+(const Complejo &z1, const Complejo &z2) {
    return { z1.get_real() + z2.get_real(),
              z1.get_imag() + z2.get_imag() };
}

Complejo operator*(const Complejo &z1, const Complejo &z2) {
    double z1_real = z1.get_real(), z1_imag = z1.get_imag();
    double z2_real = z2.get_real(), z2_imag = z2.get_imag();
    return { z1_real * z2_real - z1_imag * z2_imag,
              z1_real * z2_imag + z1_imag * z2_real };
}
```

- Puede sobrecargarse un operador creando una función con nombre **operator[?]**, donde **[?]** es un operador de C++.

# Ejemplo de uso

```
int main() {
    Complejo z1(2.0, -3.0), z2(1.0, 0.0);
    Complejo z3 = suma(z1, z2);
    Complejo z4 = suma(multiplica(z1, z1), z2);

    z3.display();
    std::cout << std::endl;

    z4.display();
    std::cout << std::endl;

    return 0;
}
```



# Ejemplo de uso

```
int main() {  
    Complejo z1(2.0, -3.0), z2(1.0, 0.0);  
    Complejo z3 = z1 + z2;—  
    Complejo z4 = z1 * z1 + z2;  
  
    z3.display();  
    std::cout << std::endl;  
  
    z4.display();  
    std::cout << std::endl;  
  
    return 0;  
}
```

Equivale a  
operator+(z1, z2)

Equivale a  
operator+(operator\*(z1, z1), z2)

# ¿Qué operadores pueden sobrecargarse?

+ - \* / % ^ & | << >>  
== <= >= != < > && || !  
= += -= \*= /=  
++ --  
[] () →  
new delete  
etc.

# Sobrecarga del operador << para E/S

# Generalizando el método `display()`

```
class Complejo {  
public:  
    ...  
    void display() const;  
private:  
    ...  
};
```



```
    void Complejo::display() const {  
        std::cout << real << ... << "i";  
    }
```

- El método `display()` envía una representación en cadena del objeto a la salida estándar (`std :: cout`).
  - ¿Y si quiero escribirla en un fichero (clase `ofstream`)?
  - ¿Y si quiero escribirla un `string` (clase `ostringstream`)?
- Todas heredan de la clase `ostream`.

# Generalizando el método `display()`

```
class Complejo {  
public:  
    ...  
    void display(ostream &out) const; → } {  
private:  
    ...  
};
```

void Complejo::display(ostream &out) const {  
 out << real << ... << "i";  
}

- El método `display()` envía una representación en cadena del objeto a la salida estándar (`std :: cout`).
- ¿Y si quiero escribirla en un fichero (clase `ofstream`)?
- ¿Y si quiero escribirla un `string` (clase `ostringstream`)?  
Todas heredan de la clase `ostream`.

# Actualizando el ejemplo

```
int main() {  
    Complejo z1(2.0, -3.0), z2(1.0, 0.0);  
    Complejo z3 = z1 + z2;  
    Complejo z4 = z1 * z1 + z2;  
  
    z3.display(std::cout);  
    std::cout << std::endl;  
  
    z4.display(std::cout);  
    std::cout << std::endl;  
  
    return 0;  
}
```



# El operador << para E/S

```
std :: cout << "Hola"
```

Instancia de  
ostream

Instancia de  
string

```
std :: cout << z1
```

Instancia de  
ostream

Instancia de  
Complejo

# Sobrecargando << para números complejos

```
void operator<<(std::ostream &out, Complejo &z) {  
    z.display(out);  
}
```

```
int main() {  
    ...  
  
    z3.display(std::cout);  
    std::cout << std::endl;  
  
    z4.display(std::cout);  
    std::cout << std::endl;  
  
    return 0;  
}
```

```
int main() {  
    ...  
  
    std::cout << z3;  
    std::cout << std::endl;  
  
    std::cout << z4;  
    std::cout << std::endl;  
  
    return 0;  
}
```

# Sobrecargando << para números complejos

```
void operator<<(std::ostream &out, Complejo &z) {  
    z.display(out);  
}
```

```
int main() {  
    ...  
  
    z3.display(std::cout);  
    std::cout << std::endl;  
  
    z4.display(std::cout);  
    std::cout << std::endl;  
  
    return 0;  
}
```

```
int main() {  
    ...  
  
    std::cout << z3 << std::endl << z4 << std::endl; X  
  
    return 0;  
}
```

# Sobrecargando << para números complejos

```
std::ostream & operator<<(std::ostream &out, Complejo &z) {  
    z.display(out);  
    return out;  
}
```

```
std::cout << z3 << std::endl << z4 << std::endl; ✓
```

# Sobrecarga dentro de una clase

# Sobrecarga fuera de una clase

- Las definiciones de sobrecarga vistas hasta ahora son funciones que no pertenecen a ninguna clase:

```
Complejo operator+(const Complejo &z1, const Complejo &z2) {  
    return { z1.get_real() + z2.get_real(),  
             z1.get_imag() + z2.get_imag() };  
}
```

# Sobrecarga dentro de una clase

- También habríamos podido definirlas como métodos de la clase Complejo.
- Si lo hacemos así, el primer operando es `this`.
- Ventaja: podemos acceder a los atributos privados.

```
class Complejo {  
public:  
    ...  
  
    Complejo operator+(const Complejo &z2) const {  
        return { real + z2.real, imag + z2.imag };  
    }  
  
private:  
    double real, imag;  
};
```

$z1 + z2$

equivale a

`z1.operator+(z2)`

# ¿Podemos hacer lo mismo con...?

```
Complejo operator*(const Complejo &z1, const Complejo &z2) {  
    double z1_real = z1.get_real(), z1_imag = z1.get_imag();  
    double z2_real = z2.get_real(), z2_imag = z2.get_imag();  
    return { z1_real * z2_real - z1_imag * z2_imag,  
             z1_real * z2_imag + z1_imag * z2_real };  
}
```

```
std::ostream & operator<<(std::ostream &out, Complejo &z) {  
    z.display(out);  
    return out;  
}
```



¡No podemos añadir  
métodos a la clase  
`ostream`!

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Operador de asignación

Manuel Montenegro Montes

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

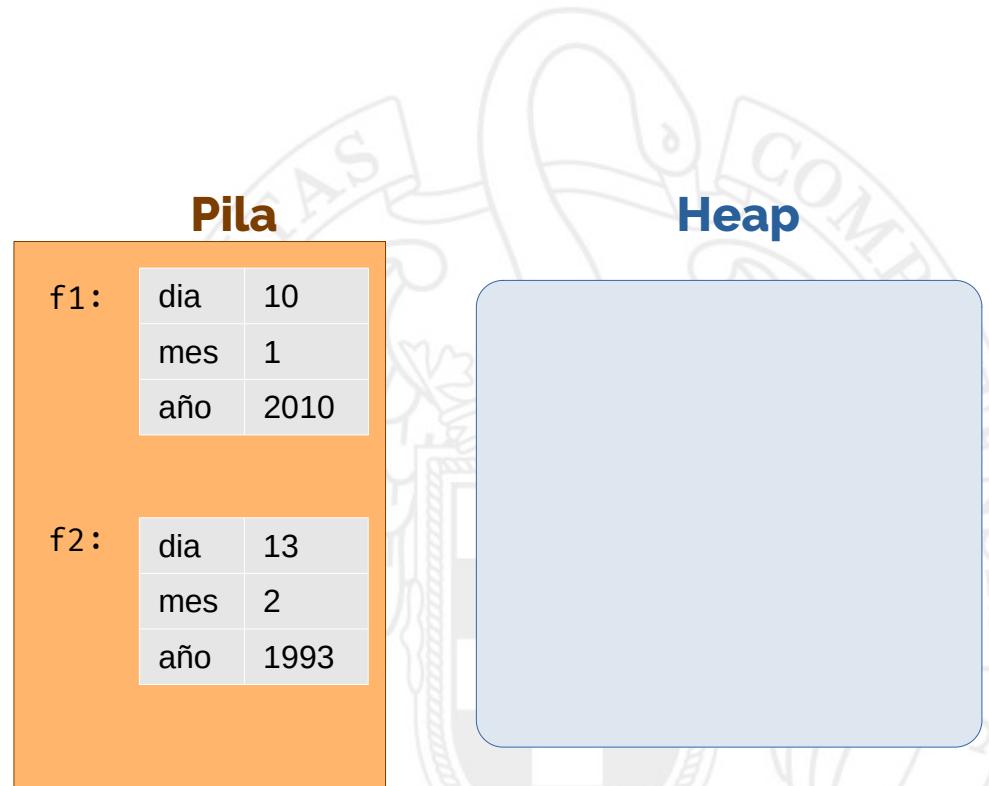
# Recordatorio: clase Fecha

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo);  
    Fecha(int anyo);  
    Fecha();  
  
    int get_dia() const;  
    void set_dia(int dia);  
    int get_mes() const;  
    void set_mes(int mes);  
    int get_anyo() const;  
    void set_anyo(int anyo);  
    void imprimir();  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```



# Asignar un objeto Fecha a otro

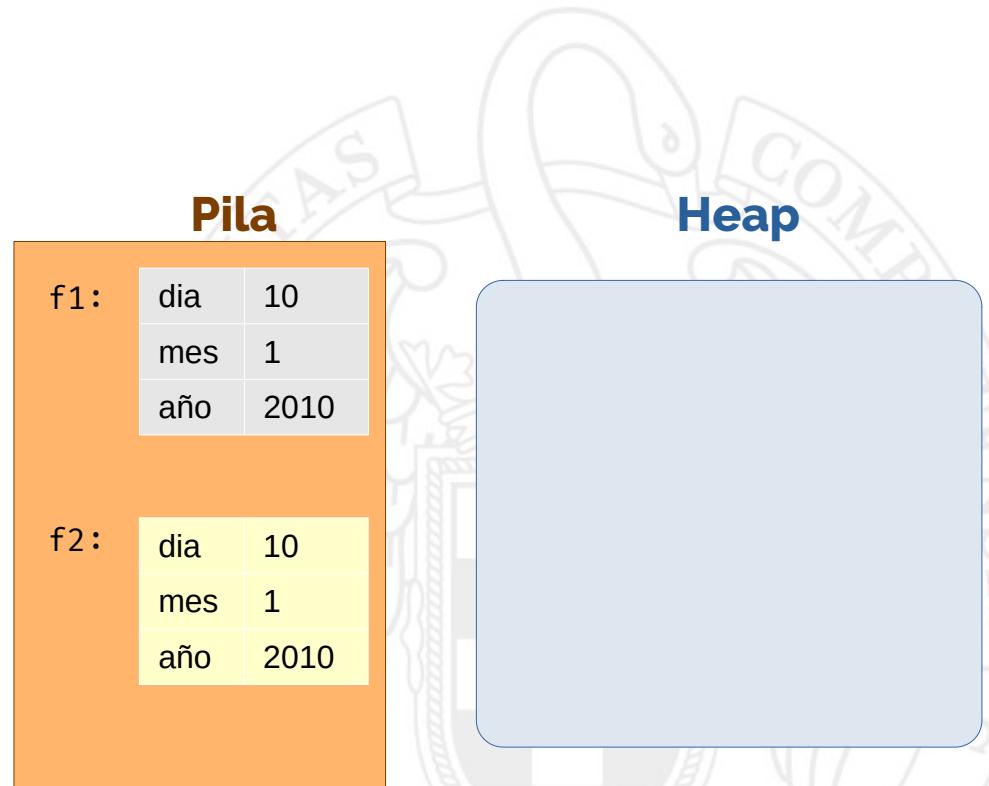
```
Fecha f1(10, 1, 2010);  
Fecha f2(13, 2, 1993);  
  
f2 = f1;
```



# Asignar un objeto Fecha a otro

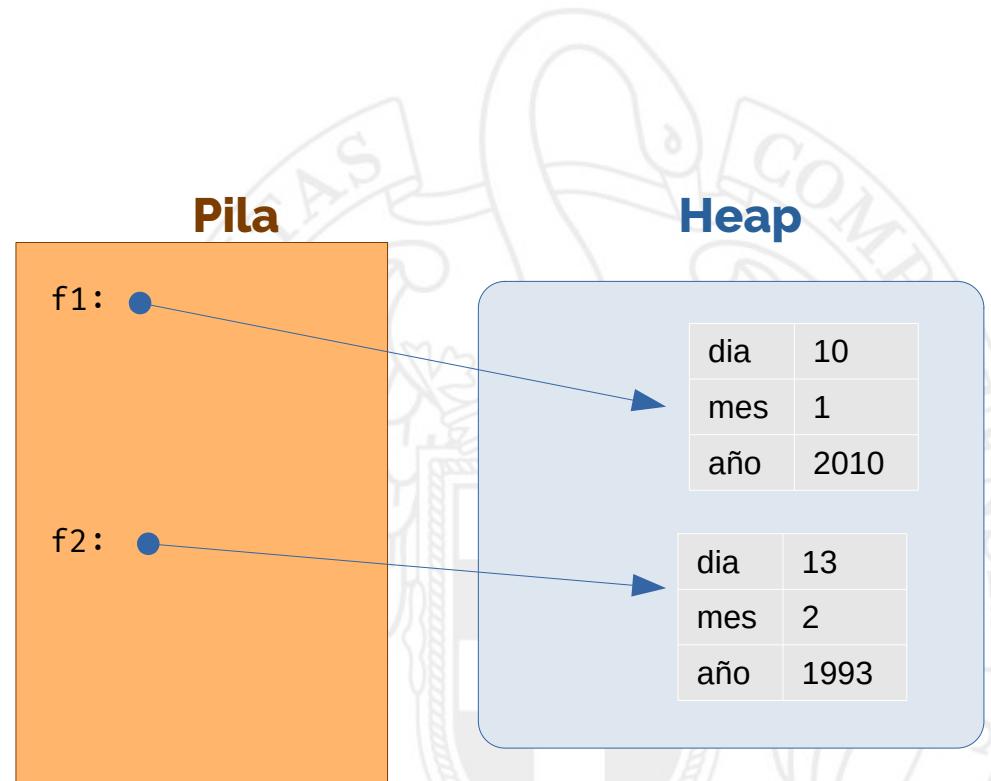
```
Fecha f1(10, 1, 2010);  
Fecha f2(13, 2, 1993);
```

```
f2 = f1;
```



# Asignar un objeto Fecha a otro

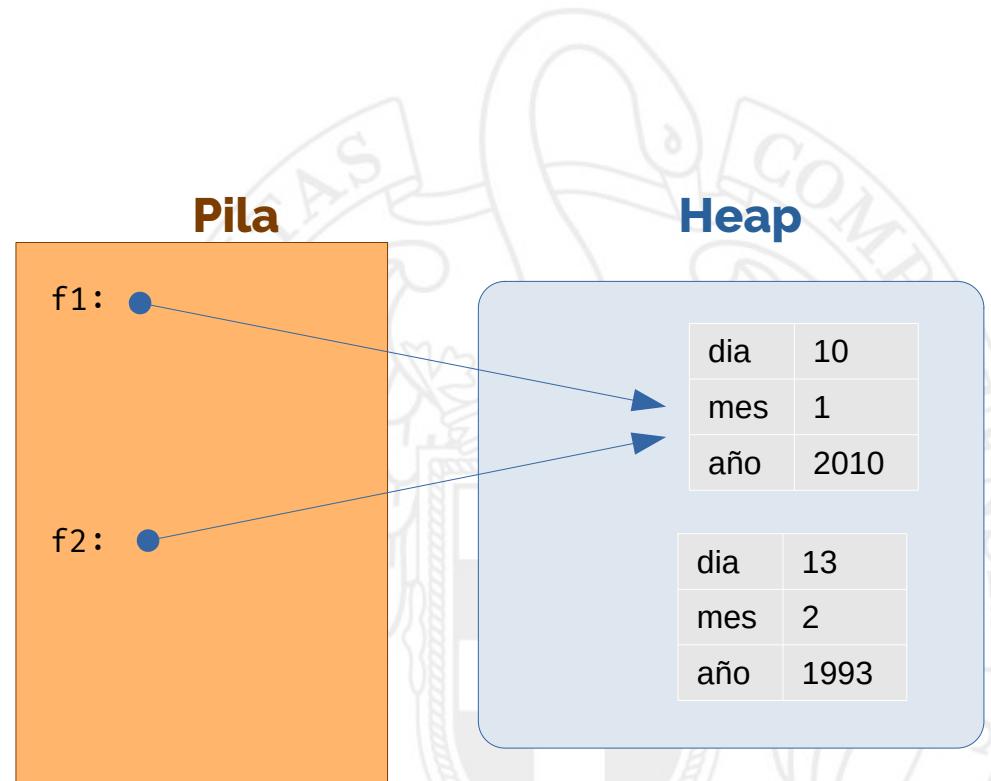
```
Fecha *f1 = new Fecha(10, 1, 2010);  
Fecha *f2 = new Fecha(13, 2, 1993);  
  
f2 = f1;
```



# Asignar un objeto Fecha a otro

```
Fecha *f1 = new Fecha(10, 1, 2010);  
Fecha *f2 = new Fecha(13, 2, 1993);
```

```
f2 = f1;
```



# Recordatorio: clase Persona

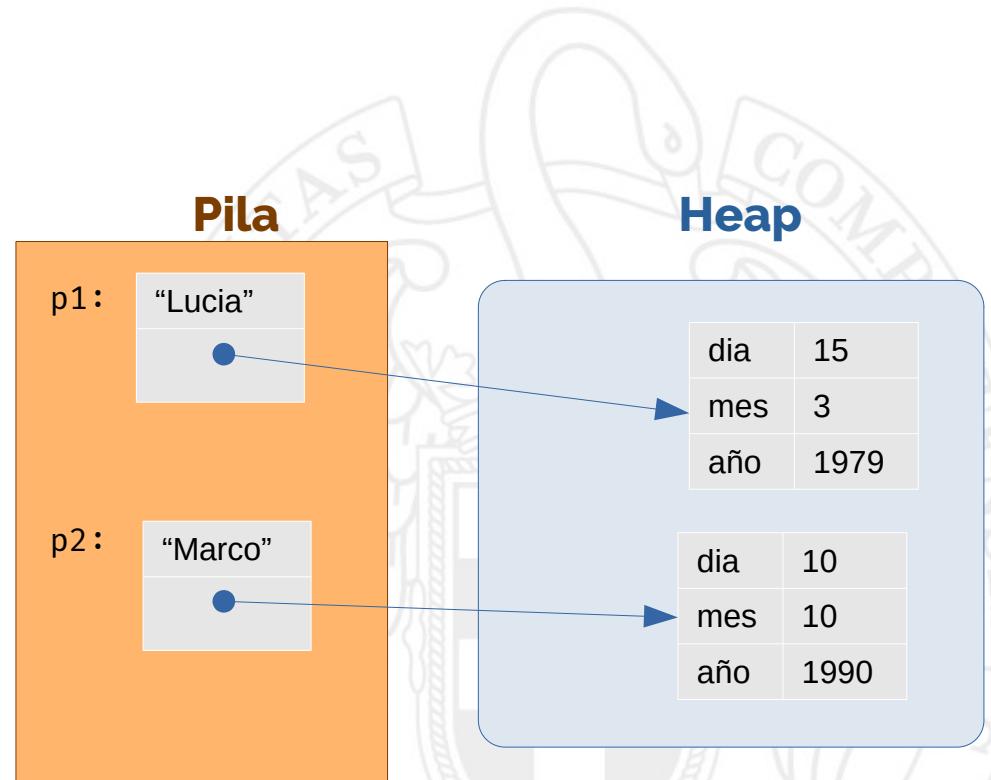
```
class Persona {  
public:  
    Persona(std::string nombre,  
            int dia,  
            int mes,  
            int anyo);  
  
    ~Persona() {  
        delete fecha_nacimiento;  
    }  
  
    ...  
  
private:  
    std::string nombre;  
    Fecha *fecha_nacimiento;  
};
```



# Asignar un objeto Persona a otro

```
Persona p1("Lucía", 15, 3, 1979);  
Persona p2("Marco", 10, 10, 1990);
```

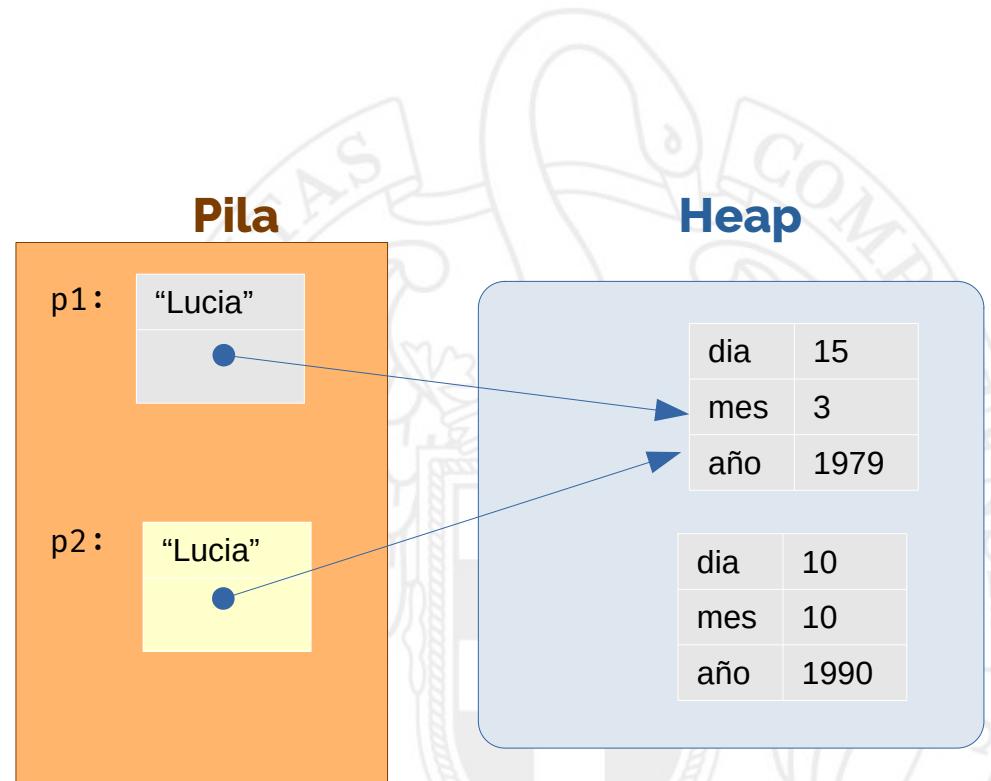
```
p2 = p1;
```



# Asignar un objeto Persona a otro

```
Persona p1("Lucía", 15, 3, 1979);  
Persona p2("Marco", 10, 10, 1990);
```

```
p2 = p1;
```



# Sobrecargando el operador de asignación

```
class Persona {  
public:  
    ...  
  
    void operator=(const Persona &other) {  
        nombre = other.nombre;  
        fecha_nacimiento→set_dia(other.fecha_nacimiento→get_dia());  
        fecha_nacimiento→set_mes(other.fecha_nacimiento→get_mes());  
        fecha_nacimiento→set_anyo(other.fecha_nacimiento→get_anyo());  
    }  
  
    ...  
  
private:  
    std::string nombre;  
    Fecha *fecha_nacimiento;  
};
```

p2 = p1

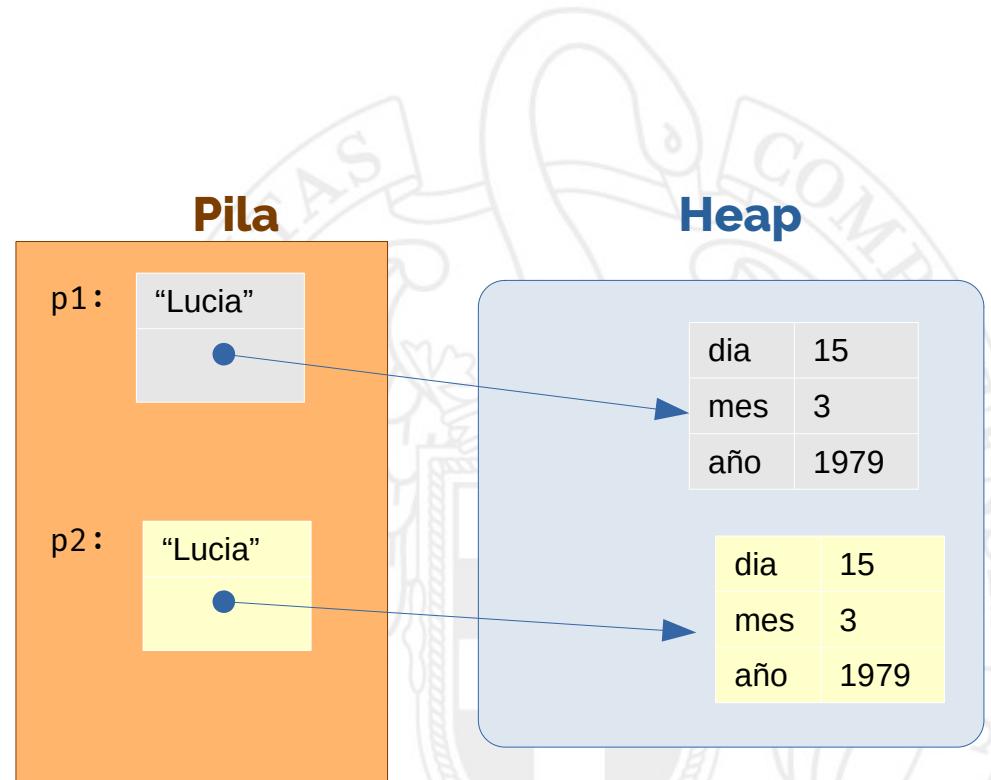
equivale a

p2.operator=(p1)

# Asignar un objeto Persona a otro

```
Persona p1("Lucía", 15, 3, 1979);  
Persona p2("Marco", 10, 10, 1990);
```

```
p2 = p1;
```



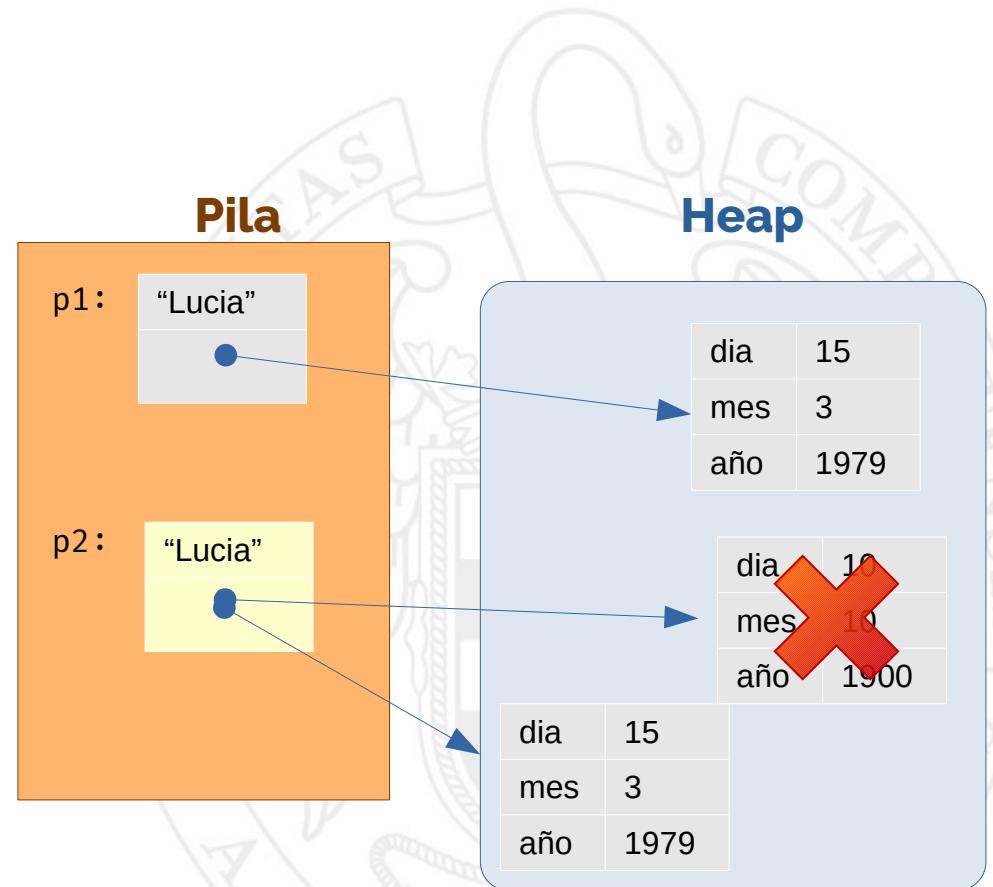
# Otra posibilidad

```
class Persona {  
public:  
  
    ...  
  
    void operator=(const Persona &other) {  
        nombre = other.nombre;  
        delete fecha_nacimiento;  
        fecha_nacimiento = new Fecha(*other.fecha_nacimiento);  
    }  
  
    ...  
  
private:  
    std::string nombre;  
    Fecha *fecha_nacimiento;  
};
```

# Asignar un objeto Persona a otro

```
Persona p1("Lucía", 15, 3, 1979);  
Persona p2("Marco", 10, 10, 1990);
```

```
p2 = p1;
```

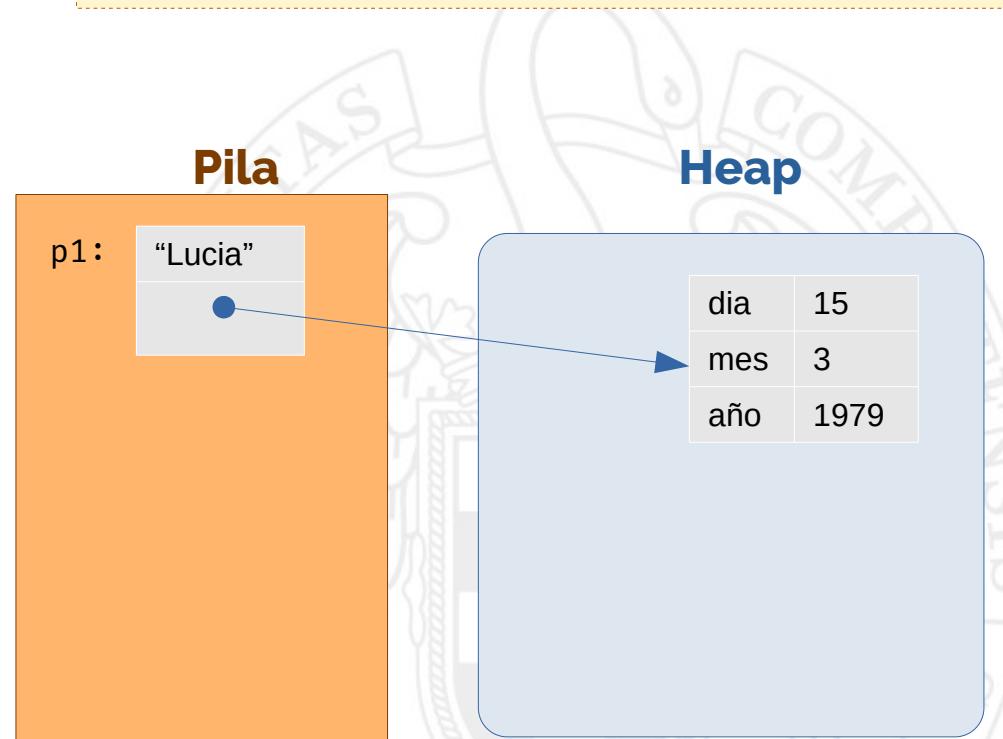


# **El problema de la autoasignación**

# Asignar un objeto persona a sí mismo

```
Persona p1("Lucía", 15, 3, 1979);  
p1 = p1;
```

```
void operator=(const Persona &other) {  
    nombre = other.nombre;  
    delete fecha_nacimiento;  
    fecha_nacimiento = new Fecha(*other.fecha_nacimiento);  
}
```



# Evitando la autoasignación

```
class Persona {  
public:  
  
...  
  
void operator=(const Persona &other) {  
    if (this != &other) {  
        nombre = other.nombre;  
        delete fecha_nacimiento;  
        fecha_nacimiento = new Fecha(*other.fecha_nacimiento);  
    }  
}  
...  
  
private:  
    std::string nombre;  
    Fecha *fecha_nacimiento;  
};
```



# Encadenar asignaciones

# Encadenar asignaciones

```
int x, y, z;  
x = y = z = 0;
```

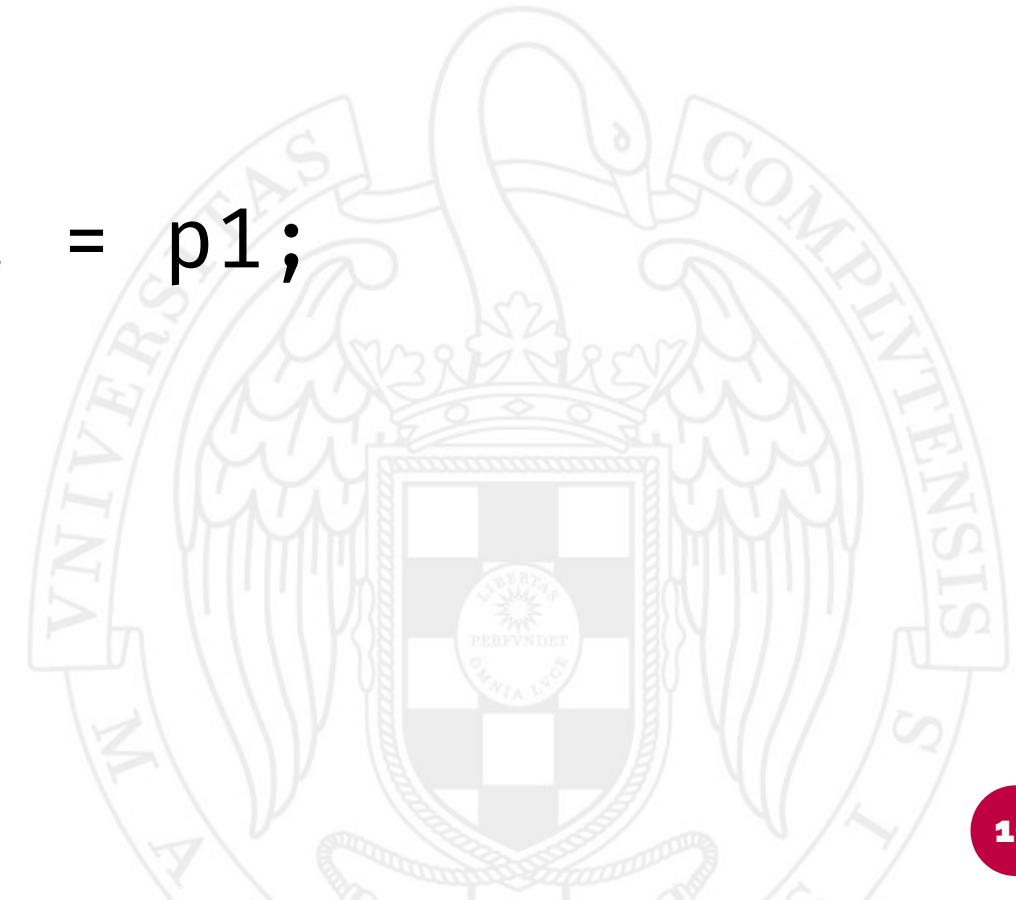
x = y = z = 0;

# ¿Podemos hacer lo mismo con objetos Persona?

```
Persona p1("Lucía", 15, 3, 1979);  
Persona p2("Marco", 10, 10, 1990);  
Persona p3("Laura", 1, 3, 1980);
```

p3 = p2 = p1; 

p3 = p2 = p1;



# Devolviendo referencia a this

```
class Persona {  
public:  
  
...  
  
    Persona & operator=(const Persona &other) {  
        if (this != &other) {  
            nombre = other.nombre;  
            delete fecha_nacimiento;  
            fecha_nacimiento = new Fecha(*other.fecha_nacimiento);  
        }  
  
        return *this;  
    }  
  
...  
  
private:  
    std::string nombre;  
    Fecha *fecha_nacimiento;  
};
```

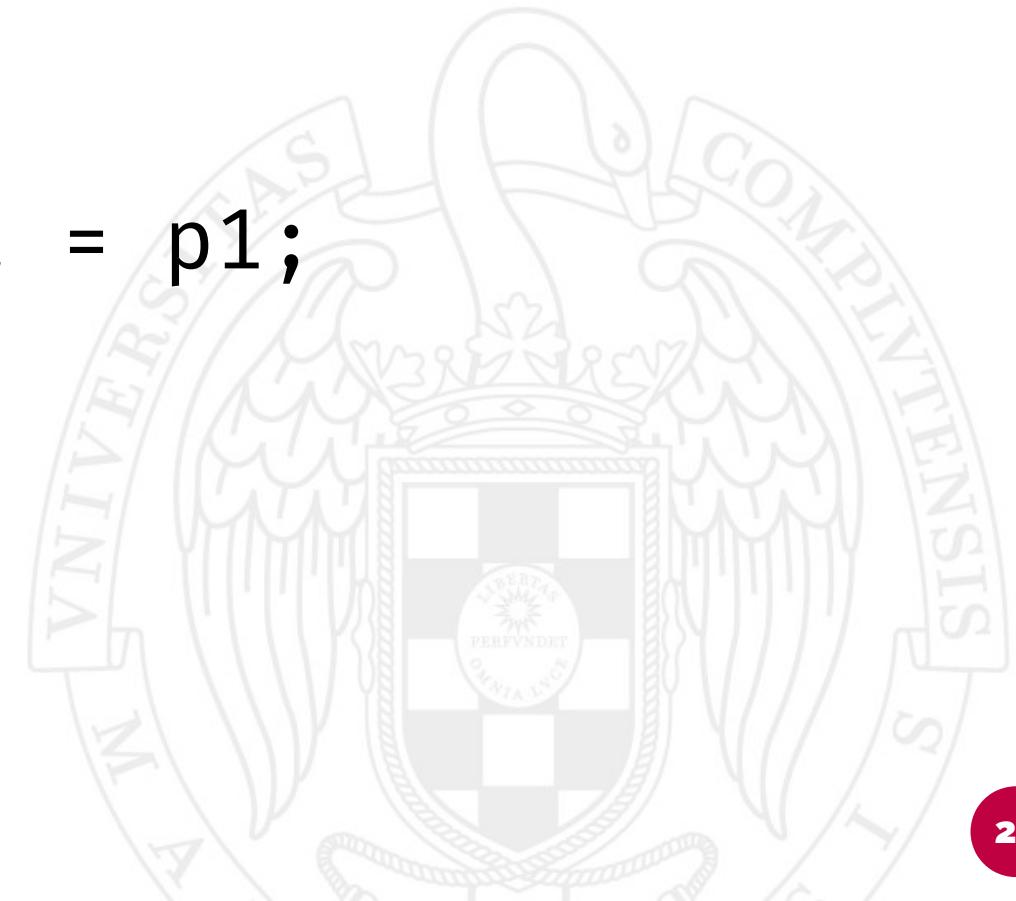


# ¿Podemos hacer lo mismo con objetos Persona?

```
Persona p1("Lucía", 15, 3, 1979);  
Persona p2("Marco", 10, 10, 1990);  
Persona p3("Laura", 1, 3, 1980);
```

p3 = p2 = p1; 

p3 = p2 = p1;



# Constructor de copia    vs.    Operador asignación

- Para crear un objeto nuevo con la misma información que otro existente.

```
Persona p1(...);  
Persona p2 = p1;
```

- No devuelve nada.
- No puede producirse autoasignación:

```
Persona p2 = p2;
```

- Para copiar la información de un objeto existente a otro existente.

```
Persona p1(...);  
Persona p2(...);  
p2 = p1;
```

- Devuelve `*this`.
- Hay que tener en cuenta la autoasignación:

```
p2 = p2;
```

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Plantillas en funciones

Manuel Montenegro Montes

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

# Ejemplo

- Implementamos una función que calcula el mínimo de dos enteros:

```
int min(int a, int b) {  
    if (a ≤ b) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

- ¿Y si quiero calcular el mínimo de dos `float`? ¿y el mínimo de dos `double`?
- ¿Y si quiero calcular el mínimo de dos `string` utilizando el orden lexicográfico? Por ejemplo: `min("AA", "AB") = "AA"`.

# Ejemplo

```
int min(int a, int b) {  
    if (a <= b) {  
        return a;  
    } else {  
        return b;  
    }  
}  
  
float min(float a, float b) {  
    if (a <= b) {  
        return a;  
    } else {  
        return b;  
    }  
}  
  
double min(double a, double b) {  
    if (a <= b) {  
        return a;  
    } else {  
        return b;  
    }  
}  
  
const std::string & min(const std::string &a, const std::string &b) {  
    if (a <= b) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

- ¡Cuanta duplicidad!
- Todas tienen la misma implementación. ¡Solo difieren en los tipos!

# Programación genérica

- Sería deseable tener una única versión genérica, que pudiese funcionar con varios tipos.

```
??? min( ??? a, ??? b) {  
    if (a <= b) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

- **Solución:** plantillas (*templates*) en C++.



# Plantillas en C++

- Son definiciones con «huecos» (**parámetros de plantilla**).
- Se especifican mediante la palabra `template`, seguida de los parámetros de plantilla, y seguida de la definición de función paramétrica.

```
template <typename T>
T min(T a, T b) {
    if (a ≤ b) {
        return a;
    } else {
        return b;
    }
}
```

# Llamada a funciones plantilla

- Basta con indicar el tipo con el que queremos «rellenar» el marcador.

```
min<int>(6, 2)
min<double>(3.3, 5.5)
min<std::string>("Pepito", "Paula")
```

- Cada vez que se hace una llamada a la función genérica, se hace una versión específica para el tipo indicado en el marcador. A esto se le llama **instanciación de plantillas**.

```
template <typename T>
T min(T a, T b) {
    if (a ≤ b) {
        return a;
    } else {
        return b;
    }
}
```

T = int  
→

```
int min<int>(int a, int b) {
    if (a ≤ b) {
        return a;
    } else {
        return b;
    }
}
```

# Instanciación de plantillas

```
template <typename T>
T min(T a, T b) {
    if (a ≤ b) {
        return a;
    } else {
        return b;
    }
}
```

T = int

T = std::string

T = double

```
std::string min(std::string a, std::string b) {
    if (a ≤ b) {
        return a;
    } else {
        return b;
    }
}
```

```
int min<int>(int a, int b) {
    if (a ≤ b) {
        return a;
    } else {
        return b;
    }
}
```

```
double min<double>(double a, double b) {
    if (a ≤ b) {
        return a;
    } else {
        return b;
    }
}
```

# Instanciación de plantillas

- En esta última instancia (con un `string`) podemos indicar un tipo más preciso:

```
std::cout << min<const std::string &>("Pepito", "Ramiro") << std::endl;
```

- O bien modificar nuestra función genérica:

```
template <typename T>
const T & min(const T &a, const T &b) {
    if (a <= b) {
        return a;
    } else {
        return b;
    }
}
```

# Deducción de argumentos de plantilla

- Cada vez que hemos llamado a una función genérica, hemos indicado el tipo con el que debe instanciarse:

```
std::cout << min<std::string>("Pepito", "Ramiro") << std::endl;
```

- C++ permite omitirlo en la mayoría de los casos.
  - En ese caso intenta deducir el argumento de la plantilla.

```
std::cout << min("Pepito", "Ramiro") << std::endl;
```



# ¡Cuidado con las instanciaciones!

- ¿Qué pasa si instancio la plantilla con dos complejos?

```
Complejo z1(1.0, 3.0), z2(4.0, -5.0);  
std::cout << min(z1, z2) << std::endl;
```

- C++ realiza esta instancia:

```
template <typename T>  
const T & min(const T &a, const T &b) {  
    if (a <= b) {  
        return a;  
    } else {  
        return b;  
    }  
}
```

T = Complejo →

```
const Complejo & min(const Complejo &a,  
                      const Complejo &b)  
{  
    if (a <= b) {  
        return a;  
    } else {  
        return b;  
    }  
}
```



# ¡Cuidado con las instanciaciones!

- Los errores provocados por instanciaciones incorrectas suelen ser crípticos, largos, y difíciles de interpretar:

Test1.cpp: En la instanciaación de ‘const T& min(const T&, const T&) [con T = Complejo]’:

Test1.cpp:76:28: se requiere desde aquí

Test1.cpp:40:11: error: no match for ‘operator<=’ (operand types are ‘const Complejo’ and ‘const Complejo’)

```
40 |     if (a <= b) {  
|         ~~~~^~~~~~
```

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Plantillas en clases

Manuel Montenegro Montes

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

# Repaso: números complejos

```
class Complejo {  
public:  
    Complejo(double real, double imag);  
  
    double get_real() const;  
    double get_imag() const;  
    void display() const;  
  
    Complejo operator+(const Complejo &z) const;  
    Complejo operator*(const Complejo &z) const;  
  
private:  
    double real, imag;  
};
```

- ¿Y si quisiera también una clase Complejo en la que las partes reales o imaginarias sean float, en lugar de double?
- Para evitar duplicidad de código puedo utilizar plantillas.

# Generalización de una clase

```
template<typename T>
class Complejo {
public:
    Complejo(T real, T imag);

    T get_real() const;
    T get_imag() const;

    void display(std::ostream &out) const;

    Complejo operator+(const Complejo &z) const;
    Complejo operator*(const Complejo &z) const;

private:
    T real, imag;
};
```



# Generalización de los métodos

```
template<typename T>
class Complejo {
public:
    ...
    T get_real() const {
        return real;
    };
    T get_imag() const {
        return imag;
    };
    ...
private:
    T real, imag;
};
```

- Si el método se implementa dentro de la clase, no es necesario hacer nada nuevo.

# Generalización de los métodos

```
template<typename T>
class Complejo {
public:
    ...
    Complejo operator+(const Complejo &z1) const;
    Complejo operator*(const Complejo &z1) const;
    ...
private:
    T real, imag;
};

template<typename T>
Complejo<T> Complejo<T>::operator+(const Complejo<T> &z) const {
    return { real + z.real, imag + z.imag };
}
```

- Si el método se implementa fuera de la clase, es necesario indicar que el método también es una plantilla.

# Uso de una clase genérica

- A la hora de crear una instancia de una clase genérica, hay que indicar el tipo con el que se instancia:

```
Complejo<double> z1(2.0, -3.0), z2(1.0, 0.0);
```

```
Complejo<float> z4(2.0, -3.0);
```



*En las clases, es **obligatorio** indicar el tipo con el que instanciar la plantilla*

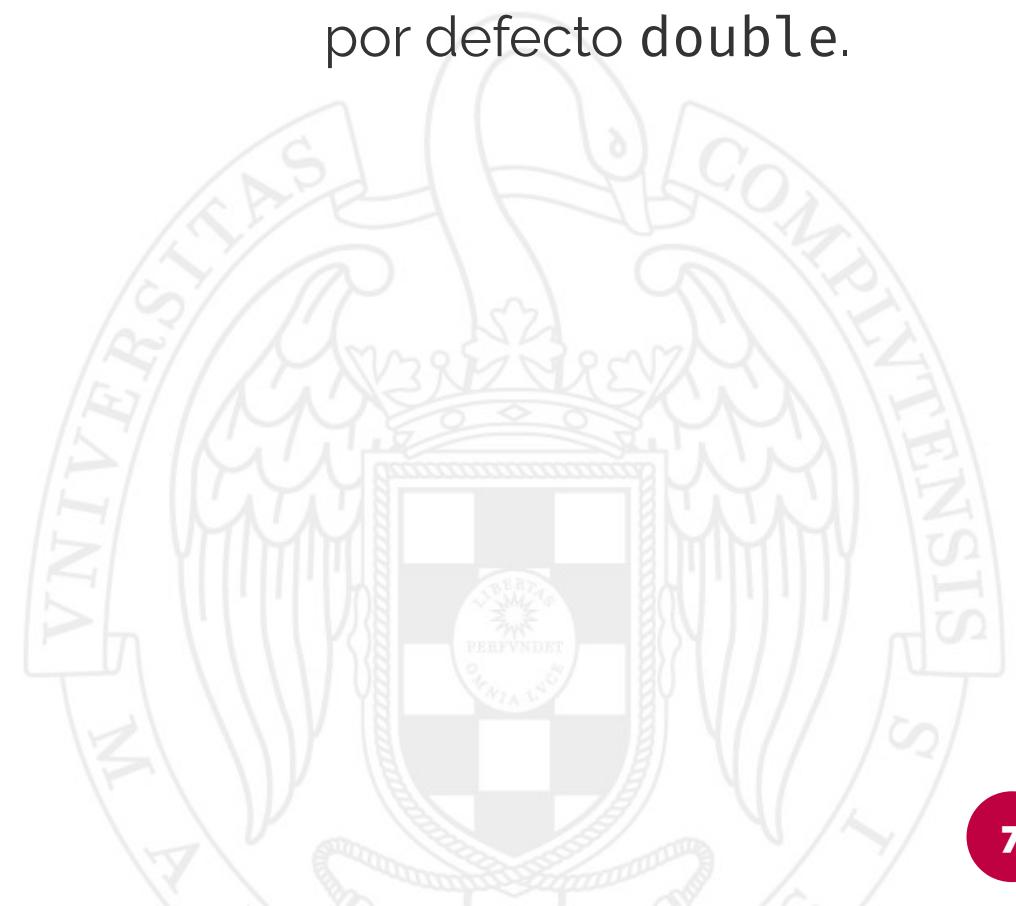
Complejo z4(2.0, -3.0); 

- ... aunque es posible indicar un tipo por defecto para la instancia.

# Plantillas: argumentos por defecto

```
template<typename T = double>
class Complejo {
    ...
};
```

- Si no se indica el tipo en la instancia, se utilizará por defecto double.



# Plantillas: argumentos por defecto

- Aún si queremos utilizar argumentos por defecto, es necesario indicar los delimitadores < y >, aunque no tengan nada en su interior.

```
Complejo<> z1(2.0, -3.0), z2(1.0, 0.0); ← Correcto (= Complejo<double>)  
Complejo z1(2.0, -3.0), z2(1.0, 0.0); ← Incorrecto
```

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# STL: Contenedores lineales

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

# ¿Qué es la STL?

**STL = *Standard Template Library***

- Es una librería estándar de C++ que proporciona una serie de utilidades al programador/a.
  - Tipos abstractos de datos para almacenar colecciones de elementos: listas, pilas, colas, conjuntos, diccionarios, etc.
  - Iteradores.
  - Algoritmos sobre estos tipos abstractos de datos.

# Tipos de datos lineales en la STL

Clase	Fich. cabecera	Estructura
std::vector	<vector>	TAD Lista (arrays)
std::list	<list>	TAD Lista (listas doblemente enlazadas)
std::forward_list	<forward_list>	TAD Lista (listas enlazadas simples)
std::deque	<deque>	TAD doble cola
std::stack	<stack>	TAD pila
std::queue	<queue>	TAD cola

# Operaciones

- Tienen exactamente el mismo nombre que las que hemos visto a lo largo del curso:
  - `push_back()`
  - `push_front()`
  - `operator[]`
  - `begin()`
  - etc.



# Algunas excepciones

- `vector` no implementa `push_front()` o `pop_front()`.
- `list` no implementa `at()` ni el operador `[ ]`.
- No tienen ninguna función `display()`, ni sobrecargan el operador `<<`.



# Ejemplo

```
int main() {
    vector<int> v;
    for (int i = 0; i < 10; i++) {
        v.push_back(i * 3);
    }

    cout << v.size() << endl;
    int suma = 0;
    for (int x : v) {
        suma += x;
    }

    cout << "Suma total: " << suma << endl;
    return 0;
}
```



ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# STL: Iteradores

Manuel Montenegro Montes

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

# Tipos de iteradores

- Iteradores de entrada.
- Iteradores de salida.
- Iteradores hacia delante.
- Iteradores bidireccionales.
- Iteradores de acceso aleatorio.



# Iteradores de entrada

Entrada

`... = *it`

*Acceso*

`it++`

*Avance*

`it1 == it2`

*Comparación*



# Iteradores de salida

Entrada

`... = *it`

*Acceso*

`it1 == it2`

*Comparación*

`it++`

*Avance*

`*it = ...`

*Escritura*

Salida

# Iteradores hacia delante

Entrada

`... = *it`

*Acceso*

`it1 == it2`

*Comparación*

`it++`

*Avance*

`*it = ...`

*Escritura*

Salida

Hacia delante

# Iteradores bidireccionales

`... = *it`

*Acceso*

`it++`

*Avance*

`*it = ...`

*Escritura*

`it1 == it2`

*Comparación*

`it--`

*Retroceso*

Hacia delante

Bidireccionales

# Iteradores de acceso aleatorio

`... = *it`

*Acceso*

`it++`

*Avance*

`*it = ...`

*Escritura*

`it1 == it2`

*Comparación*

Hacia delante

`it--`

*Retroceso*

Bidireccionales

`it = it ± n`

*Avance/retroceso  
por saltos*

Acceso aleatorio

# Tipos de iteradores

- Cada implementación de TAD soporta un tipo de iterador determinado.

Expresión	Tipo de iterador
<code>vector :: begin()</code>	Acceso aleatorio
<code>list :: begin()</code>	Bidireccional
<code>deque :: begin()</code>	Acceso aleatorio
<code>forward_list :: begin()</code>	Hacia delante
<code>ostream_iterator</code>	Salida
<code>istream_iterator</code>	Entrada
<i>Punteros</i>	Acceso aleatorio

# Iterador de salida: ostream\_iterator

- Es un iterador asociado a un flujo de salida (fichero, salida estándar, etc.)
- Cada vez que se modifica el valor apuntado por el iterador, se realiza una operación de salida.
- Cada vez que se incrementa el iterador, no se hace nada.
- Es útil para la función `copy()`

# Ejemplo

```
int main() {  
    std::ostream_iterator<int> it(std::cout, " ");  
  
    *it = 10;  
    it++; // Opcional. No hace nada.  
    *it = 20;  
    it++; // Opcional. No hace nada.  
  
    std::cout << std::endl;  
  
    return 0;  
}
```

Flujo de salida

Separador

std::cout

it

# Ejemplo

```
int main() {  
    std::ostream_iterator<int> it(std::cout, " ");  
  
    *it = 10;  
    it++; // Opcional. No hace nada.  
    *it = 20;  
    it++; // Opcional. No hace nada.  
  
    std::cout << std::endl;  
  
    return 0;  
}
```

Flujo de salida

Separador

std::cout

10\_

it

# Ejemplo

```
int main() {  
    std::ostream_iterator<int> it(std::cout, " ");  
  
    *it = 10;  
    it++; // Opcional. No hace nada.  
    *it = 20;  
    it++; // Opcional. No hace nada.  
  
    std::cout << std::endl;  
  
    return 0;  
}
```

Flujo de salida

Separador

std::cout

10\_20\_

it

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# STL: Algoritmos (1)

Manuel Montenegro Montes

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

# La función copy()

# La función `copy()`

- Definida en `<algorithm>`

*copy(source\_begin, source\_end, destination\_begin)*

donde:

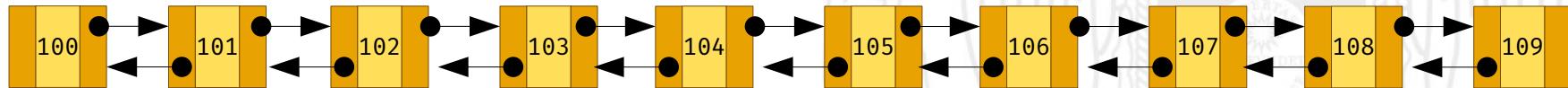
- `source_begin`, `source_end` son iteradores de entrada.
  - `destination_begin` es iterador de salida.
- Copia el intervalo de elementos delimitado por `source_begin` y `source_end` (excluyendo este último), a la posición apuntada por el iterador `destination_begin`.

# Ejemplo

```
int main() {
    vector<int> origen;
    list<int> destino;

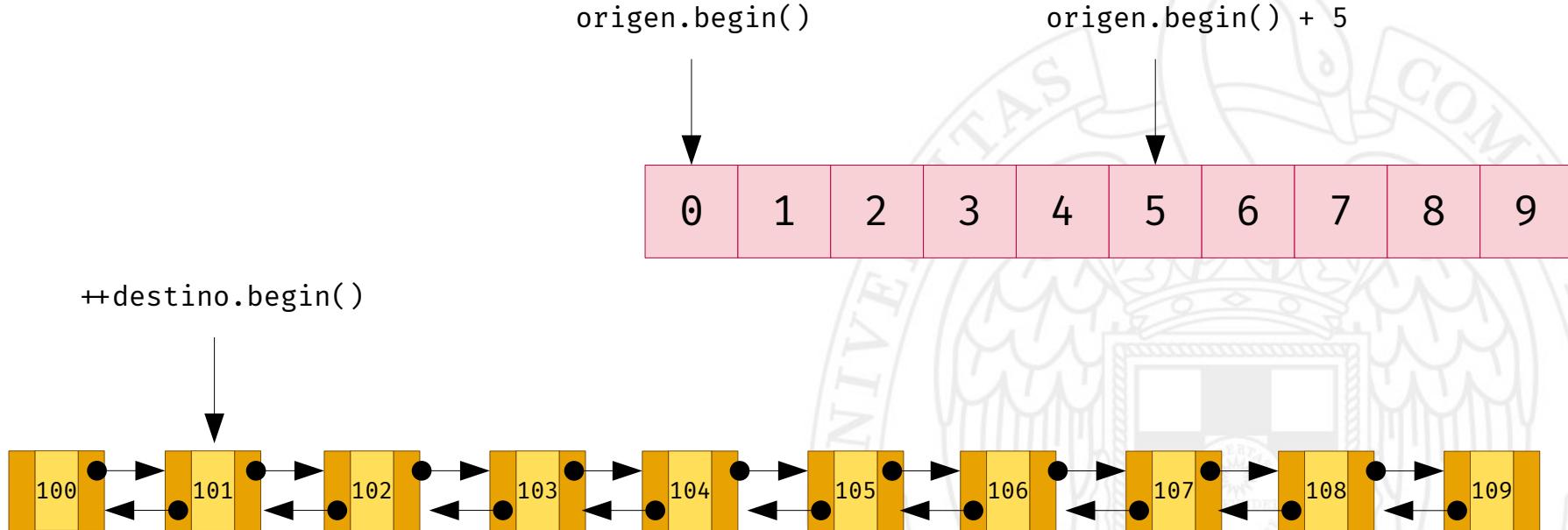
    for (int i = 0; i < 10; i++) {
        origen.push_back(i);
        destino.push_back(100 + i);
    }
    ...
}
```

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---



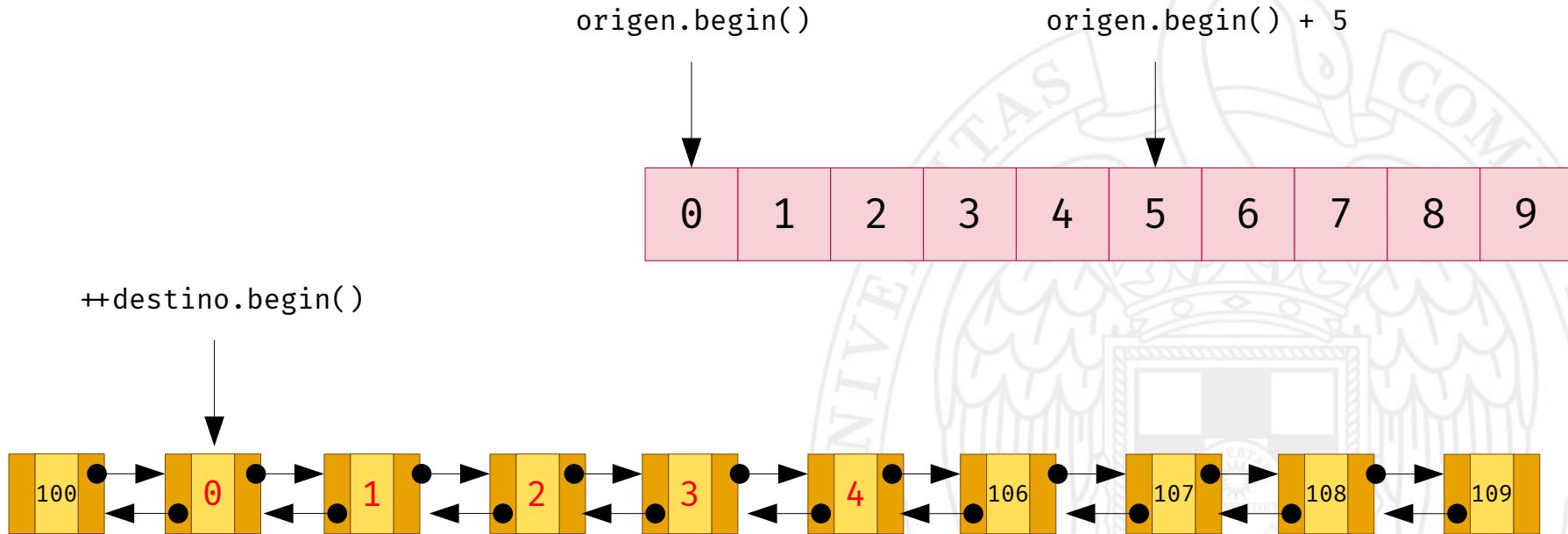
# Ejemplo

```
copy(origen.begin(), origen.begin() + 5, ++destino.begin());
```



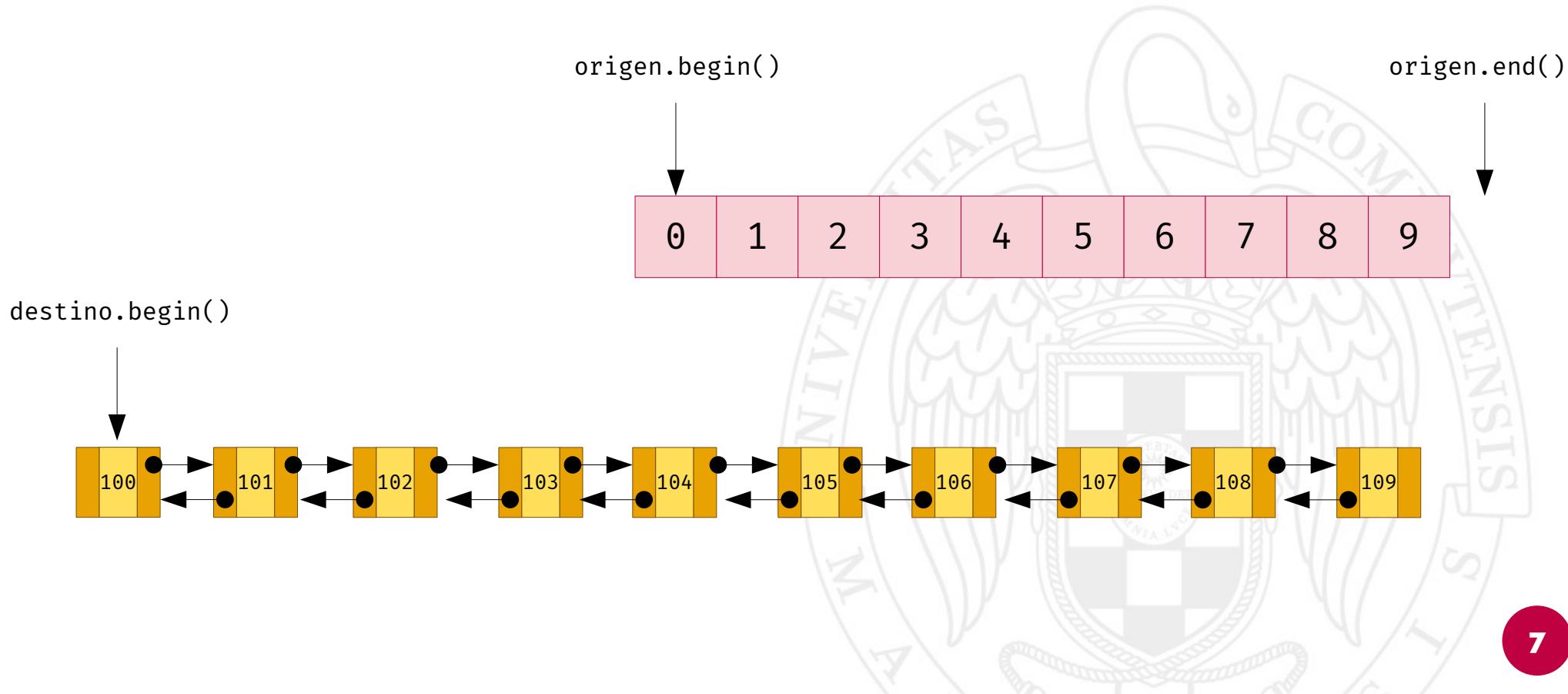
# Ejemplo

```
copy(origen.begin(), origen.begin() + 5, ++destino.begin());
```



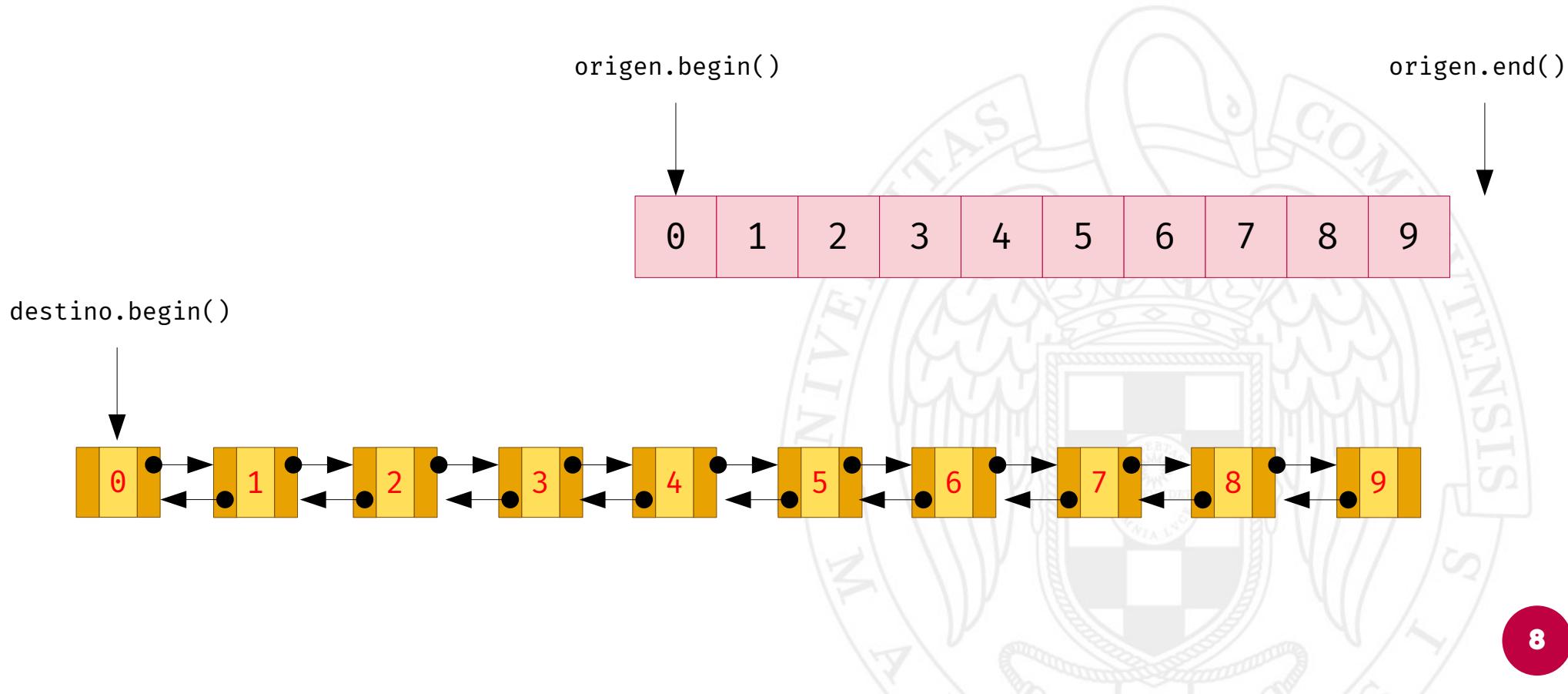
# Ejemplo

```
copy(origen.begin(), origen.end(), destino.begin());
```



# Ejemplo

```
copy(origen.begin(), origen.end(), destino.begin());
```

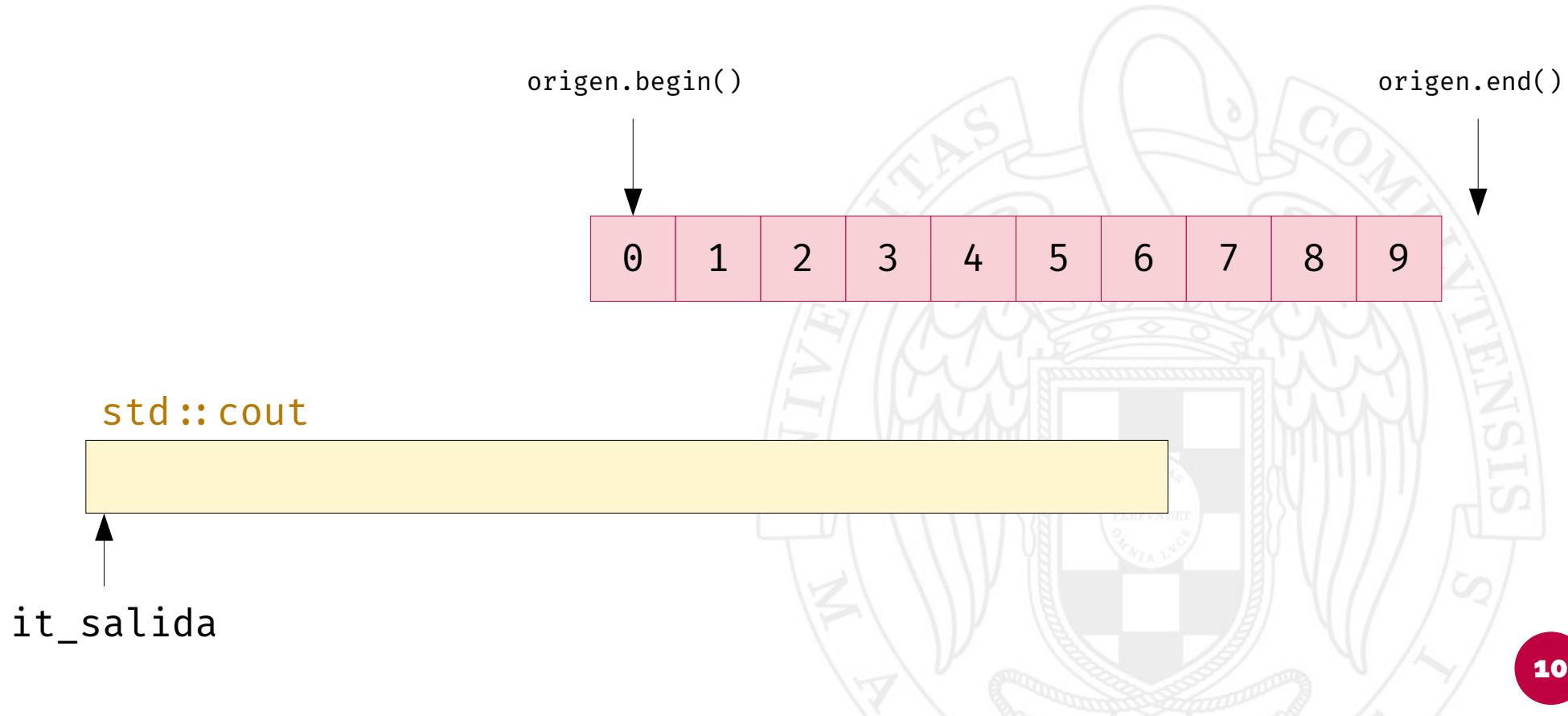


# Utilidad de función `copy()`

- Se puede utilizar para multitud de casos:
  - De un `vector` a un `list` y viceversa.
  - De `vector` a `vector`.
  - De `list` a `deque`.
  - De un `array` a `vector` y viceversa.
  - De un `array` a `list` y viceversa.
  - De un `vector/list/array` a un `ostream_iterator`.

# Otro ejemplo

```
ostream_iterator<int> it_salida(cout, " ");
copy(origen.begin(), origen.end(), it_salida);
```

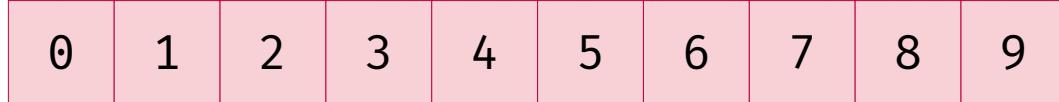


# Otro ejemplo

```
ostream_iterator<int> it_salida(cout, " ");
copy(origen.begin(), origen.end(), it_salida);
```

origen.begin()

origen.end()



std::cout

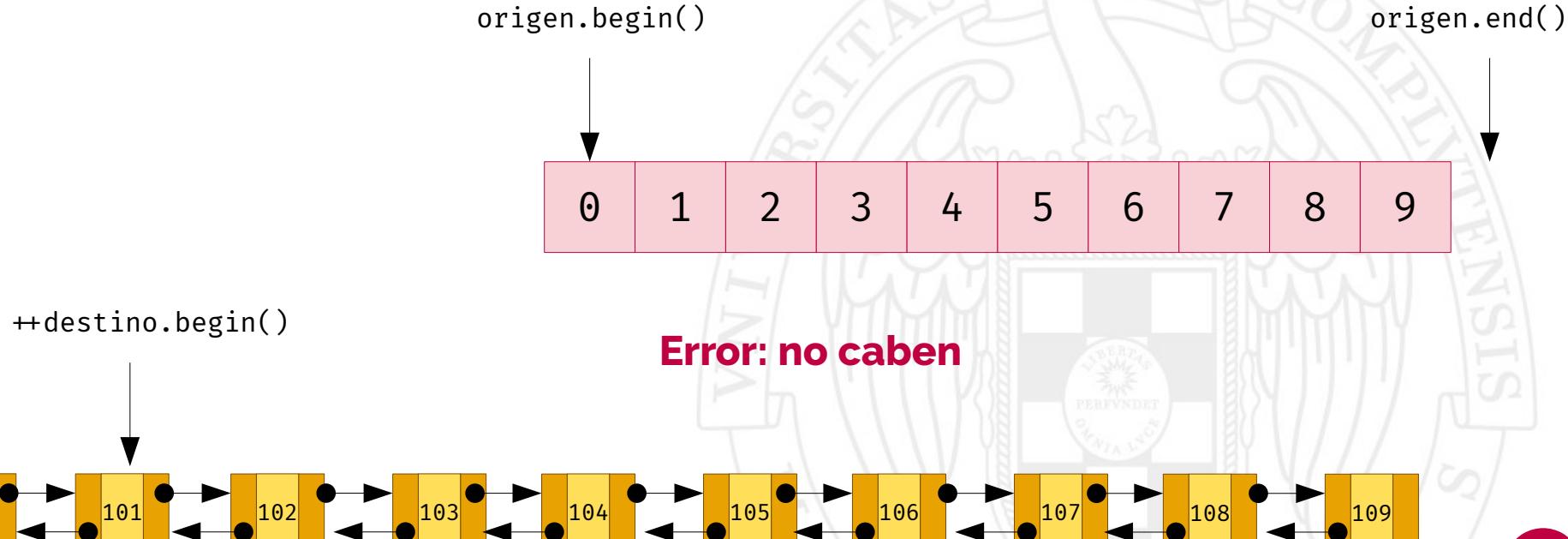
0\_1\_2\_3\_4\_5\_6\_7\_8\_9\_

it\_salida

# Cuidado!

- Para que la copia tenga éxito, el iterador de destino debe poderse incrementar tantas veces como elementos deseen copiarse.

```
copy(origen.begin(), origen.end(), +destino.begin());
```



# Los iteradores back\_insert\_iterator

- Son iteradores de salida que van asociados a un contenedor secuencial (list, vector, deque, etc).
- Cuando se escribe en el iterador, se añade un elemento al contenedor.
- Cuando se incrementa el iterador, no se hace nada.

# Ejemplo

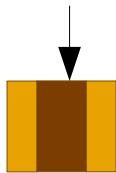
```
int main() {
    vector<int> origen;
    list<int> lista_destino;

    // inicializar origen
    ...
    // suponemos que lista_destino queda vacía

    back_insert_iterator<list<int>> it_dest(lista_destino);
    copy(origen.begin(), origen.end(), it_dest);

    imprimir(cout, lista_destino);
}
```

it\_dest



0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---

# Ejemplo

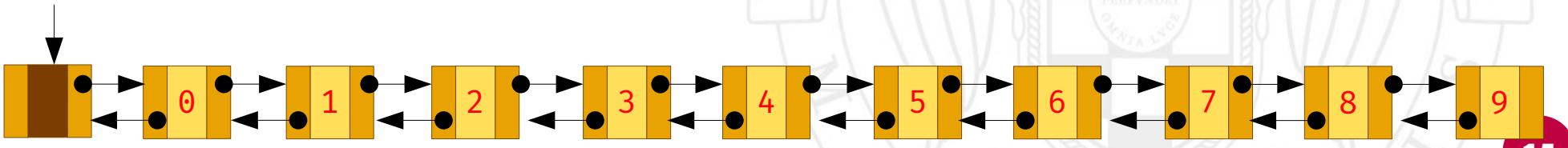
```
int main() {
    vector<int> origen;
    list<int> lista_destino;

    // inicializar origen
    ...
    // suponemos que lista_destino queda vacía

    back_insert_iterator<list<int>> it_dest(lista_destino);
    copy(origen.begin(), origen.end(), it_dest);

    imprimir(cout, lista_destino);
}
```

it\_dest



# La función sort()

# La función sort()

- También definida en <algorithm>.

`sort(begin, end)`

donde:

- `begin`, `end` son iteradores con acceso aleatorio.
- Ordena ascendente los elementos contenidos entre los iteradores `begin` y `end` (excluyendo este último).
- Utiliza el operador `<` para comparar los elementos.

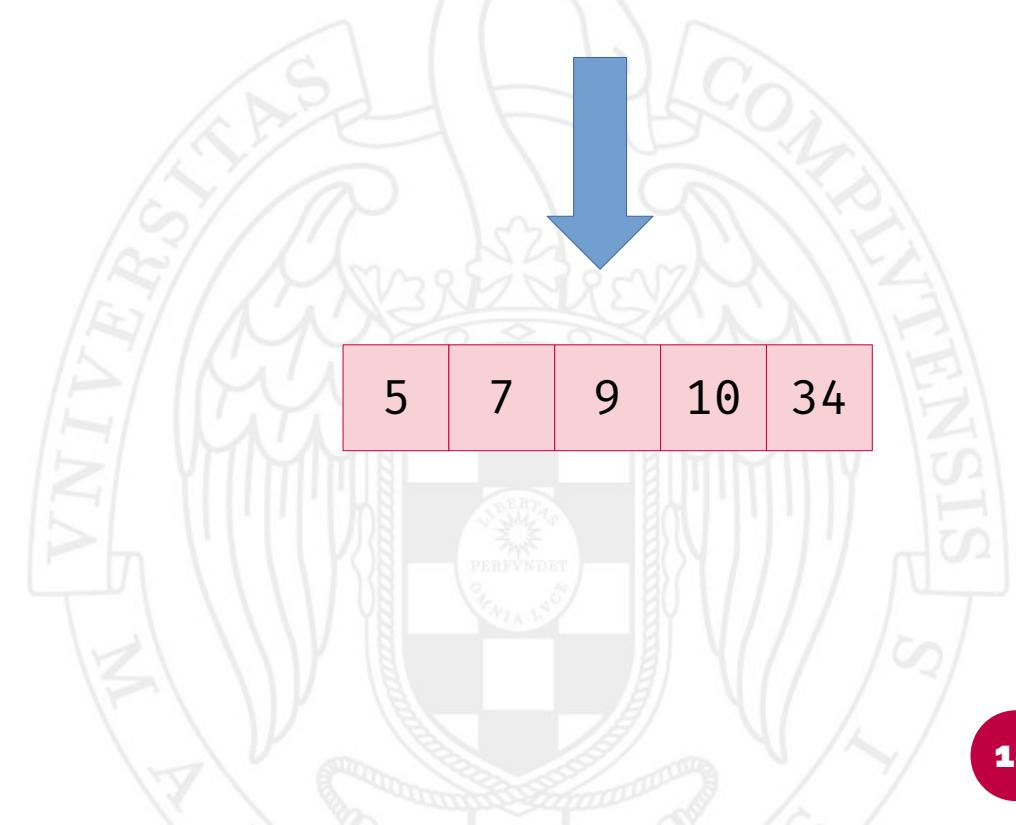
# Ejemplo

```
int main() {  
    vector<int> v;  
    v.push_back(10);  
    v.push_back(34);  
    v.push_back(5);  
    v.push_back(7);  
    v.push_back(9);  
  
    sort(v.begin(), v.end());  
}
```

10	34	5	7	9
----	----	---	---	---



5	7	9	10	34
---	---	---	----	----



# Otro ejemplo

```
int main() {  
    int elems[] = {14, 5, 1, 20, 4, 7};  
    sort(elems, elems + 6);  
}
```

14	5	1	20	4	7
----	---	---	----	---	---



1	4	5	7	14	20
---	---	---	---	----	----

# Más funciones en <algorithm>

- `find(begin, end, value)`
- `fill(begin, end, value)`
- `unique(begin, end)`
- `binary_search(begin, end, value)`
- `max(begin, end)`

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Punteros inteligentes

Manuel Montenegro Montes

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

# ¿Qué es un puntero inteligente?

- Es un TAD que permite las mismas operaciones que un puntero, pero añadiendo nuevas características.
- En particular se encarga de liberar automáticamente el objeto apuntado por él, sin que tengamos que hacerlo nosotros mediante `delete`.
- Las librerías de C++ definen dos tipos de punteros inteligentes en el fichero de cabecera `<memory>`:
  - `std :: unique_ptr<T>` - Puntero exclusivo a un dato de tipo T.  
No puede haber otros punteros apuntando al mismo dato.
  - `std :: shared_ptr<T>` - Puntero compartido a un dato de tipo T.  
Se permiten otros punteros apuntando al mismo dato.

# Recordatorio: clase Fecha

```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo);  
    Fecha(int anyo);  
    Fecha();  
  
    int get_dia() const;  
    void set_dia(int dia);  
    int get_mes() const;  
    void set_mes(int mes);  
    int get_anyo() const;  
    void set_anyo(int anyo);  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};  
  
std::ostream & operator<<(std::ostream &out, const Fecha &f);
```



# Punteros exclusivos – std :: unique\_ptr

# Puntero normal vs unique\_ptr

- Ejemplo: crear un objeto en el heap mediante un puntero normal:

```
new Fecha(25, 12, 2019)
```

Esto devuelve un valor de tipo `Fecha *`.

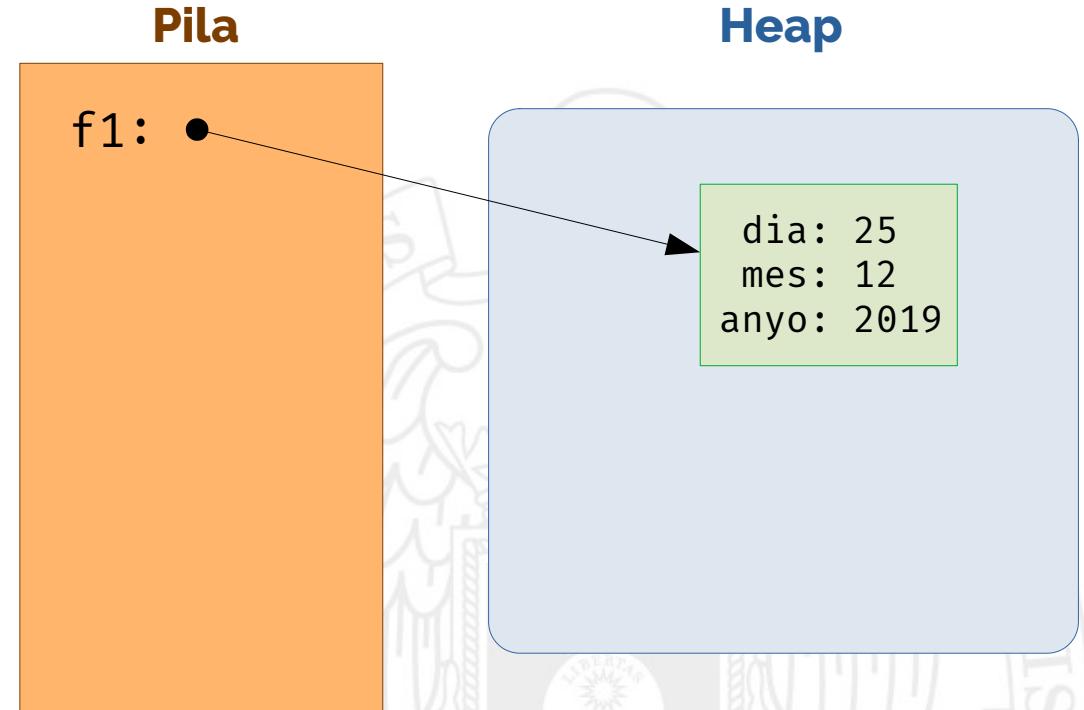
- Ejemplo: crear un objeto en el heap mediante un puntero exclusivo:

```
std::make_unique<Fecha>(25, 12, 2019)
```

Esto devuelve un objeto de tipo `std::unique_ptr<Fecha>`.

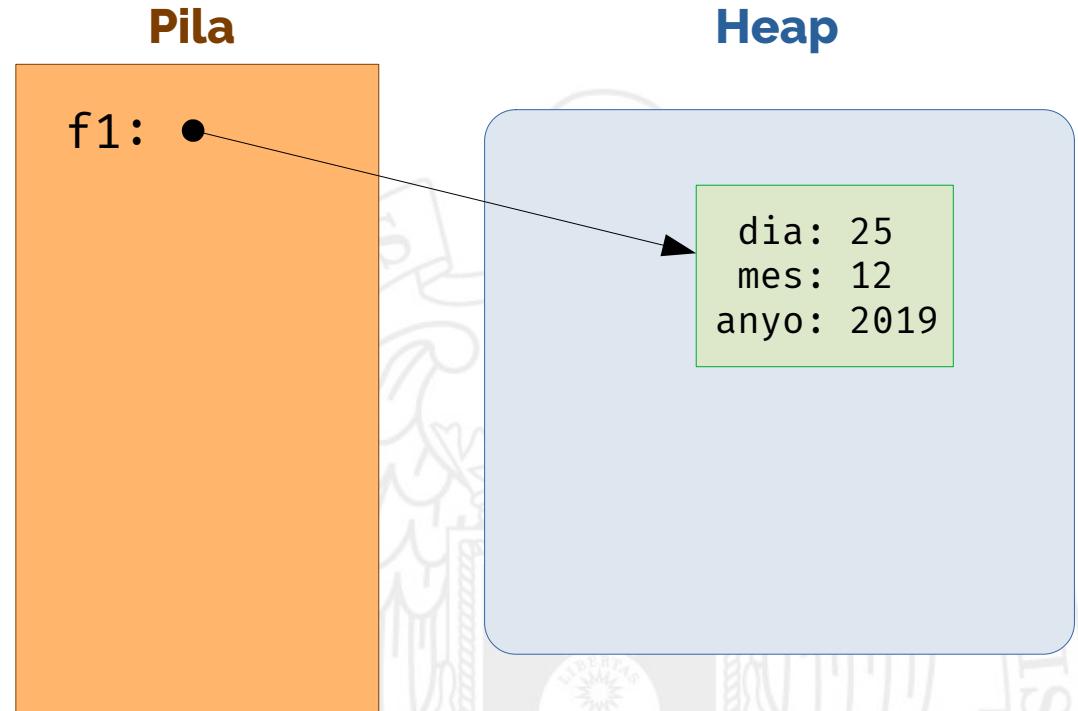
# Ejemplo

```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);
```



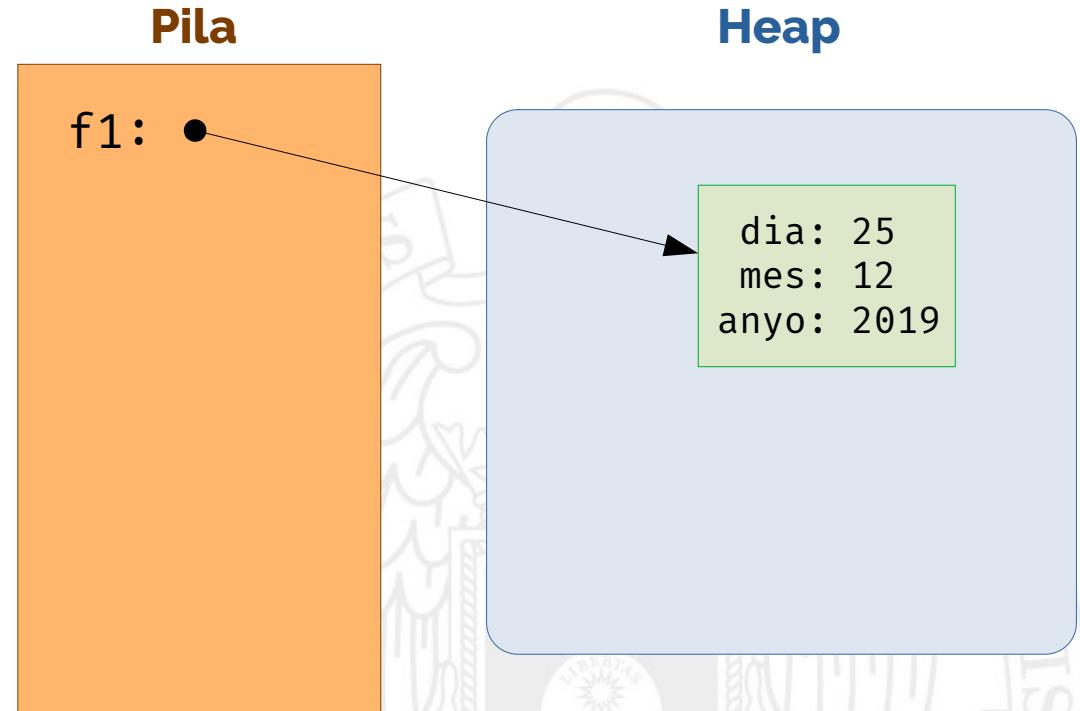
# Ejemplo

```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);  
  
if (f1 != nullptr) {  
    std::cout << f1->get_anyo() << std::endl;  
    std::cout << *f1 << std::endl;  
}
```



# Un unique\_ptr no puede ser copiado

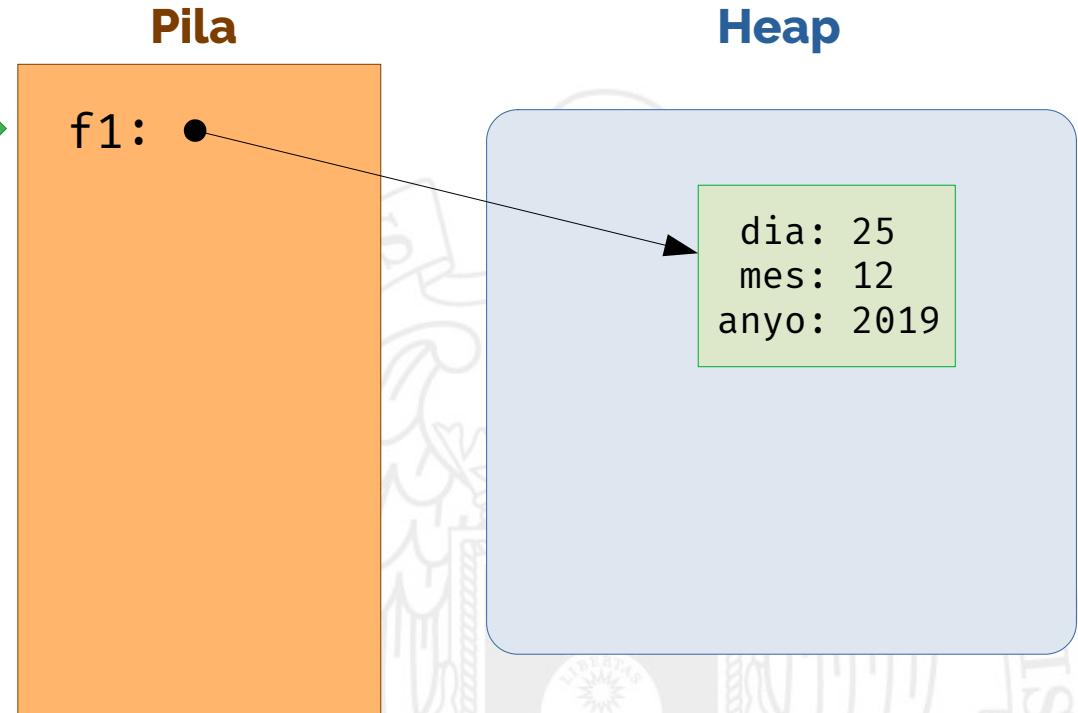
```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);  
  
std::unique_ptr<Fecha> f2 = f1; 
```



# Un unique\_ptr puede ser transferido

```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);
```

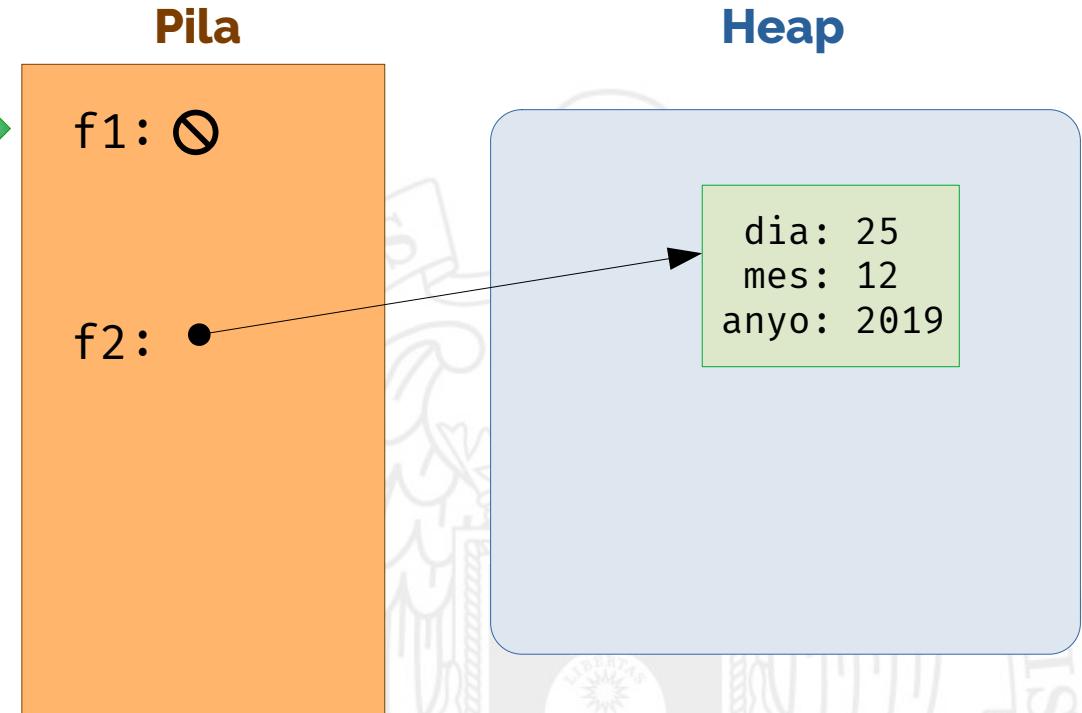
```
std::unique_ptr<Fecha> f2 = std::move(f1); ✓
```



# Un unique\_ptr puede ser transferido

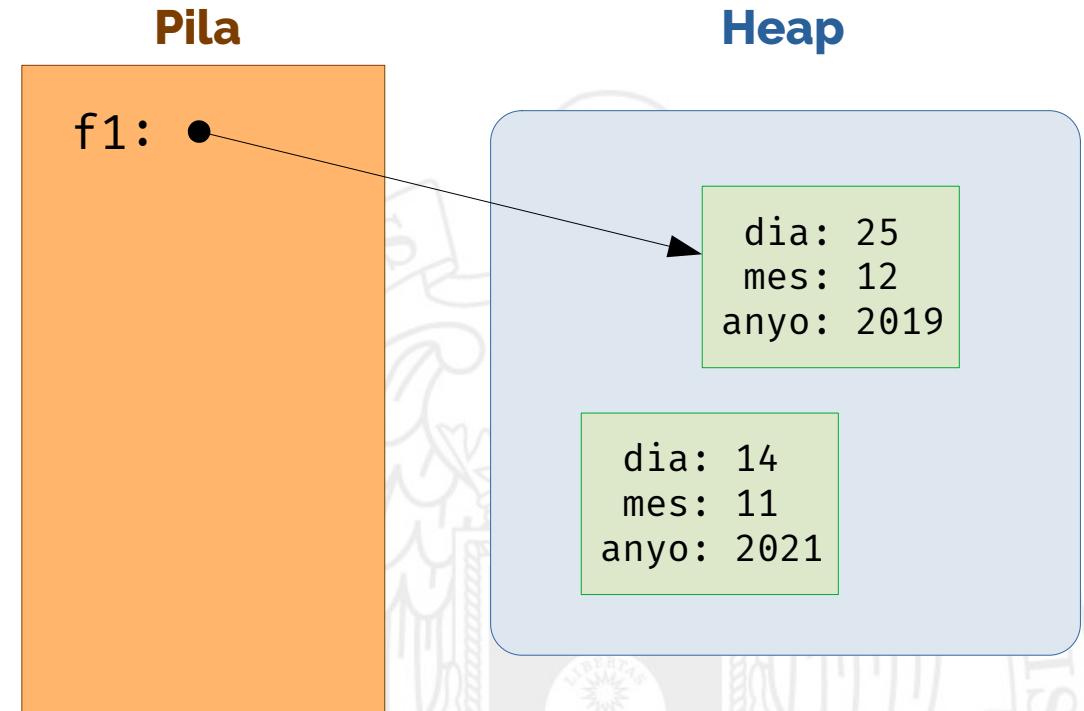
```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);
```

```
std::unique_ptr<Fecha> f2 = std::move(f1); ✓
```



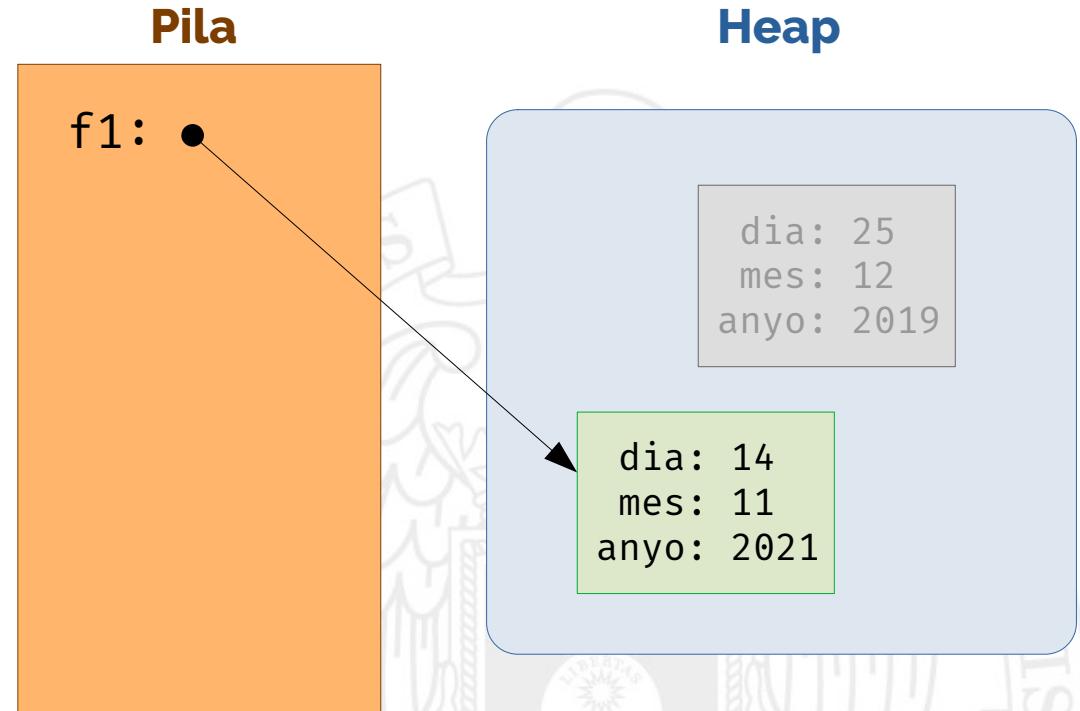
# Reasignando un unique\_ptr

```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);  
  
f1 = std::make_unique<Fecha>(14, 11, 2021);
```



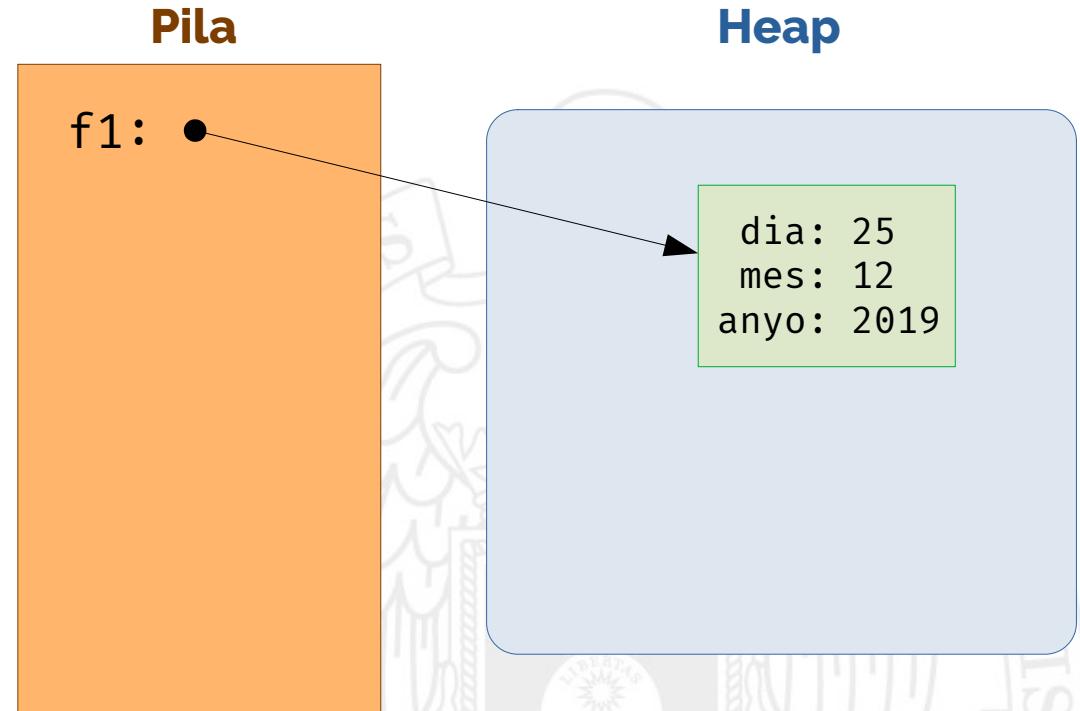
# Reasignando un unique\_ptr

```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);  
  
f1 = std::make_unique<Fecha>(14, 11, 2021);
```



# Reasignando un unique\_ptr

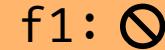
```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);  
  
f1 = nullptr;
```



# Reasignando un unique\_ptr

```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);  
  
f1 = nullptr;
```

Pila

f1: 

Heap

dia: 25  
mes: 12  
anyo: 2019

# Punteros compartidos – std :: shared\_ptr

# Crear un `shared_ptr`

- Para crear un objeto en el heap mediante un puntero compartido:

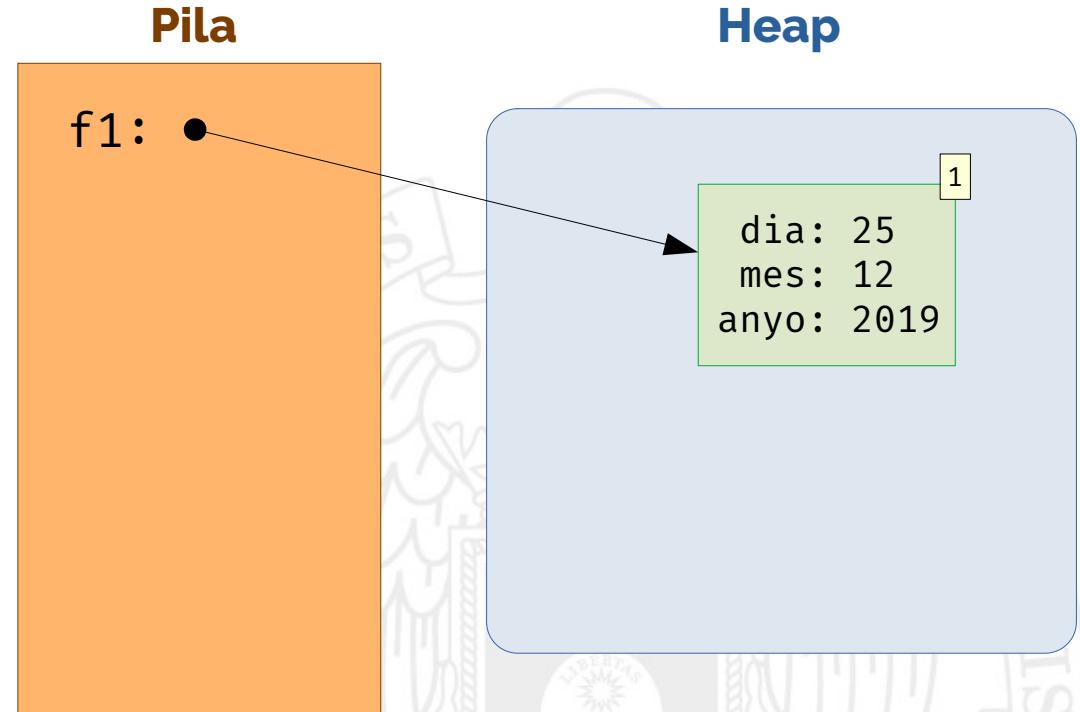
```
std :: make_shared<Fecha>(25, 12, 2019)
```

Esto devuelve un objeto de tipo `std :: shared_ptr<Fecha>`.

- Los objetos del *heap* apuntados por un puntero compartido llevan un **contador de referencias** que indica el número de punteros compartidos que apuntan hacia él.
  - Cuando este contador llega a 0, el objeto se libera.

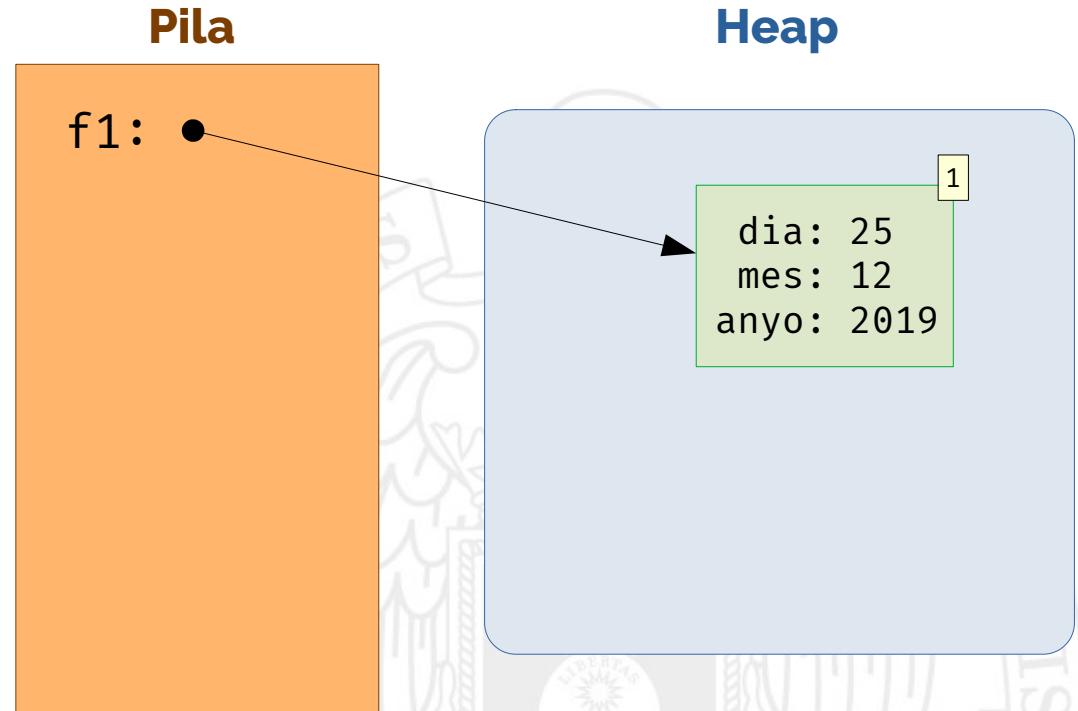
# Ejemplo

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);
```



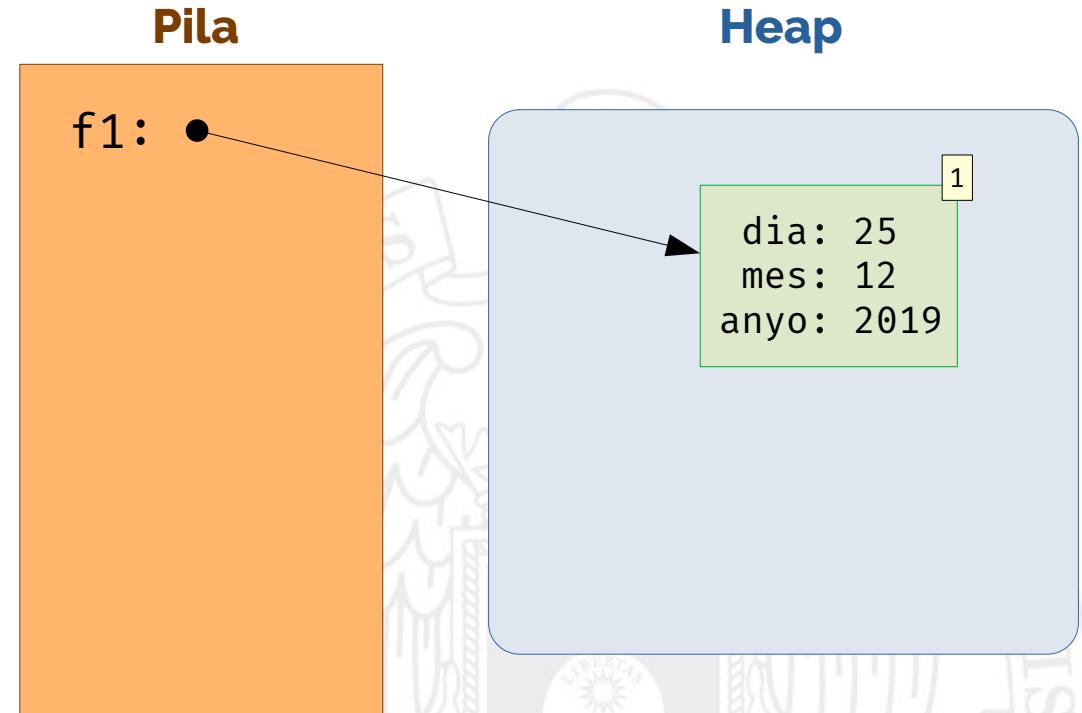
# Ejemplo

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
if (f1 != nullptr) {  
    std::cout << f1->get_anyo() << std::endl;  
    std::cout << *f1 << std::endl;  
}
```



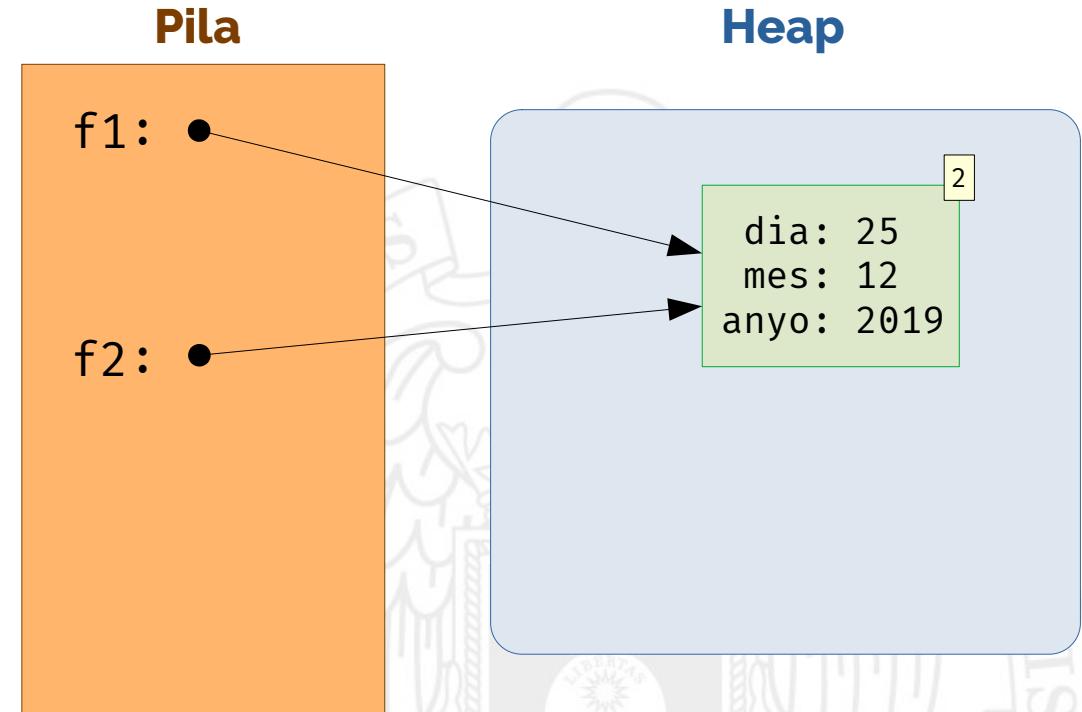
# Copia de un shared\_ptr

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
std::shared_ptr<Fecha> f2 = f1;
```



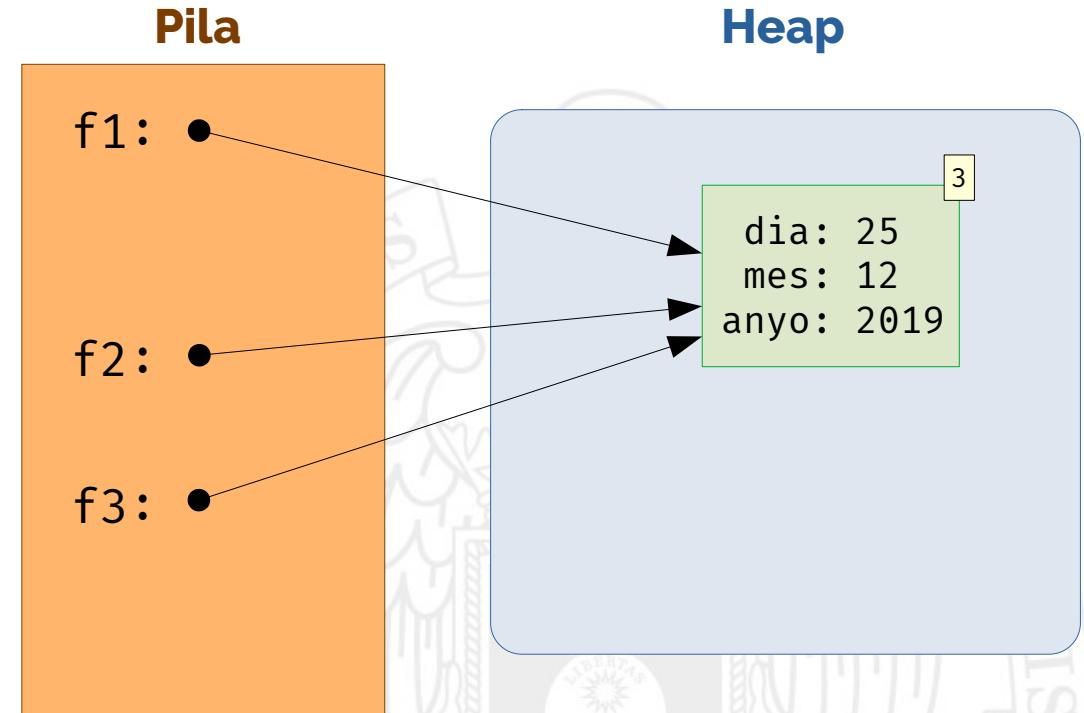
# Copia de un shared\_ptr

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
std::shared_ptr<Fecha> f2 = f1;
```



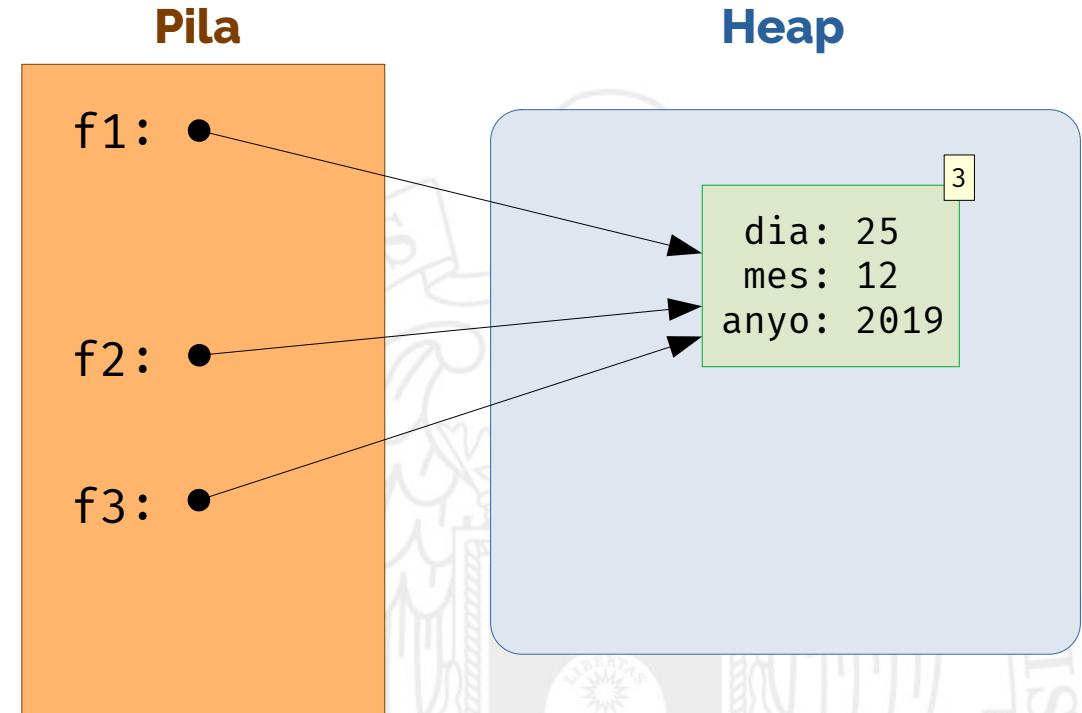
# Copia de un shared\_ptr

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
std::shared_ptr<Fecha> f2 = f1;  
std::shared_ptr<Fecha> f3 = f2;
```



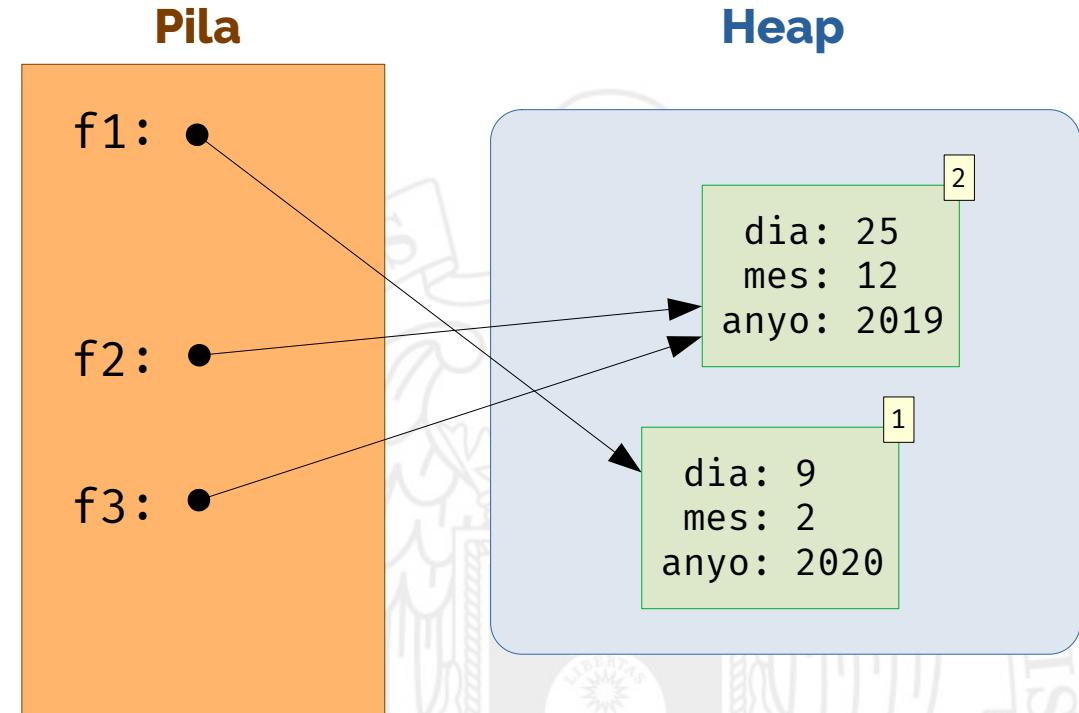
# Copia de un shared\_ptr

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
std::shared_ptr<Fecha> f2 = f1;  
std::shared_ptr<Fecha> f3 = f2;  
  
f1 = std::make_shared<Fecha>(9, 2, 2020);
```



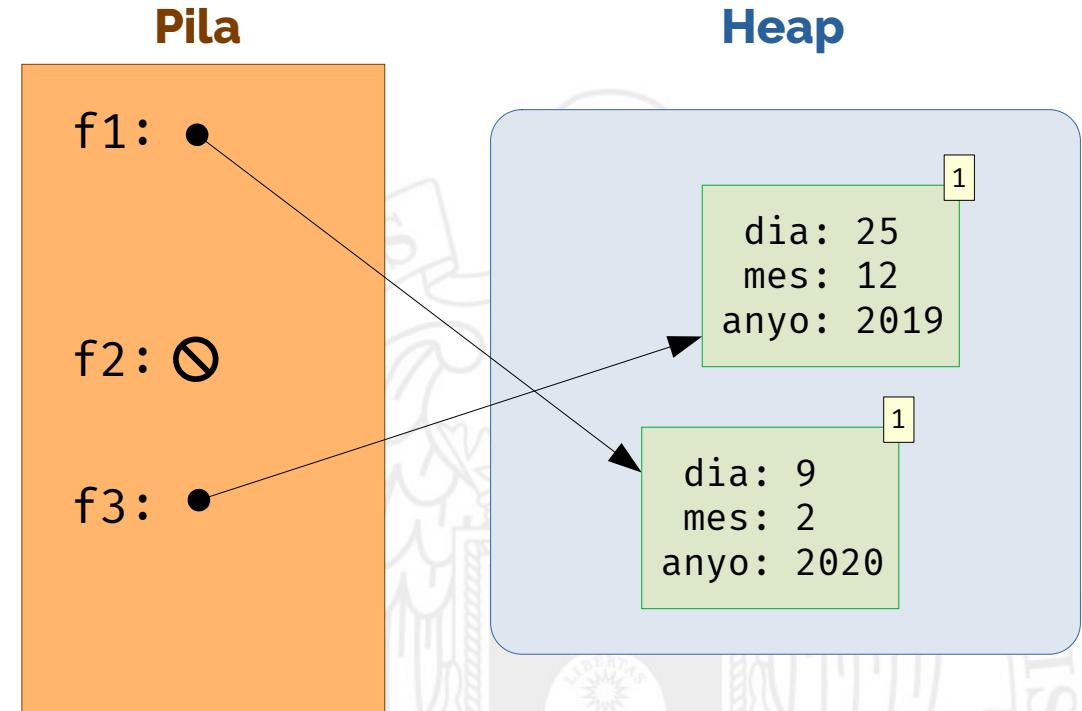
# Copia de un shared\_ptr

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
std::shared_ptr<Fecha> f2 = f1;  
std::shared_ptr<Fecha> f3 = f2;  
  
f1 = std::make_shared<Fecha>(9, 2, 2020);
```



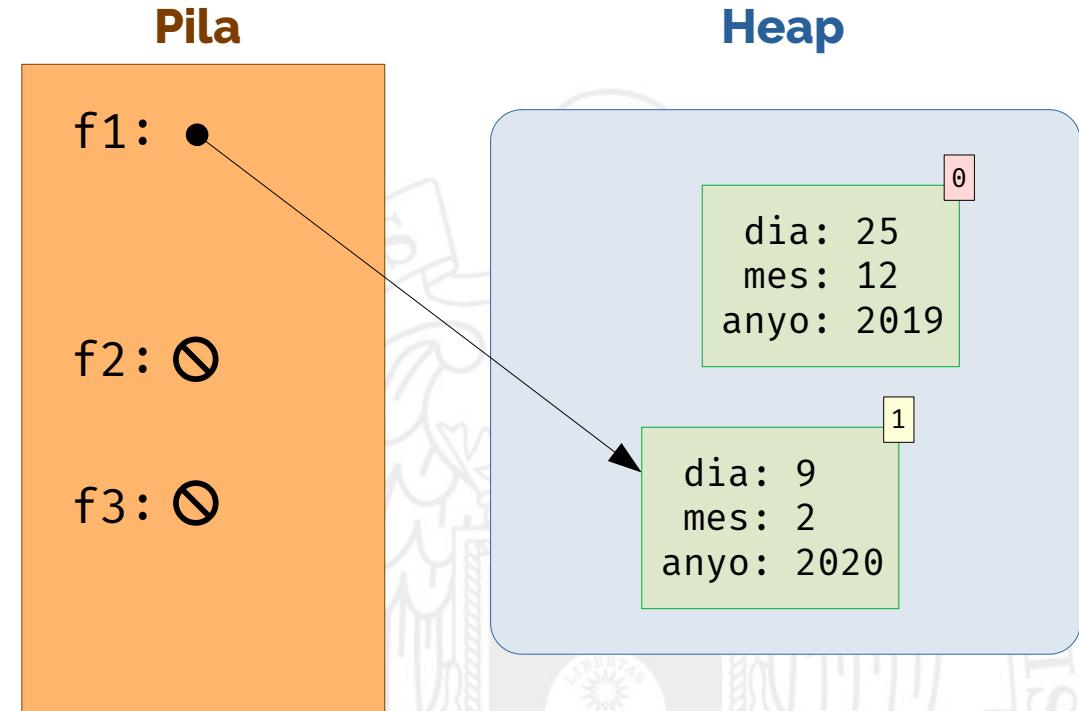
# Copia de un shared\_ptr

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
std::shared_ptr<Fecha> f2 = f1;  
std::shared_ptr<Fecha> f3 = f2;  
  
f1 = std::make_shared<Fecha>(9, 2, 2020);  
f2 = nullptr;
```



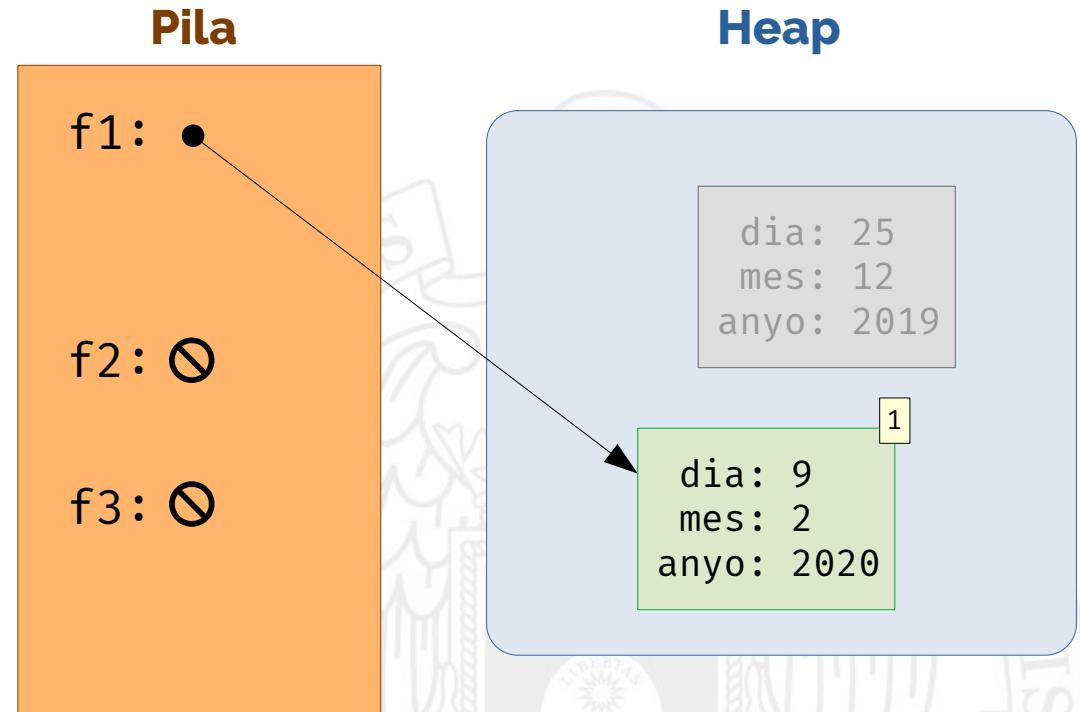
# Copia de un shared\_ptr

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
std::shared_ptr<Fecha> f2 = f1;  
std::shared_ptr<Fecha> f3 = f2;  
  
f1 = std::make_shared<Fecha>(9, 2, 2020);  
f2 = nullptr;  
f3 = nullptr;
```

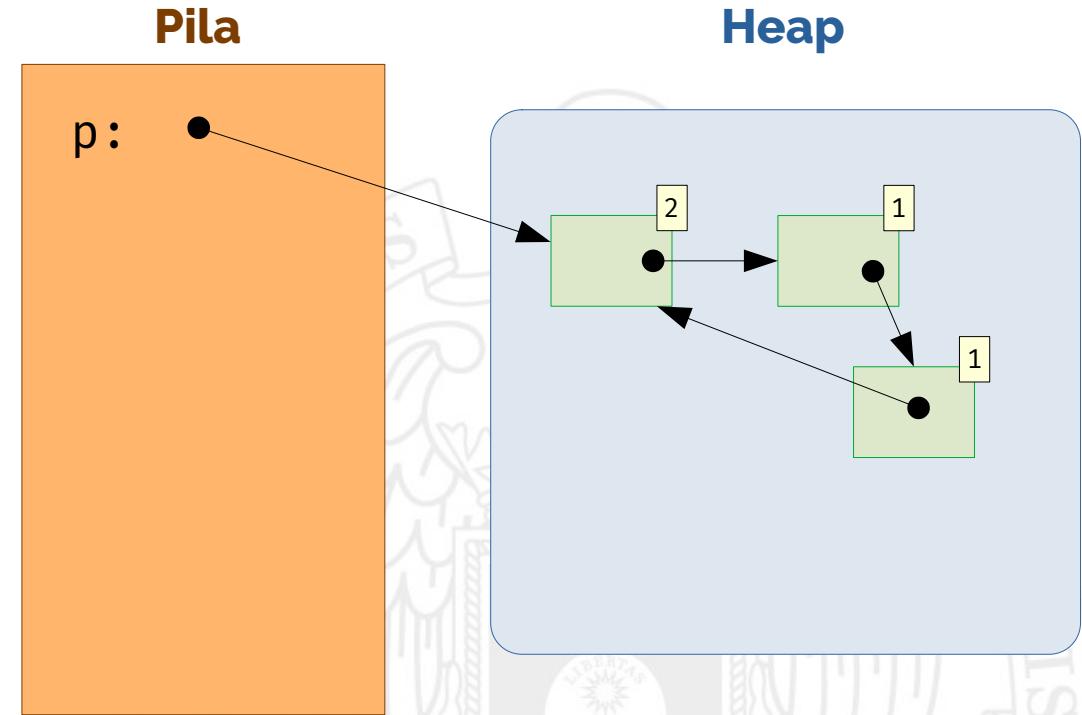


# Copia de un shared\_ptr

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
std::shared_ptr<Fecha> f2 = f1;  
std::shared_ptr<Fecha> f3 = f2;  
  
f1 = std::make_shared<Fecha>(9, 2, 2020);  
f2 = nullptr;  
f3 = nullptr;
```



# ¡Cuidado con las referencias circulares!



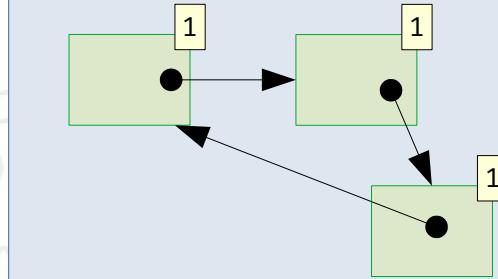
# ¡Cuidado con las referencias circulares!

```
p = nullptr;
```

Pila

p:  $\text{\textcircled{0}}$

Heap



ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Funciones de orden superior

Manuel Montenegro Montes

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

# Ejercicio

- Función que recibe una lista de enteros y elimina los números pares de la misma.

```
bool es_par(int x) { return x % 2 == 0; }

void eliminar_pares(std::list<int> &elems) {
    auto it = elems.begin();
    while (it != elems.end()) {
        if (es_par(*it)) {
            it = elems.erase(it);
        } else {
            ++it;
        }
    }
}
```

# Ejemplo

```
std::list<int> v1 = {1, 5, 6, 9, 10, 20};  
eliminar_pares(v1);  
std::cout << v1 << std::endl;
```

[1, 5, 9]

# Ejercicio

- Función que recibe una lista de enteros y elimina los números **impares** de la misma.

```
bool es_impar(int x) { return x % 2 == 1; }

void eliminar_impares(std::list<int> &elems) {
    auto it = elems.begin();
    while (it != elems.end()) {
        if (es_impar(*it)) {
            it = elems.erase(it);
        } else {
            ++it;
        }
    }
}
```

# Ejemplo

```
std::list<int> v1 = {1, 5, 6, 9, 10, 20};  
std::list<int> v2 = v1;  
eliminar_pares(v1);  
std::cout << v1 << std::endl;  
  
eliminar_impares(v2);  
std::cout << v2 << std::endl;
```

[1, 5, 9]

[6, 10, 20]

# Ejercicio

- Función que recibe una lista de enteros y elimina los números **positivos** de la misma.

```
bool es_positivo(int x) { return x > 0; }

void eliminar_positivos(std::list<int> &elems) {
    auto it = elems.begin();
    while (it != elems.end()) {
        if (es_positivo(*it)) {
            it = elems.erase(it);
        } else {
            ++it;
        }
    }
}
```

# Ejemplo

```
std::list<int> v1 = {1, 5, 6, 9, 10, 20};  
std::list<int> v2 = v1;  
eliminar_pares(v1);  
std::cout << v1 << std::endl;  
  
eliminar_impares(v2);  
std::cout << v2 << std::endl;  
  
std::list<int> v3 = {-2, 3, 10, -6, 20};  
eliminar_positivos(v3);  
std::cout << v3 << std::endl;
```

[1, 5, 9]

[6, 10, 20]

[-2, -6]

# ¡Cuánta duplicación!

```
void eliminar_positivos(std::list<int> &elems) {
    auto it = elems.begin();
    while (it != elems.end()) {
        if (es_positivo(*it)) {
            it = elems.erase(it);
        } else {
            ++it;
        }
    }
}
```

```
void eliminar_pares(std::list<int> &elems) {
    auto it = elems.begin();
    while (it != elems.end()) {
        if (es_par(*it)) {
            it = elems.erase(it);
        } else {
            ++it;
        }
    }
}
```

```
void eliminar_impar(std::list<int> &elems) {
    auto it = elems.begin();
    while (it != elems.end()) {
        if (es_impar(*it)) {
            it = elems.erase(it);
        } else {
            ++it;
        }
    }
}
```

- La solución para unificar estas tres funciones es **parametrizarlas** en aquello en lo que se diferencian.
- ¡Pero aquí se diferencian en una **función**!
- ¿Es posible pasar funciones como parámetros en C++?

# Sí, es posible, pero...

¿Qué tipo tiene ese parámetro?

```
void eliminar_positivos(std::list<int> &elems) {  
    auto it = elems.begin();  
    while (it != elems.end()) {  
        if (es_positivo(*it)) {  
            it = elems.erase(it);  
        } else {  
            ++it;  
        }  
    }  
}
```



```
void eliminar(std::list<int> &elems, ??? func) {  
    auto it = elems.begin();  
    while (it != elems.end()) {  
        if (func(*it)) {  
            it = elems.erase(it);  
        } else {  
            ++it;  
        }  
    }  
}
```

# Sí, es posible, pero...

¿Qué tipo tiene ese parámetro?

- **Puntero a función**
  - Mecanismo heredado de C.
- **Variable plantilla**
  - Utiliza el mecanismo de plantillas de C++.
  - Dejamos que el compilador infiera el tipo.
  - Compatible con objetos función.

**Siguiente video**

# Uso de variable de plantilla

```
template <typename T>
void eliminar(std::list<int> &elems, T func) {
    auto it = elems.begin();
    while (it != elems.end()) {
        if (func(*it)) {
            it = elems.erase(it);
        } else {
            ++it;
        }
    }
}
```

# Ejemplo

```
bool es_par(int x) { return x % 2 == 0; }
bool es_impar(int x) { return x % 2 == 1; }
bool es_positivo(int x) { return x > 0; }

std::list<int> v1 = {1, 5, 6, 9, 10, 20};
std::list<int> v2 = v1;
eliminar_pares(v1);
std::cout << v1 << std::endl;

eliminar_impares(v2);
std::cout << v2 << std::endl;

std::list<int> v3 = {-2, 3, 10, -6, 20};
eliminar_positivos(v3);
std::cout << v3 << std::endl;
```



# Ejemplo

```
bool es_par(int x) { return x % 2 == 0; }
bool es_impar(int x) { return x % 2 == 1; }
bool es_positivo(int x) { return x > 0; }
```

```
std::list<int> v1 = {1, 5, 6, 9, 10, 20};
std::list<int> v2 = v1;
eliminar_pares(v1); → eliminar(v1, es_par);
std::cout << v1 << std::endl;

eliminar_impares(v2); → eliminar(v2, es_impar);
std::cout << v2 << std::endl;

std::list<int> v3 = {-2, 3, 10, -6, 20};
eliminar_positivos(v3); → eliminar(v3, es_positivo);
std::cout << v3 << std::endl;
```

# Ejemplo

```
bool es_par(int x) { return x % 2 == 0; }
bool es_impar(int x) { return x % 2 == 1; }
bool es_positivo(int x) { return x > 0; }

std::list<int> v1 = {1, 5, 6, 9, 10, 20};
std::list<int> v2 = v1;
eliminar(v1, es_par);
std::cout << v1 << std::endl;

eliminar(v2, es_impar);
std::cout << v2 << std::endl;

std::list<int> v3 = {-2, 3, 10, -6, 20};
eliminar(v3, es_positivo);
std::cout << v3 << std::endl;
```

[1, 5, 9]

[6, 10, 20]

[-2, -6]

# Orden superior

- Cuando una función o método  $f$  recibe otras funciones como parámetros, o devuelve una función como resultado, decimos que  $f$  es una función o método de **orden superior**.
- La función `eliminar` es de orden superior.



# Una pequeña generalización

- Podemos hacer que eliminar funcione sobre listas de cualquier tipo; no solo sobre listas de `int`.

```
template <typename T, typename U>
void eliminar(std::list<U> &elems, T func) {
    auto it = elems.begin();
    while (it != elems.end()) {
        if (func(*it)) {
            it = elems.erase(it);
        } else {
            ++it;
        }
    }
}
```

# Ejemplo

```
std::list<Fecha> v4 = { {25, 12, 2010}, {10, 21, 2020}, {25, 12, 1900}, {1, 1, 2000} };  
eliminar(v4, es_navidad);  
std::cout << v4 << std::endl;
```

[10/21/2020, 01/01/2000]



ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Los tipos pair y tuple

Manuel Montenegro Montes

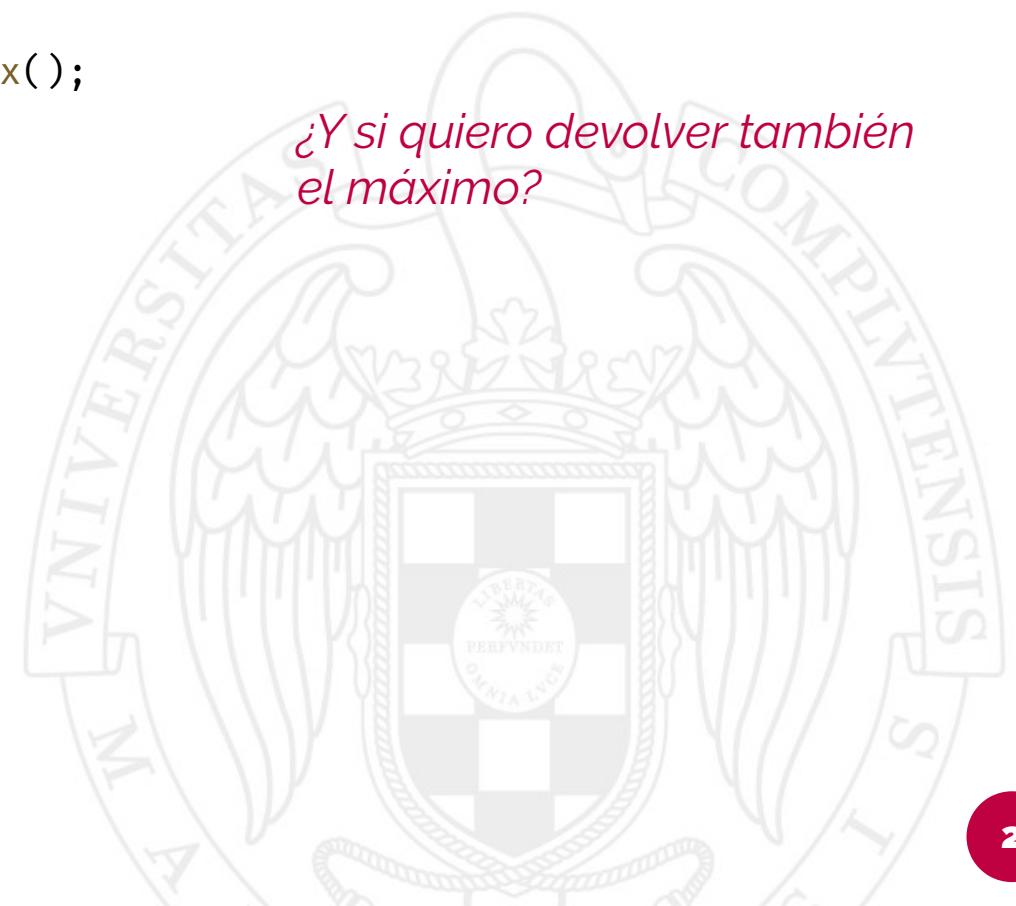
Departamento de Sistemas Informáticos y Computación  
Facultad de Informática – Universidad Complutense de Madrid

# Ejemplo

- Calcular el elemento mínimo de un array.

```
int min(int *array, int longitud) {  
    int min = std::numeric_limits<int>::max();  
  
    for (int i = 0; i < longitud; i++) {  
        min = std::min(min, array[i]);  
    }  
  
    return min;  
}
```

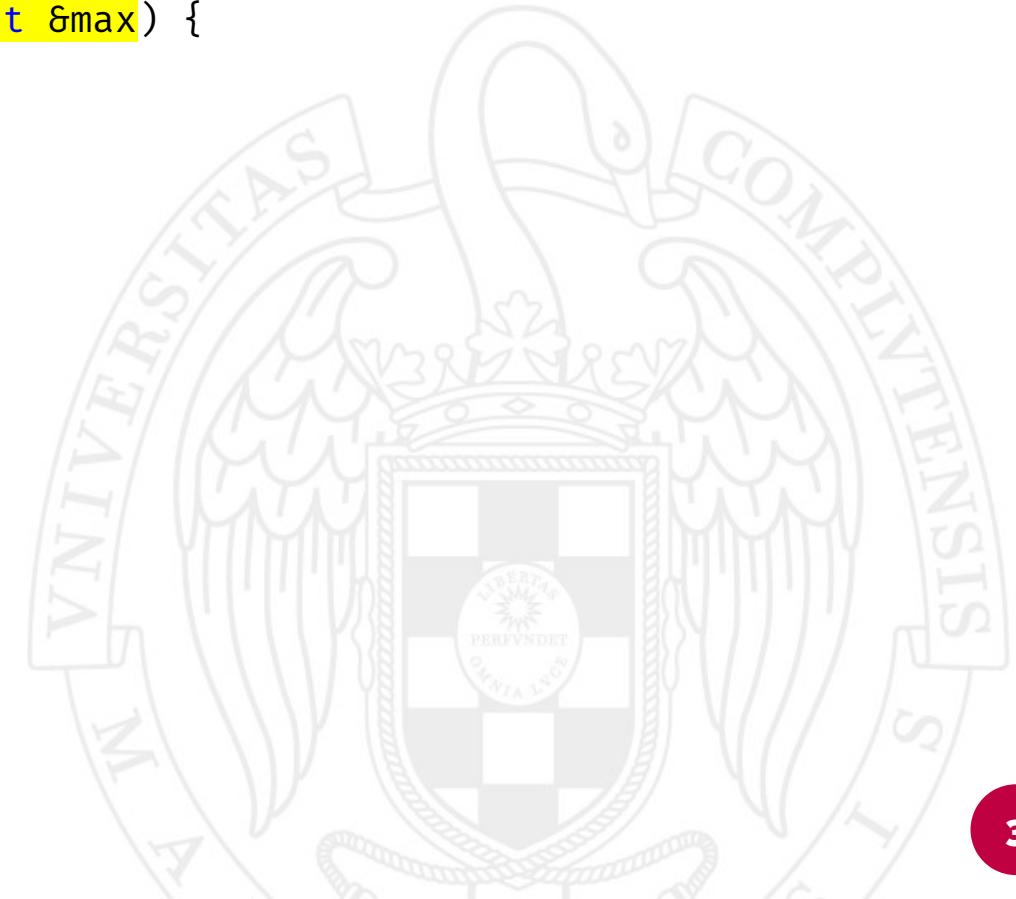
*¿Y si quiero devolver también el máximo?*



# Ejemplo

- Calcular el elemento mínimo y máximo de un array.

```
int min_max(int *array, int longitud, int &max) {  
}  
}
```



# Ejemplo

- Calcular el elemento mínimo y máximo de un array.

```
void min_max(int *array, int longitud, int &min, int &max) {  
    min = std::numeric_limits<int>::max();  
    max = std::numeric_limits<int>::min();  
  
    for (int i = 0; i < longitud; i++) {  
        min = std::min(min, array[i]);  
        max = std::max(max, array[i]);  
    }  
}
```

- ¿Son parámetros de salida o de E/S?
- Llamadas a función:

```
int min, max;  
min_max(arr, longitud, min, max);
```

# Múltiples resultados

- ¿Cómo podemos especificar varios valores de retorno para una función, sin tener que recurrir a parámetros de salida?



# Tipo específico

```
struct MinMaxResult {  
    int min;  
    int max;  
};  
  
MinMaxResult min_max(int *array, int longitud) {  
    MinMaxResult res;  
    res.min = std::numeric_limits<int>::max();  
    res.max = std::numeric_limits<int>::min();  
  
    for (int i = 0; i < longitud; i++) {  
        res.min = std::min(res.min, array[i]);  
        res.max = std::max(res.max, array[i]);  
    }  
  
    return res;  
}
```

# Tipo específico

- Llamada a la función:

```
MinMaxResult r = min_max(arr, longitud);
std::cout << "Min = " << r.min << " | Max = " << r.max;
```

- Problema: tener que definir un tipo específico.

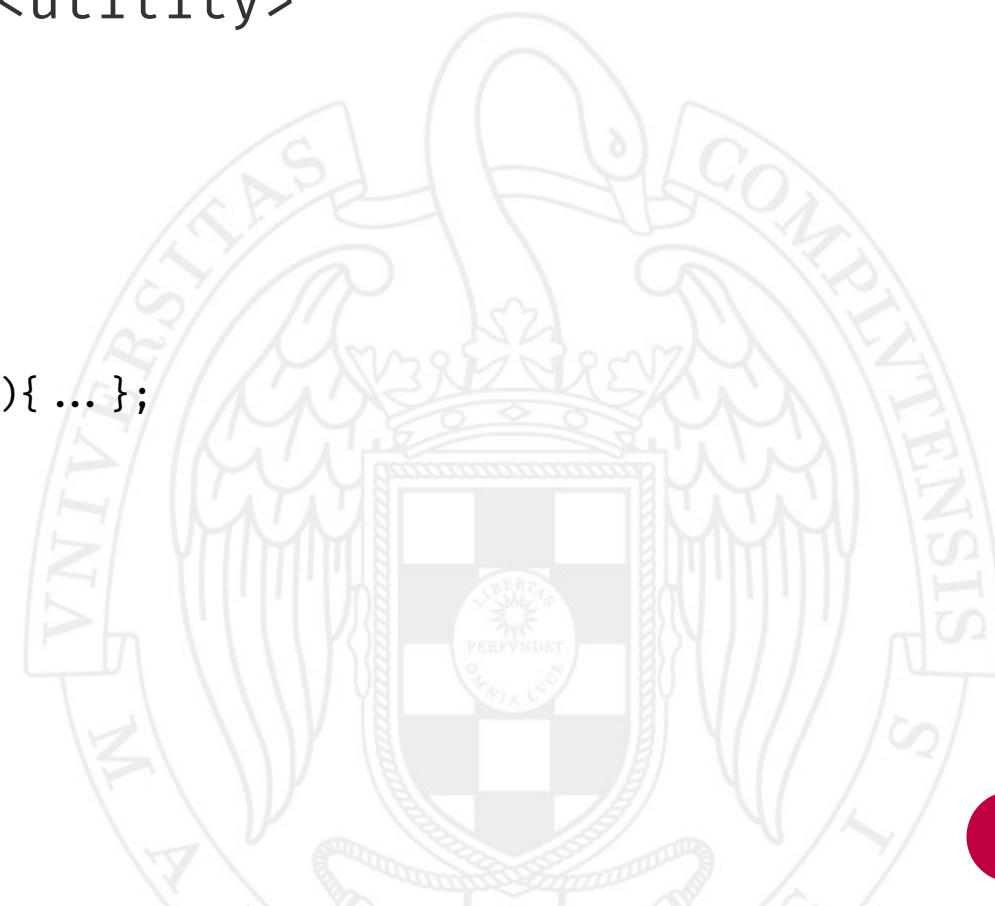
# Pares – std :: pair

# La clase pair

- Denota un par de elementos ( $x, y$ ), que pueden ser de distinto tipo.
- Definida en el fichero de cabecera `<utility>`

```
template <typename T1, typename T2>
class pair {
public:
    T1 first;
    T2 second;

    pair(const T1 &first, const T2 &second){ ... };
    ...
};
```



# La clase pair

```
std::pair<int, int> min_max(int *array, int longitud) {
    int min = std::numeric_limits<int>::max();
    int max = std::numeric_limits<int>::min();

    for (int i = 0; i < longitud; i++) {
        min = std::min(min, array[i]);
        max = std::max(max, array[i]);
    }

    return std::pair<int, int>(min, max);
}
```



# La clase pair

```
std::pair<int, int> min_max(int *array, int longitud) {
    int min = std::numeric_limits<int>::max();
    int max = std::numeric_limits<int>::min();

    for (int i = 0; i < longitud; i++) {
        min = std::min(min, array[i]);
        max = std::max(max, array[i]);
    }

    return {min, max};
}
```



# La clase pair

- Llamada a la función:

```
std::pair<int, int> p = min_max(arr, longitud);
std::cout << "Min = " << p.first << " | Max = " << p.second;
```

- Sintaxis abreviada (*structured binding declaration*) de C++17.

```
auto [min, max] = min_max(arr, longitud);
std::cout << "Min = " << min << " | Max = " << max << std::endl;
```

- En Visual Studio 2019 es necesario activar la opción /std:c++17 o /std:c++latest.

# La clase pair

- Hace explícitos los valores de salida.
- No requiere declarar ninguna clase.
- Pero... conviene documentar el significado de las componentes:

```
// Devuelve un par de enteros.  
// - La primera componente es el valor mínimo del array  
// - La segunda componente es el valor máximo del array  
  
std::pair<int, int> min_max(int *array, int longitud) {  
    ...  
}
```

*¿Y si la función devuelve más de dos valores?*

## Tuplas – std :: tuple

# La clase tuple

- Definida en el fichero de cabecera <tuple>

```
// Devuelve una tupla con tres componentes:  
// - La primera componente es el valor mínimo del array  
// - La segunda componente es el valor máximo del array  
// - La tercera componente es la suma de los valores del array  
  
std::tuple<int, int, int> min_max_sum(int *array, int longitud) {  
    int min = std::numeric_limits<int>::max();  
    int max = std::numeric_limits<int>::min();  
    int sum = 0;  
  
    for (int i = 0; i < longitud; i++) {  
        min = std::min(min, array[i]);  
        max = std::max(max, array[i]);  
        sum += array[i];  
    }  
  
    return {min, max, sum};  
}
```

# La clase tuple

- Llamada:

```
auto [min, max, sum] = min_max_sum(arr, longitud);
std::cout << "Min = " << min << " | Max = " << max << " | Sum = " << sum;
```



ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Objetos función

Manuel Montenegro Montes

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

# Recordatorio

- Función que recibe una lista, una función booleana y elimina aquellos elementos para los que la función devuelve `true`.

```
template <typename T, typename U>
void eliminar(std::list<U> &elems, T func) {
    auto it = elems.begin();
    while (it != elems.end()) {
        if (func(*it)) {
            it = elems.erase(it);
        } else {
            ++it;
        }
    }
}
```

# ¿Qué puedo pasar como parámetro func?

```
template <typename T, typename U>
void eliminar(std::list<U> &elems, T func) {
    ...
    if (func(*it)) { ... }
    ...
}
```

- Cualquier cosa sobre la que se pueda realizar una llamada.
  - En particular, cualquier función que acepte un solo parámetro.
  - ...¿algo más?

# ¿Qué operadores pueden sobrecargarse?

+ - \* / % ^ & | << >>  
== <= >= != < > && || !  
= += -= \*= /=  
++ --  
[] () →  
new delete  
etc.

# Sobrecarga del operador ()

- C++ permite sobrecargar el operador () .

```
class Prueba {  
public:  
    void operator()(parametros) { ... }  
};
```

- Este operador es invocado cuando se evalúa una expresión de la forma x( args ), donde x es una instancia de la clase Prueba.

# Ejemplo

```
class SumaUno {  
public:  
    int operator()(int x) { return x + 1; }  
};
```

- Supongamos que declaro una instancia de la clase SumaUno:  
`SumaUno s;`
- La expresión `s(3)` equivale a `s.operator()(3)` y se evaluará al valor 4.
- ¡Ojo! `s` no es una función; es un objeto que se comporta como una función.

# Objetos función

- Un **objeto función** es una instancia de una clase que sobrecarga el operador ( ).
- En nuestro ejemplo:

SumaUno s;

s es un objeto función.



# Uso de los objetos función

- Los objetos función pueden ser utilizados en cualquier contexto en el que se requiera una función.

```
template <typename T, typename U>
void eliminar(std::list<U> &elems, T func) {
    ...
    if (func(*it)) { ... }
    ...
}
```

# Ejemplo

```
class EsPar {  
public:  
    bool operator()(int x) { return x % 2 == 0; }  
};  
  
int main() {  
    ...  
    EsPar obj_fun;  
    eliminar(v1, obj_fun);  
    ...  
}
```



# ¿Para qué sirven los objetos función?

# ¿Cuál es la diferencia?

Entre esto...

```
class EsPar {  
public:  
    bool operator()(int x) { return x % 2 == 0; }  
};
```

...y esto...

```
bool es_par(int x) { return x % 2 == 0; }
```

# Ejemplo: criba de Eratóstenes

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30

# Ejemplo: criba de Eratóstenes

- Supongamos que tenemos una lista con los números 2, 3, 4, 5, ..., 100.
  - Eliminamos los múltiplos de 2.
  - Eliminamos los múltiplos de 3.
  - Eliminamos los múltiplos de 5.
  - etc.



# Ejemplo: criba de Eratóstenes

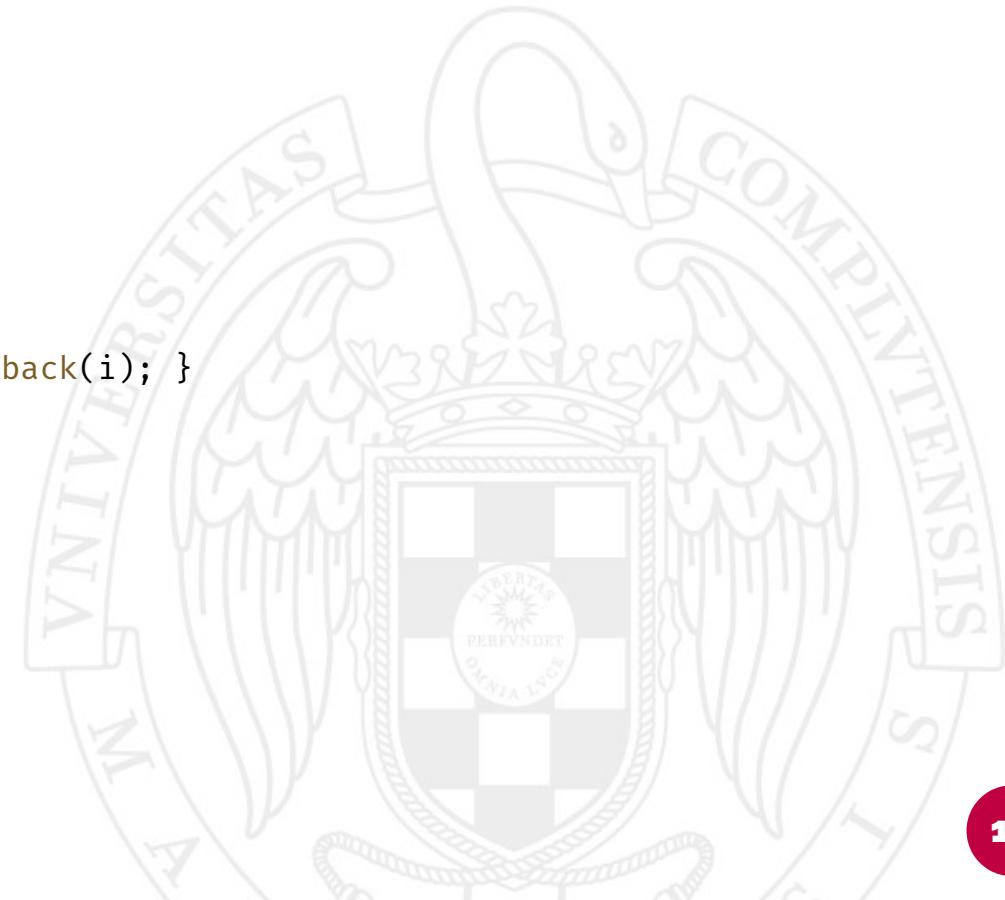
```
bool es_multiplo_de_dos(int x) {
    return x % 2 == 0;
}

bool es_multiplo_de_tres(int x) {
    return x % 3 == 0;
}

bool es_multiplo_de_cinco(int x) { ... }

int main() {
    std::list<int> lista;
    for (int i = 2; i <= 100; i++) { lista.push_back(i); }

    std::list<int> primos;
    primos.push_back(lista.front());
    eliminar(lista, es_multiplo_de_dos);
    primos.push_back(lista.front());
    eliminar(lista, es_multiplo_de_tres);
    primos.push_back(lista.front());
    eliminar(lista, es_multiplo_de_cinco);
    ...
}
```



# Ejemplo: criba de Eratóstenes

```
bool es_multiplo_de_y(int x, int y) {
    return x % y == 0;
}

int main() {
    std::list<int> lista;
    for (int i = 2; i <= 100; i++) { lista.push_back(i); }

    std::list<int> primos;
    primos.push_back(lista.front());
    eliminar(lista, es_multiplo_de_y);
    primos.push_back(lista.front());
    eliminar(lista, es_multiplo_de_y);
    primos.push_back(lista.front());
    eliminar(lista, es_multiplo_de_y);
    ...
}
```



# Ejemplo: criba de Eratóstenes

```
class EsMultiploDeY {
private:
    int y;
public:
    EsMultiploDeY(int y): y(y) { }
    bool operator()(int x) { return x % y == 0; }
};

int main() {
    std::list<int> lista;
    for (int i = 2; i <= 100; i++) { lista.push_back(i); }

    EsMultiploDeY mult_dos(2), mult_tres(3), mult_cinco(5);
    std::list<int> primos;
    primos.push_back(lista.front());
    eliminar(lista, mult_dos);
    primos.push_back(lista.front());
    eliminar(lista, mult_tres);
    primos.push_back(lista.front());
    eliminar(lista, mult_cinco);
    ...
}
```

# Ejemplo: criba de Eratóstenes

```
class EsMultiploDeY {
private:
    int y;
public:
    EsMultiploDeY(int y): y(y) { }
    bool operator()(int x) { return x % y == 0; }
};

int main() {
    std::list<int> lista;
    for (int i = 2; i <= 100; i++) { lista.push_back(i); }

    std::list<int> primos;
    while (!lista.empty()) {
        primos.push_back(lista.front());
        EsMultiploDeY multiplos_de_front(lista.front());
        eliminar(lista, multiplos_de_front);
    }
}
```

# ¿Para qué sirve un objeto función?

- Cuando queremos pasar una función como parámetro, pero esa función, además de sus argumentos, depende de otros valores.

```
class EsMultiploDeY {  
private:  
    int y;  
public:  
    EsMultiploDeY(int y): y(y) { }  
    bool operator()(int x) { return x % y == 0; }  
};
```



ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Expresiones lambda (C++11)

Manuel Montenegro Montes

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

# Recordatorio

- Función que recibe una lista, una función booleana y elimina aquellos elementos para los que la función devuelve `true`.

```
template <typename T, typename U>
void eliminar(std::list<U> &elems, T func) {
    auto it = elems.begin();
    while (it != elems.end()) {
        if (func(*it)) {
            it = elems.erase(it);
        } else {
            ++it;
        }
    }
}
```

# Recordatorio

Hasta ahora hemos pasado como argumento func:

- **Funciones:**

```
bool es_par(int x) { return x % 2 == 0; }  
...  
eliminar(v1, es_par);
```

- **Objetos función:**

```
class EsMultiploDeY { ... }  
...  
EsMultiploDeY multiplo_de_dos(2);  
eliminar(v1, multiplo_de_dos);
```

- En cualquier caso, tenemos que definir una función o una clase aparte.
  - y es posible que solamente se utilice una vez.

# Expresiones lambda

- Nos permiten declarar un objeto función en el sitio en el que se utiliza, con una sintaxis más breve.
- Sintaxis:

```
[capturas] (parámetros) { cuerpo }
```

# Ejemplo

- En lugar de

```
bool es_par(int x) { return x % 2 == 0; }
...
eliminar(v1, es_par);
```

- Puede escribirse

```
eliminar(v1, [](int x) { return x % 2 == 0; });
```

# Más ejemplos

```
std::list<int> v1 = {1, 5, 6, 9, 10, 20};
```

```
std::list<int> v2 = v1;
```

```
eliminar(v1, [](int x) { return x % 2 == 0; });
```

```
std::cout << v1 << std::endl;
```

```
eliminar(v2, [](int x) { return x % 2 == 1; });
```

```
std::cout << v2 << std::endl;
```

```
std::list<int> v3 = {-2, 3, 10, -6, 20};
```

```
eliminar(v3, [](int x) { return x > 0; });
```

```
std::cout << v3 << std::endl;
```

```
std::list<Fecha> v4 = { {25, 12, 2010}, {10, 21, 2020}, {25, 12, 1900}, {1, 1, 2000} };
```

```
eliminar(v4, [](const Fecha &f) { return f.get_dia() == 25 && f.get_mes() == 12; });
```

```
std::cout << v4 << std::endl;
```

# Capturas

- Las expresiones lambda pueden tener, en su cuerpo, referencias a variables *externas* (esto es, variables distintas a los parámetros).

```
int y = 3;  
eliminar(v, [](int x) { return x % y = 0; });
```

- Cuando esto ocurre, decimos que la variable y está **capturada** por la expresión lambda.
- C++ nos obliga a declarar las variables capturadas dentro de [ ].

```
int y = 3;  
eliminar(v, [y](int x) { return x % y = 0; });
```

# Capturas

Hay dos maneras de capturar variables:

- **Por valor**

```
[y](int x) { /* ... */ }
```

Dentro de la lambda expresión no se pueden realizar cambios sobre la variable y.

- **Por referencia**

```
[&y](int x) { /* ... */ }
```

La lambda expresión trabaja con una **referencia** a la variable y.

Cualquier cambio que se haga sobre la variable y dentro de la lambda expresión afectará a la variable y externa.

# Ejemplo

```
int y = 3;  
auto f = [&y]() { y++; };  
f();  
std::cout << y << std::endl;
```



# Criba de eratóstenes: el retorno

```
std::list<int> lista;
std::list<int> primos;
...
while (!lista.empty()) {
    int primero = lista.front();
    primos.push_back(primer);
    eliminar(lista, [primer](int x) { return x % primero == 0; });
}
std::cout << primos << std::endl;
```

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# STL: Algoritmos (2)

Manuel Montenegro Montes

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

# Funciones de orden superior

# La función transform

`transform(ini, fin, dest, fun)`

- Definida en `<algorithm>`
- Aplica la función `fun` al conjunto de elementos contenido entre los iteradores `[ini, fin]`.
- Los resultados devueltos por `fun` son copiados a partir del iterador `dest`.
- Si se desea modificar la lista original, utilizar `dest = ini`.

# Ejemplos

```
vector<int> v = { 3, 10, 9, 3, 15 };
transform(v.begin(), v.end(), v.begin(), [](int x) { return x * 2; });
```

v = [6, 20, 18, 6, 30]

```
vector<string> nombres = {"Juan", "Rosario", "Amalia"};
vector<int> longitudes;
```

```
transform(nombres.begin(), nombres.end(),
         back_insert_iterator<vector<int>>(longitudes),
         [](const string &x) { return x.length(); });
```

longitudes = [4, 7, 6]

# La función `remove_if`

`remove_if(ini, fin, fun)`

- Definida en `<algorithm>`
- Elimina del rango de elementos `[ini, fin]` aquellos para los que `fun` devuelve `true`.
- Devuelve un iterador tras el último elemento de la colección resultante.

# Ejemplo

```
vector<int> v2 = { 3, 10, 8, 7, 4 };
auto it_end = remove_if(v2.begin(), v2.end(), [](int x) { return x % 2 == 0; });

copy(v2.begin(), it_end, ostream_iterator<int>(cout, " "));
```

3 7

# Las funciones `find_if` y `count_if`

`find_if(ini, fin, fun)`

- Devuelve un iterador al primer elemento del rango `[ini, fin]` para el que `fun` devuelve `true`.

`count_if(ini, fin, fun)`

- Devuelve el número de elementos del rango `[ini, fin]` para los que `fun` devuelve `true`.

# Ejemplo

```
vector<Fecha> fechas = {{10, 3, 2010}, {1, 6, 2019}, {28, 8, 1985}, {19, 3, 2001}};  
  
auto it_marzo = find_if(fechas.begin(), fechas.end(),  
                       [](const Fecha &f) { return f.get_mes() = 3; });  
  
cout << *it_marzo << endl;    10/03/2010  
  
int num_fechas_verano =  
    count_if(fechas.begin(), fechas.end(),  
             [](const Fecha &f) { return f.get_mes() ≥ 6 && f.get_mes() ≤ 8; });  
  
cout << num_fechas_verano << endl; 2
```

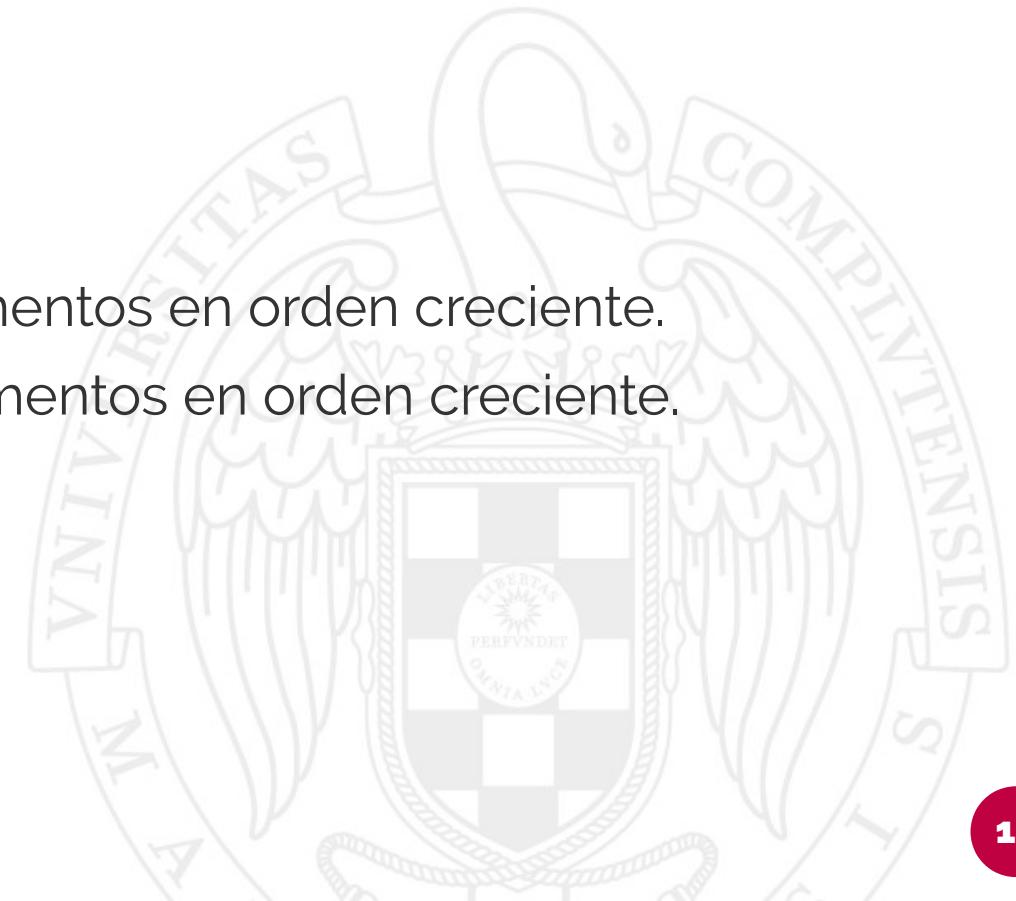
# Otras funciones de orden superior

- `all_of(ini, fin, fun)`
  - `any_of(ini, fin, fun)`
  - `none_of(ini, fin, fun)`
- 
- `accumulate(ini, fin, valor_inicial)`
  - `accumulate(ini, fin, valor_inicial, fun)`

# Funciones sobre conjuntos

# Funciones sobre conjuntos

- Las siguientes funciones pueden aplicarse sobre colecciones tales que, al ser iteradas, produzcan secuencias de elementos en orden ascendente. Esto incluye:
  - `set` (pero no `unordered_set`)
  - `map` (pero no `unordered_map`)
  - Listas que almacenen sus elementos en orden creciente.
  - Arrays que almacenen sus elementos en orden creciente.



# Funciones sobre conjuntos

- `includes(ini1, fin1, ini2, fin2)`
- `set_union(ini1, fin1, ini2, fin2, dest)`
- `set_intersection(ini1, fin1, ini2, fin2, dest)`
- `set_difference(ini1, fin1, ini2, fin2, dest)`

# Ejemplos

```
set<int> elems1 = { 6, 1, 9, 4, 3, 10 };
set<int> elems2 = { 10, 1, 4, 6 };

cout << includes(elems1.begin(), elems1.end(), elems2.begin(), elems2.end()) << endl; true
```

```
set<string> chicos = {"Ricardo", "Jaime", "Rafa", "Enrique", "Adrián", "Jose"};
set<string> chicas = {"Clara", "Susana", "Jose", "Natalia", "Elvira"};
list<string> result;
```

```
set_union(chicos.begin(), chicos.end(),
          chicas.begin(), chicas.end(),
          back_insert_iterator<list<string>>(result));
```

```
result = [Adrián, Clara, Elvira, Enrique, Jaime, Jose, Natalia, Rafa, Ricardo, Susana]
```

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# STL: Algoritmos (2)

Manuel Montenegro Montes

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

# Funciones de orden superior

# La función transform

`transform(ini, fin, dest, fun)`

- Definida en `<algorithm>`
- Aplica la función `fun` al conjunto de elementos contenido entre los iteradores `[ini, fin]`.
- Los resultados devueltos por `fun` son copiados a partir del iterador `dest`.
- Si se desea modificar la lista original, utilizar `dest = ini`.

# Ejemplos

```
vector<int> v = { 3, 10, 9, 3, 15 };
transform(v.begin(), v.end(), v.begin(), [](int x) { return x * 2; });
```

v = [6, 20, 18, 6, 30]

```
vector<string> nombres = {"Juan", "Rosario", "Amalia"};
vector<int> longitudes;
```

```
transform(nombres.begin(), nombres.end(),
         back_insert_iterator<vector<int>>(longitudes),
         [](const string &x) { return x.length(); });
```

longitudes = [4, 7, 6]

# La función `remove_if`

`remove_if(ini, fin, fun)`

- Definida en `<algorithm>`
- Elimina del rango de elementos `[ini, fin]` aquellos para los que `fun` devuelve `true`.
- Devuelve un iterador tras el último elemento de la colección resultante.

# Ejemplo

```
vector<int> v2 = { 3, 10, 8, 7, 4 };
auto it_end = remove_if(v2.begin(), v2.end(), [](int x) { return x % 2 == 0; });

copy(v2.begin(), it_end, ostream_iterator<int>(cout, " "));
```

3 7



# Las funciones `find_if` y `count_if`

`find_if(ini, fin, fun)`

- Devuelve un iterador al primer elemento del rango `[ini, fin]` para el que `fun` devuelve `true`.

`count_if(ini, fin, fun)`

- Devuelve el número de elementos del rango `[ini, fin]` para los que `fun` devuelve `true`.

# Ejemplo

```
vector<Fecha> fechas = {{10, 3, 2010}, {1, 6, 2019}, {28, 8, 1985}, {19, 3, 2001}};  
  
auto it_marzo = find_if(fechas.begin(), fechas.end(),  
                       [](const Fecha &f) { return f.get_mes() = 3; });  
  
cout << *it_marzo << endl;    10/03/2010  
  
int num_fechas_verano =  
    count_if(fechas.begin(), fechas.end(),  
             [](const Fecha &f) { return f.get_mes() ≥ 6 && f.get_mes() ≤ 8; });  
  
cout << num_fechas_verano << endl; 2
```

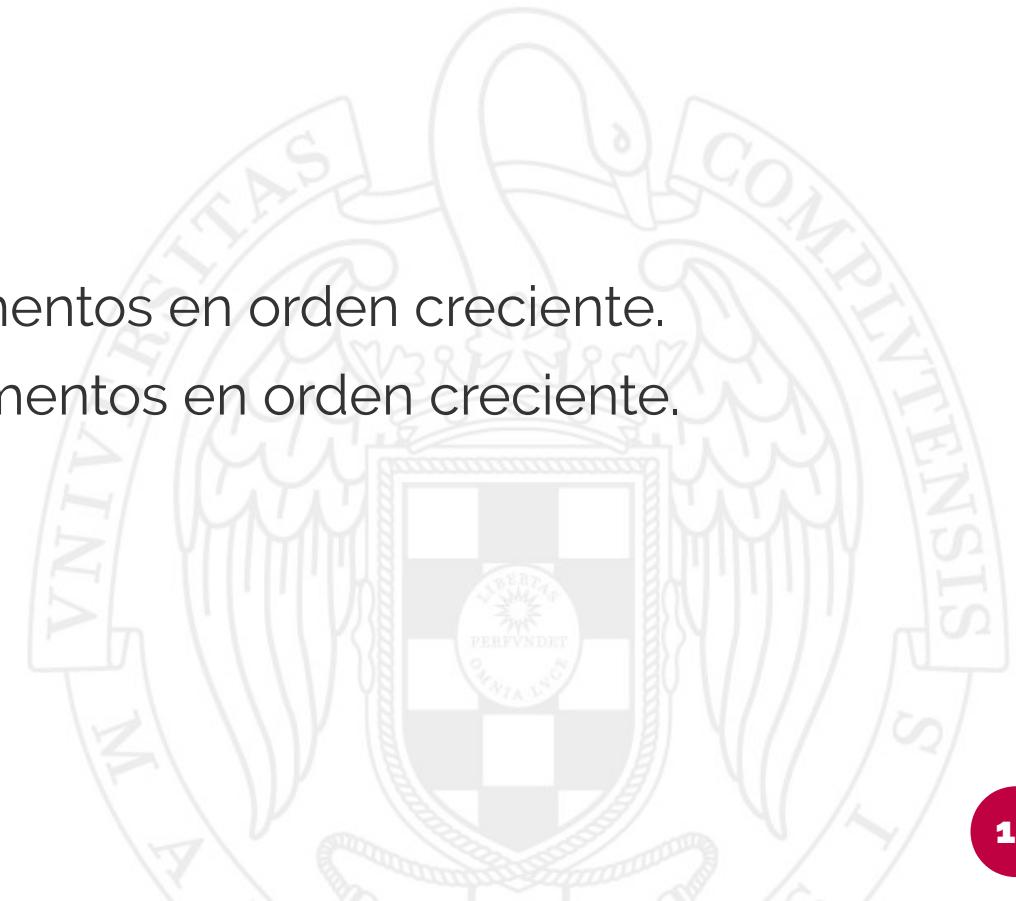
# Otras funciones de orden superior

- `all_of(ini, fin, fun)`
  - `any_of(ini, fin, fun)`
  - `none_of(ini, fin, fun)`
- 
- `accumulate(ini, fin, valor_inicial)`
  - `accumulate(ini, fin, valor_inicial, fun)`

# Funciones sobre conjuntos

# Funciones sobre conjuntos

- Las siguientes funciones pueden aplicarse sobre colecciones tales que, al ser iteradas, produzcan secuencias de elementos en orden ascendente. Esto incluye:
  - `set` (pero no `unordered_set`)
  - `map` (pero no `unordered_map`)
  - Listas que almacenen sus elementos en orden creciente.
  - Arrays que almacenen sus elementos en orden creciente.



# Funciones sobre conjuntos

- `includes(ini1, fin1, ini2, fin2)`
- `set_union(ini1, fin1, ini2, fin2, dest)`
- `set_intersection(ini1, fin1, ini2, fin2, dest)`
- `set_difference(ini1, fin1, ini2, fin2, dest)`

# Ejemplos

```
set<int> elems1 = { 6, 1, 9, 4, 3, 10 };
set<int> elems2 = { 10, 1, 4, 6 };

cout << includes(elems1.begin(), elems1.end(), elems2.begin(), elems2.end()) << endl; true
```

```
set<string> chicos = {"Ricardo", "Jaime", "Rafa", "Enrique", "Adrián", "Jose"};
set<string> chicas = {"Clara", "Susana", "Jose", "Natalia", "Elvira"};
list<string> result;
```

```
set_union(chicos.begin(), chicos.end(),
          chicas.begin(), chicas.end(),
          back_insert_iterator<list<string>>(result));
```

```
result = [Adrián, Clara, Elvira, Enrique, Jaime, Jose, Natalia, Rafa, Ricardo, Susana]
```

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Manejo de excepciones

Manuel Montenegro Montes

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

# Lanzar y capturar excepciones

# Lanzamiento de excepciones

- Se utiliza la palabra clave **throw**.
- Recibe como argumento la excepción a lanzar.
  - Puede ser un objeto (*recomendado*) o un valor básico.
- No es necesario declarar los tipos de excepciones lanzadas.

```
class division_por_cero { };

double dividir(double x, double y) {
    if (y == 0) {
        throw division_por_cero();
    } else {
        return x / y;
    }
}
```

# Captura de excepciones

- Se utilizan bloques **try/catch**, con sintaxis similar a la de Java.
- Se permiten varios bloques catch, cada uno capturando un tipo distinto.
- No existe bloque **finally**.

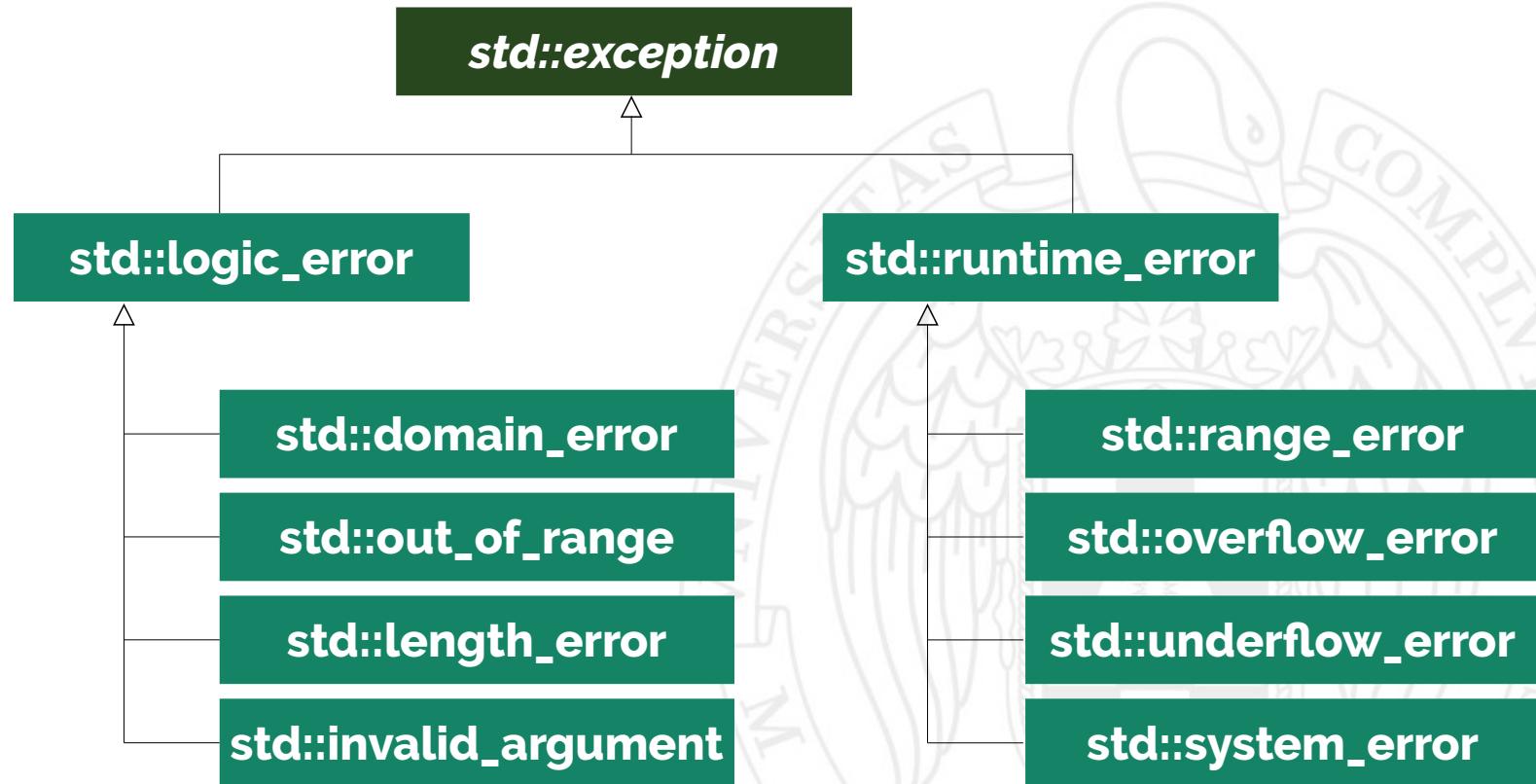
```
try {  
    dividir(1, 0);  
} catch (division_por_cero &e) {  
    std::cout << "División por cero!" << std::endl;  
}
```

La excepciones  
se capturan por  
referencia

# Jerarquía de excepciones estándar

# Excepciones estándar

- Ficheros de cabecera <exception> y <stdexcept>.



# Excepciones estándar

- Ficheros de cabecera <exception> y <stdexcept>.

***std::exception***

const char \*what()

Descripción de la excepción

# Ejemplo

```
double dividir(double x, double y) {
    if (y == 0) {
        throw std::domain_error("división por cero");
    } else {
        return x / y;
    }
}

int main() {
    try {
        dividir(1, 0);
    } catch (std::exception &e) {
        std::cout << e.what() << std::endl;
    }
}
```



# Heredar de excepciones estándar

```
class division_por_cero: public std::logic_error {  
public:  
    division_por_cero(): std::logic_error("división por cero") {}  
};  
  
double dividir(double x, double y) {  
    if (y == 0) {  
        throw division_por_cero();  
    } else {  
        return x / y;  
    }  
}  
  
int main() {  
    try {  
        dividir(1, 0);  
    } catch (division_por_cero &e) {  
        std::cout << e.what() << std::endl;  
    }  
}
```

**std::logic\_error**

**division\_por\_cero**

ESTRUCTURAS DE DATOS

NOTAS SOBRE C++

# Herencia y polimorfismo

Manuel Montenegro Montes

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

# Herencia



# Heredar de una clase

```
class Rectangulo {  
public:  
    Rectangulo(double ancho, double alto): ancho(ancho), alto(alto) {}  
  
    double area() { return ancho * alto; }  
    double perimetro() { return 2 * ancho + 2 * alto; }  
  
protected:  
    double ancho, alto;  
};  
  
class Cuadrado: public Rectangulo {  
public:  
    Cuadrado(double lado): Rectangulo(lado, lado) {}  
};
```

# Ejemplo

```
Rectangulo *r;  
double ancho, alto;  
cin >> ancho >> alto;  
  
if (ancho == alto) {  
    r = new Cuadrado(ancho);  
} else {  
    r = new Rectangulo(ancho, alto);  
}  
  
double area = r->area();  
double perimetro = r->perimetro();  
cout << "Area: " << area << endl;  
cout << "Perímetro: " << perimetro << endl;  
  
delete r;
```



# Polimorfismo y métodos virtuales

# Nuevo método: dibujar()

```
class Rectangulo {  
public:  
    ...  
    void dibujar() {  
        std::cout << "Rectángulo de ancho " << ancho << " y alto " << alto << std::endl;  
    }  
protected:  
    double ancho, alto;  
};  
  
class Cuadrado: public Rectangulo {  
public:  
    Cuadrado(double lado): Rectangulo(lado, lado) { }  
  
    void dibujar() {  
        std::cout << "Cuadrado de lado " << ancho << std::endl;  
    }  
};
```

# Ejemplo

```
Rectangulo *r;  
double ancho, alto;  
cin >> ancho >> alto;  
  
if (ancho == alto) {  
    r = new Cuadrado(ancho);  
} else {  
    r = new Rectangulo(ancho, alto);  
}  
  
r->dibujar();  
  
delete r;
```

1.2 4.5

Rectángulo de ancho 1.2 y alto 4.5

1.2 1.2

Rectángulo de ancho 1.2 y alto 1.2



# Vinculación estática vs dinámica

- C++ determina a qué método llamar en base al tipo del objeto sobre el que se realiza la llamada.

```
Rectangulo *r;  
double ancho, alto;  
cin >> ancho >> alto;  
  
if (ancho == alto) {  
    r = new Cuadrado(ancho);  
} else {  
    r = new Rectangulo(ancho, alto);  
}  
  
r->dibujar();  
  
delete r;
```

r es de tipo puntero a Rectangulo  
Por tanto el compilador determina que  
r->dibujar() llama al método  
dibujar de Rectangulo.

# Vinculación estática vs dinámica

- Si se realiza **vinculación dinámica**, decimos al compilador que se compruebe, en tiempo de ejecución, la clase a la que pertenece el objeto, y se llame al método correspondiente a esa clase, independientemente del tipo.
- Por defecto, en C++ se utiliza vinculación estática.
- Por defecto, en Java se utiliza vinculación dinámica.

# Habilitar la vinculación dinámica

```
class Rectangulo {  
public:  
    ...  
    virtual void dibujar() {  
        std::cout << "Rectángulo de ancho " << ancho << " y alto " << alto << std::endl;  
    }  
protected:  
    double ancho, alto;  
};  
  
class Cuadrado: public Rectangulo {  
public:  
    Cuadrado(double lado): Rectangulo(lado, lado) {}  
  
    virtual void dibujar() {  
        std::cout << "Cuadrado de lado " << ancho << std::endl;  
    }  
};
```

# Ejemplo

```
Rectangulo *r;  
double ancho, alto;  
cin >> ancho >> alto;  
  
if (ancho == alto) {  
    r = new Cuadrado(ancho);  
} else {  
    r = new Rectangulo(ancho, alto);  
}  
  
r->dibujar();  
  
delete r;
```

1.2 4.5

Rectángulo de ancho 1.2 y alto 4.5

1.2 1.2

Cuadrado de lado 1.2



# Reglas generales

- Cualquier método que sea susceptible de ser reescrito debe declararse como `virtual`.
- Si una clase tiene un método `virtual`, es muy aconsejable declarar su destructor como `virtual`, aunque no haga nada.



# Métodos abstractos

```
class Figura {  
public:  
    virtual double area() = 0;  
    virtual double perimetro() = 0;  
    virtual void dibujar() = 0;  
  
    virtual ~Figura() {}  
  
};  
  
class Rectangulo: public Figura { ... }
```

- Los métodos abstractos han de ser virtuales.
- Si una clase tiene un método abstracto, la clase es abstracta.
  - No pueden crearse instancias de Figura.

# Otras diferencias con Java

- En C++ no existe la noción de interfaz (*interface*).
- Se permite herencia múltiple.



ESTRUCTURAS DE DATOS

NOTAS SOBRE JAVA

# Contenedores lineales

Manuel Montenegro Montes

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

# Librería estándar de Java

- Proporciona un gran número de clases con implementaciones de los TADs más comunes, y algoritmos para su manipulación.
- Estas clases suelen estar en el paquete `java.util`.



# Clases de TAD lineales

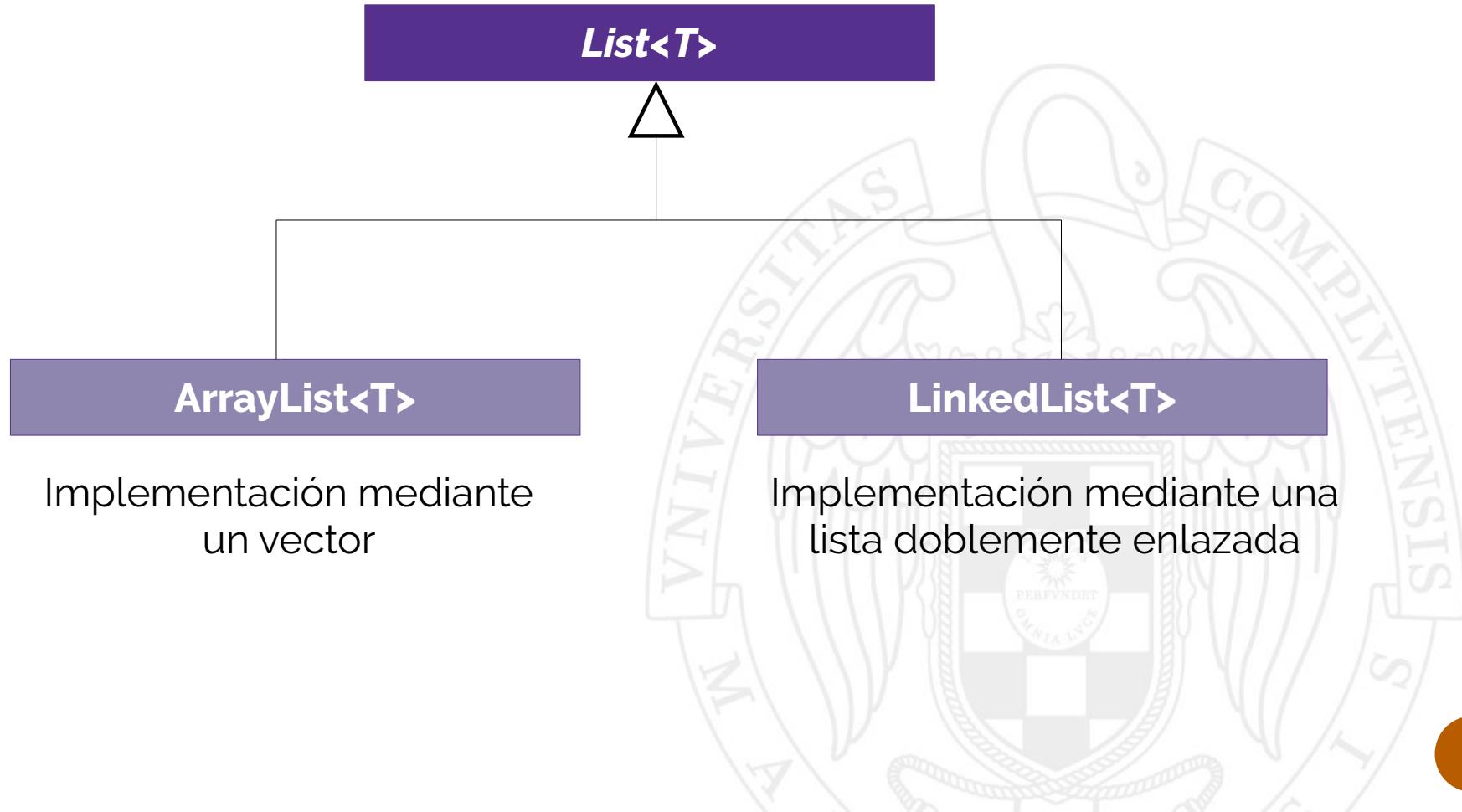
# La interfaz *List*

- Define las operaciones del TAD Lista.

## *List<T>*

- + add(T)
- + add(pos, T)
- + remove(pos)
- + contains(T): bool
- + get(pos): T
- + set(pos, T)
- + size(): int
- + subList(pos1, pos2): List<T>
- + toArray(T[]): T[]
- ...

# Implementaciones de `List<T>`



# Ejemplo

```
List<Integer> l = new ArrayList<>();
for (int i = 0; i < 10; i++) {
    l.add(i * 3);
}
System.out.println(l);
```

[0, 3, 6, 9, 12, 15, 18, 21, 24, 27]

# Otros TADs lineales

## Stack<T>

- + empty(): bool
- + peek(): T
- + pop(): T
- + push(T): bool
- ...

## Queue<T>

- + add(T)
- + element(): T
- + peek(): T
- + poll(): T
- ...

## Deque<T>

## ArrayDeque<T>

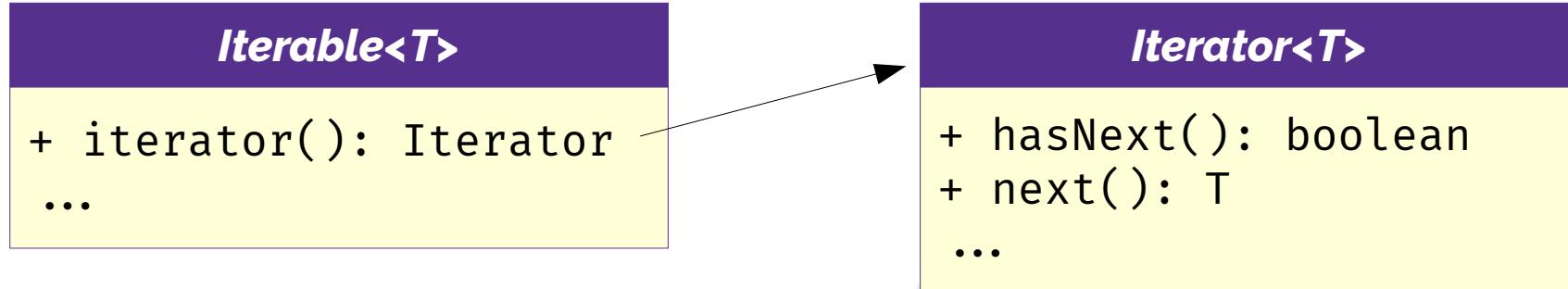
# Iteradores

# Iteradores

- Es posible obtener un iterador a partir de cualquier clase que implemente la interfaz **Iterable**.
  - List
  - Stack
  - Queue
  - etc.



# Iteradores



# Ejemplo

```
List<Integer> l = ...;  
... // Insertar elementos en l  
  
int suma = 0;  
Iterator<Integer> it = l.iterator();  
while (it.hasNext()) {  
    suma += it.next();  
}  
System.out.println(suma);
```

# Sintaxis alternativa

```
List<Integer> l = ...;  
... // Insertar elementos en l  
  
int suma = 0;  
Iterator<Integer> it = l.iterator();  
while (it.hasNext()) {  
    suma += it.next();  
}  
System.out.println(suma);
```

```
= {  
    for (Integer x: l) {  
        suma += x;  
    }  
}
```

# Funciones de utilidad

# La clase Collections

- Contiene varios métodos estáticos que trabajan con listas.
  - `Collections.copy(list_dest, list_orig)`
  - `Collections.fill(list, elem)`
  - `Collections.max(list)`
  - `Collections.binarySearch(list, elem)`
  - `Collections.sort(list)`

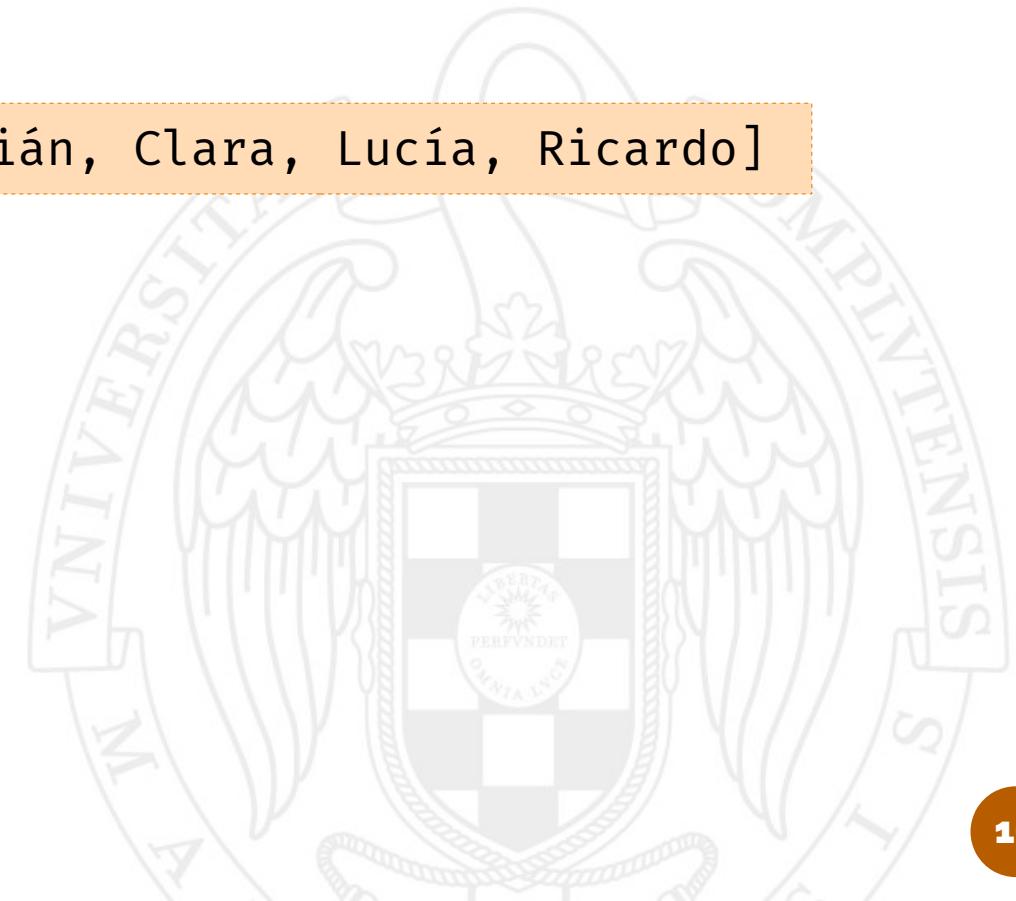
# La clase Arrays

- Utilidades similares, pero para arrays en lugar de listas.
  - `Arrays.asList(elems)`
  - `Arrays.binarySearch(array, elem)`
  - `Arrays.equals(array1, array2)`
  - `Arrays.sort(array)`
  - `Arrays.toString(array)`

# Ejemplo

```
List<String> l = Arrays.asList("Ricardo", "Adrián", "Lucía", "Clara");  
Collections.sort(l);  
System.out.println(l);
```

[Adrián, Clara, Lucía, Ricardo]



ESTRUCTURAS DE DATOS

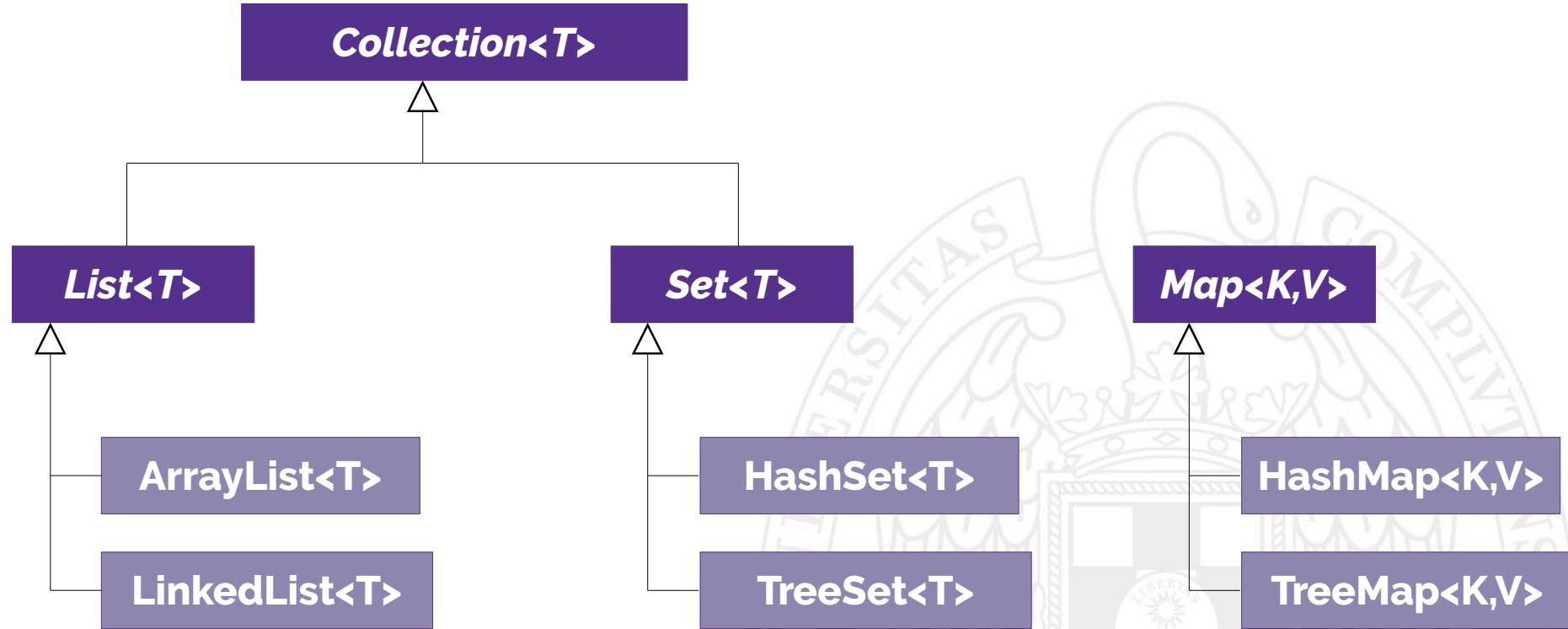
NOTAS SOBRE JAVA

# Contenedores asociativos

Manuel Montenegro Montes

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

# Jerarquía de contenedores



# Interfaz Set

## *Set<T>*

- + add(T)
- + contains(T): boolean
- + remove(T)
- + isEmpty(): boolean
- + size(): int
- + iterator(): Iterator<T>
- ...

# Interfaz Map

## ***Map<K, V>***

- + put(K, V)
- + containsKey(K): boolean
- + get(K): V
- + remove(K)
- + isEmpty(): boolean
- + size(): int
- + entrySet(): Set<Map.Entry<K, V>>
- ...

# Comparación entre elementos

## *Comparable<T>*

+ compareTo(T): int

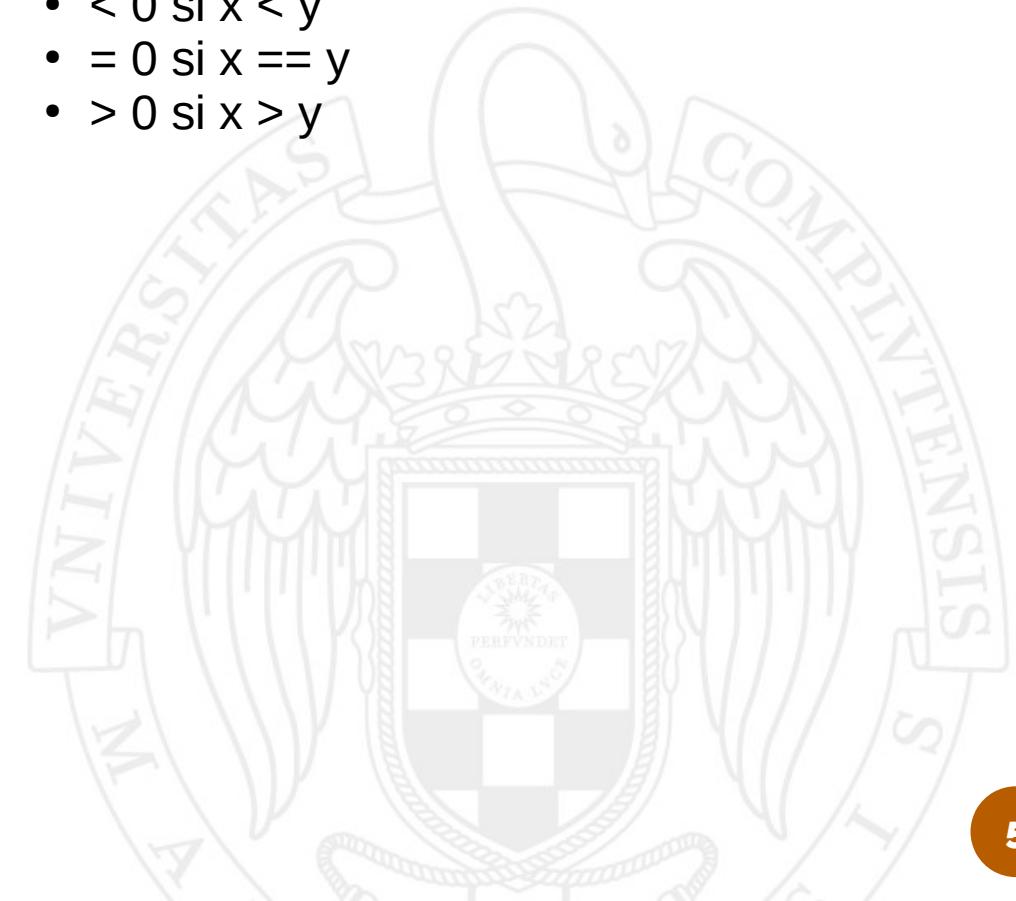
## *Object*

+ equals(Object): boolean

...

x.compareTo(y) devuelve:

- < 0 si x < y
- = 0 si x == y
- > 0 si x > y



# Ejemplo

```
public class Fecha implements Comparable<Fecha> {  
    private int dia, mes, anyo;  
    ...  
    @Override  
    public int compareTo(Fecha f) {  
        if (anyo == f.getAnyo()) {  
            if (mes == f.getMes()) {  
                return dia - f.getDia();  
            } else {  
                return mes - f.getMes();  
            }  
        } else {  
            return anyo - f.getAnyo();  
        }  
    }  
  
    @Override  
    public boolean equals(Object o) {  
        if (o instanceof Fecha) {  
            Fecha f = (Fecha)o;  
            return f.getDia() == dia && f.getMes() == mes && f.getAnyo() == anyo;  
        } else return false;  
    }  
}
```



# Ejemplo

```
Set<Fecha> sf = new TreeSet<>();  
  
sf.add(new Fecha(10, 4, 2010));  
sf.add(new Fecha(3, 10, 2011));  
sf.add(new Fecha(1, 10, 2010));  
  
for (Fecha f: sf) {  
    System.out.println(f);  
}
```

10/04/2010  
01/10/2010  
03/10/2011

# Funciones hash

## *Object*

- + equals(Object): boolean
- + hashCode(): int
- ...

Si `x.equals(y)` devuelve true,  
entonces debe cumplirse  
`x.hashCode() = y.hashCode()`

# Ejemplo

```
public class Fecha implements Comparable<Fecha> {  
    private int dia, mes, anyo;  
    ...  
    @Override  
    public int hashCode() {  
        int result = anyo;  
        result *= 100;  
        result += mes;  
        result *= 100;  
        result += dia;  
        return result;  
    }  
}
```

# Ejemplo

```
Set<Fecha> sf = new HashSet<>();  
sf.add(new Fecha(10, 4, 2010));  
sf.add(new Fecha(10, 4, 2010));  
  
for (Fecha f: sf) {  
    System.out.println(f);  
}
```

10/04/2010