

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

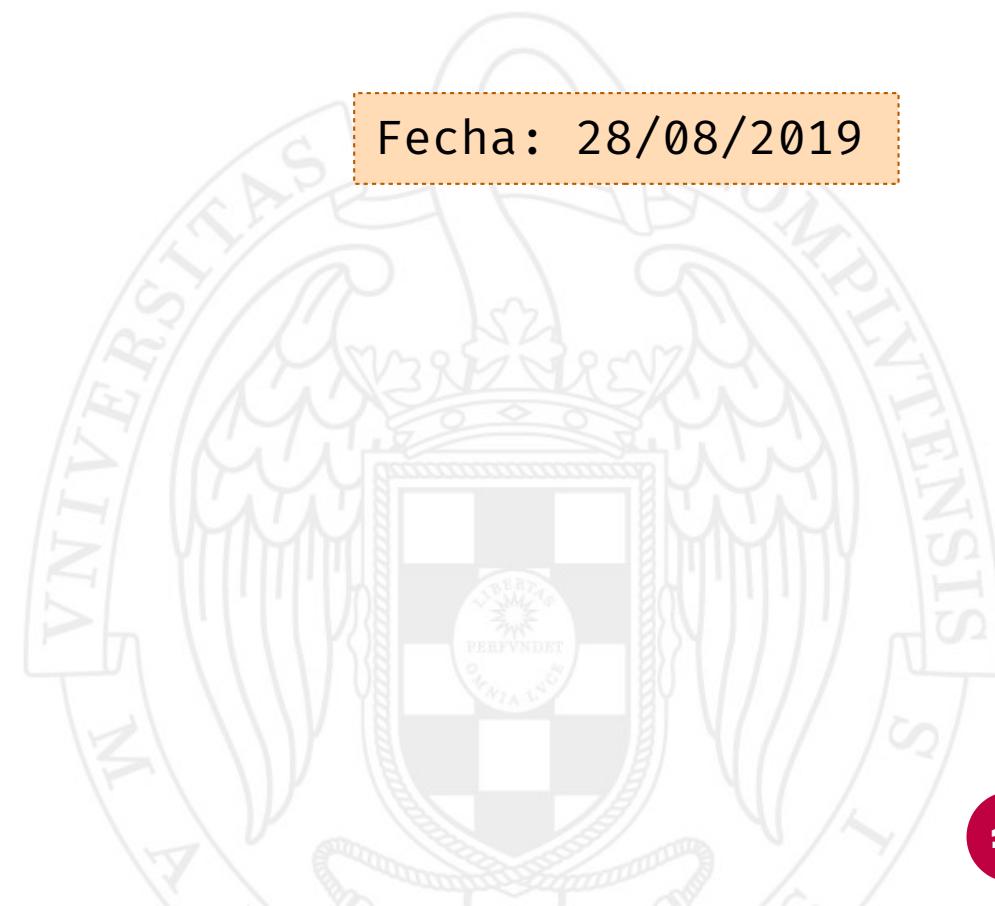
```
void Fecha::imprimir() {  
    std::cout << std::setfill('0') << std::setw(2) << dia << "/"  
        << std::setw(2) << mes << "/"  
        << std::setw(4) << anyo;  
}
```

```
#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*



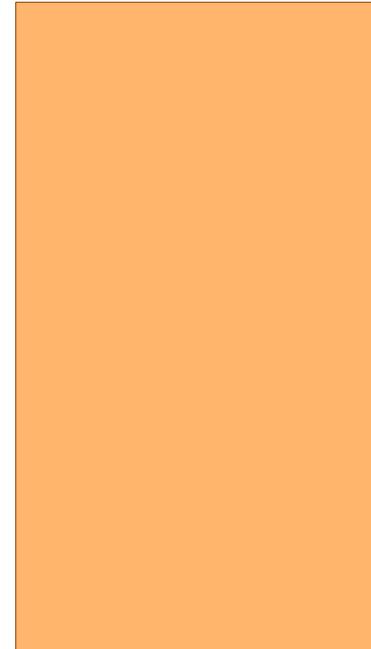
Regiones de memoria

- **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.

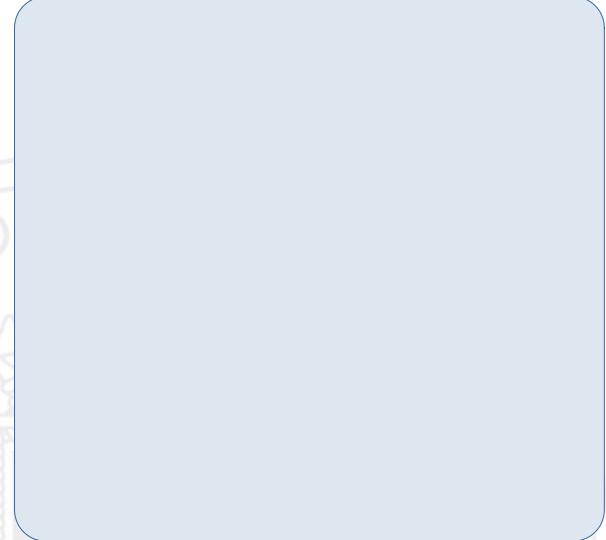
Regiones de memoria

```
int main() {  
    int x = 3;  
    int *y = new int;  
    *y = 3;  
    int *z = &x;  
  
    delete y;  
    return 0;  
}
```

Pila



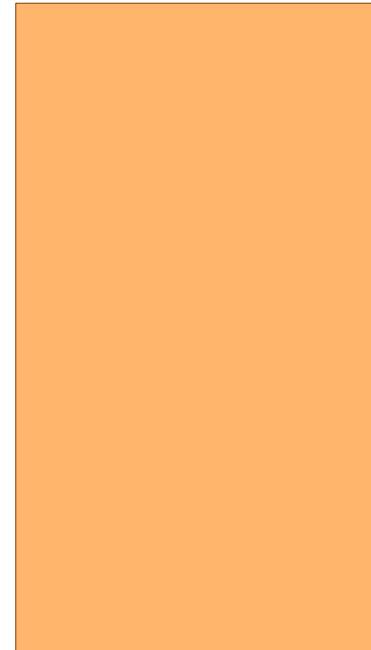
Heap



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

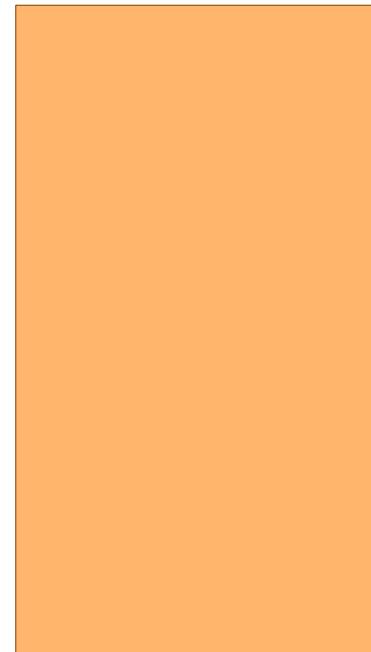
```
int main() {
    Fecha f1(28, 8, 2038);
    Fecha *f2 = new Fecha(10, 6, 2010);

    std::cout << "Fecha 1: ";
    f1.imprimir();
    std::cout << std::endl;

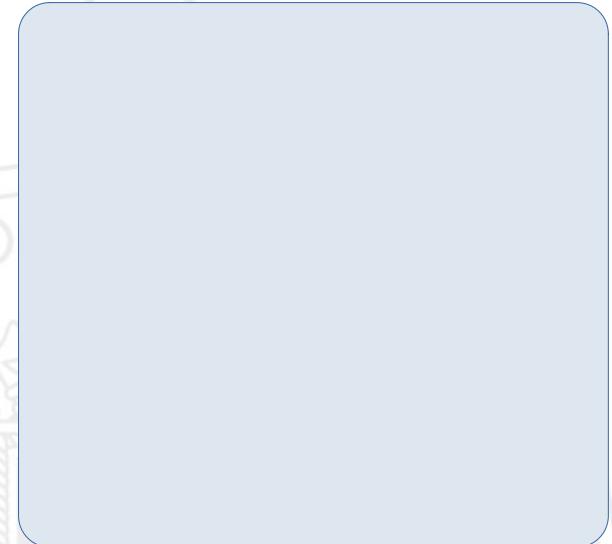
    std::cout << "Fecha 2: ";
    f2->imprimir();
    std::cout << std::endl;

    delete f2;
    return 0;
}
```

Pila



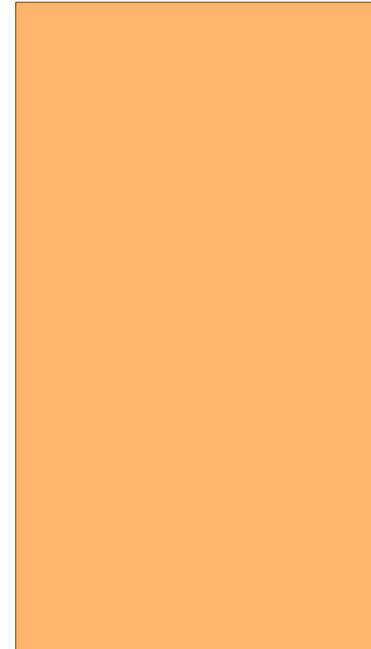
Heap



Arrays de objetos

```
int main() {  
    Fecha fs[3] =  
        { {2010}, {2011}, {2012} };  
  
    return 0;  
}
```

Pila



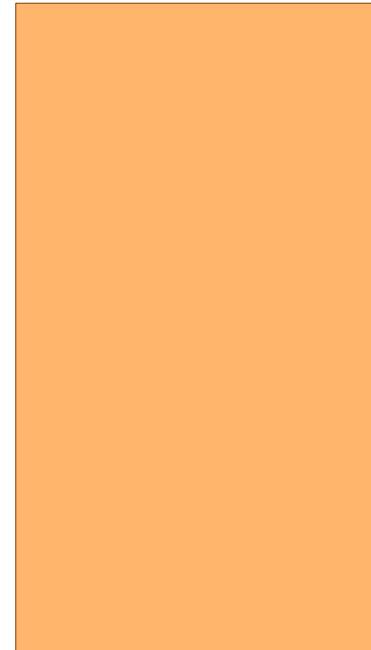
Heap



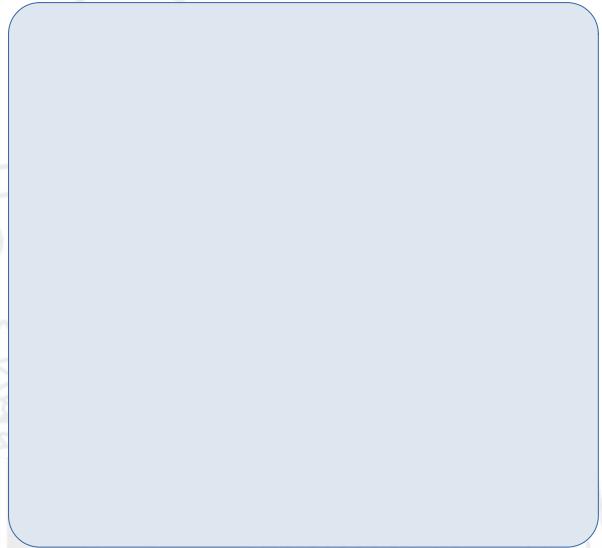
Arrays de punteros a objetos

```
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



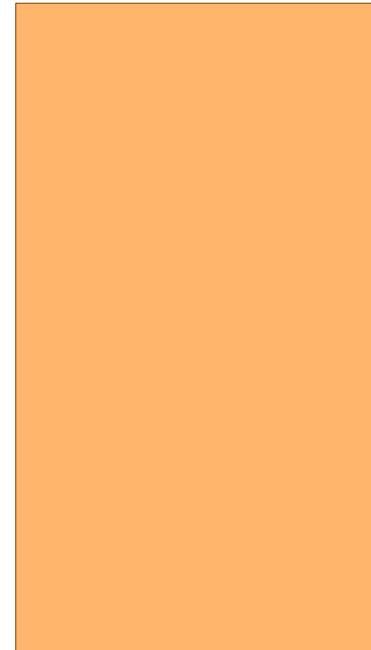
Heap



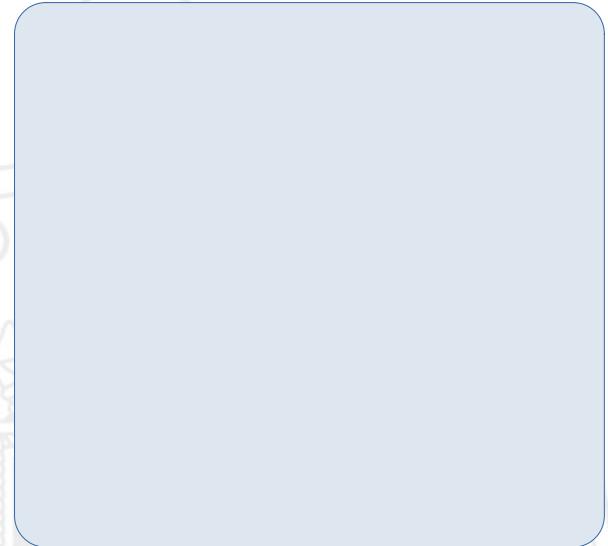
Arrays dinámicos de punteros a objetos

```
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



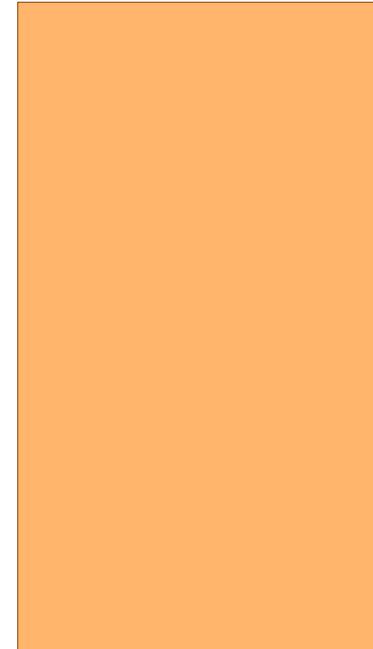
Heap



Arrays dinámicos de objetos

```
int main() {  
    Fecha *fs = new Fecha[3];  
  
    delete[] fs;  
    return 0;  
}
```

Pila



Heap



Añadiendo un constructor por defecto

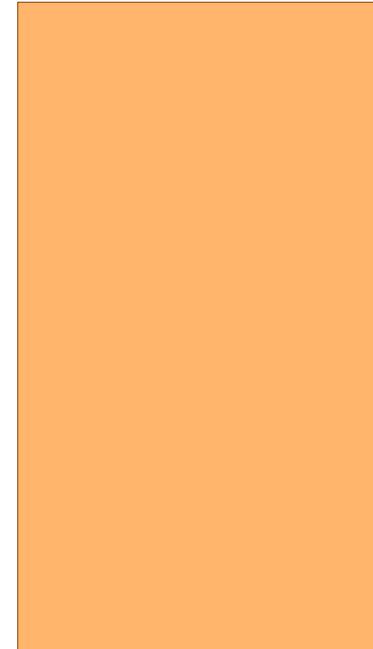
```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo);  
    Fecha(int anyo);  
    Fecha(): Fecha(1, 1, 1900) { }  
  
    int get_dia() const;  
    void set_dia(int dia);  
    int get_mes() const;  
    void set_mes(int mes);  
    int get_anyo() const;  
    void set_anyo(int anyo);  
    void imprimir();  
  
private:  
    int dia;  
    int mes;  
    int anyo;  
};
```



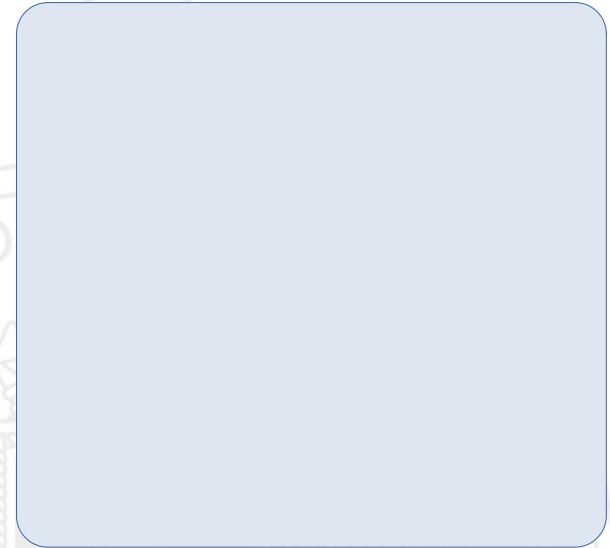
Arrays dinámicos de objetos

```
int main() {  
    Fecha *fs = new Fecha[3];  
  
    delete[] fs;  
    return 0;  
}
```

Pila



Heap

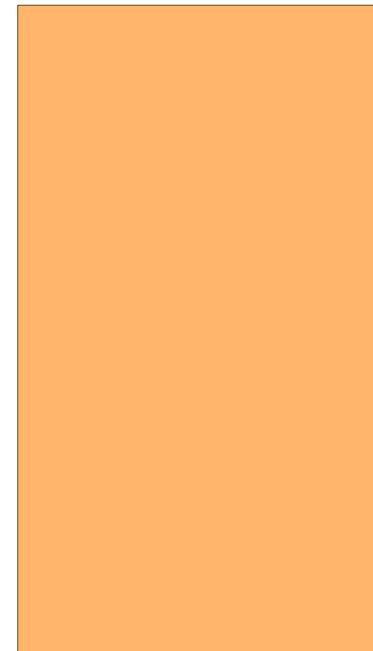


Compartición de objetos

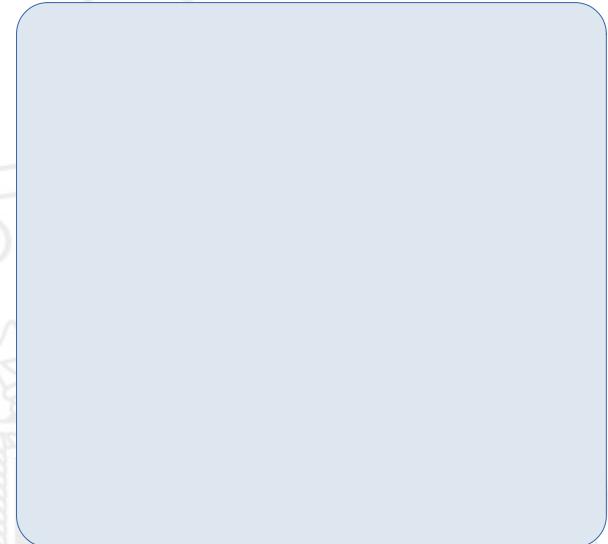
Compartición de punteros

```
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

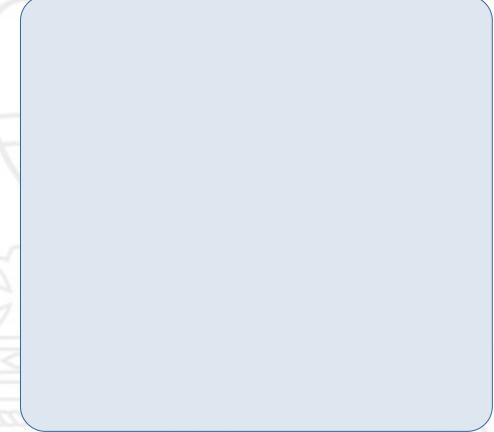


- **Todos los objetos viven en el heap.**
- La pila solo almacena valores básicos o punteros a objetos.

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;  
};
```

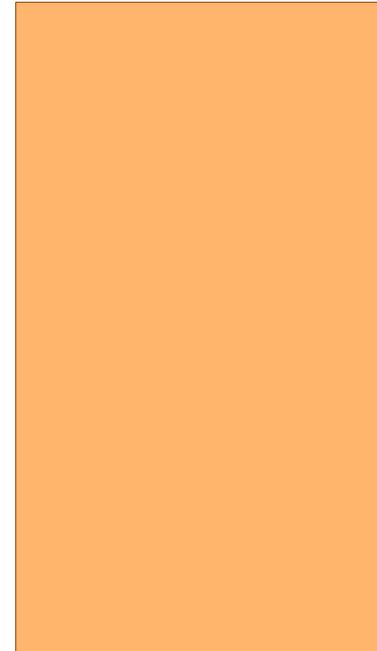


Ejemplo de uso

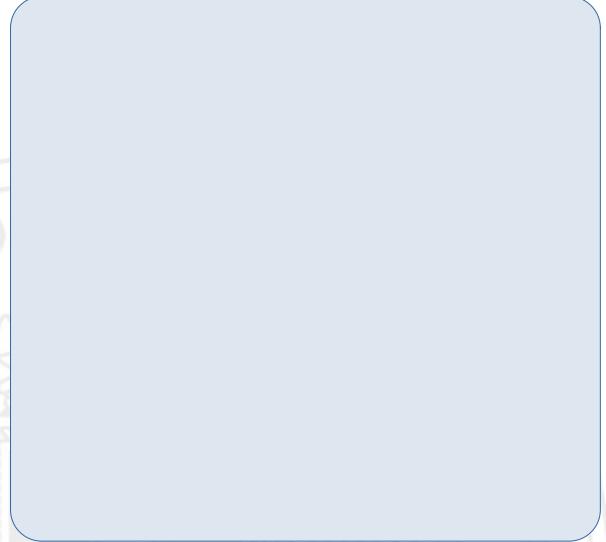
```
void ejemplo() {  
    Persona p("David", 15, 3, 1979);  
}
```

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

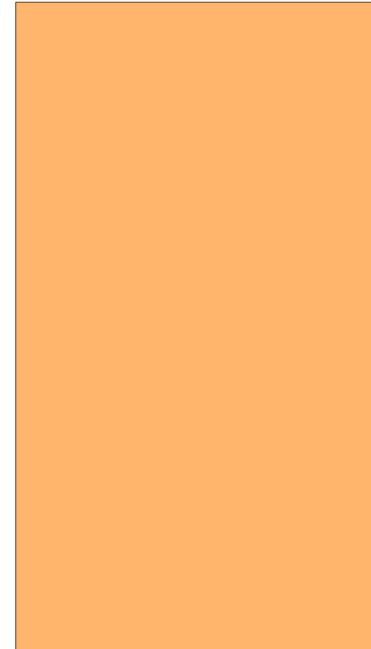
```
class Persona {  
public:  
    Persona(std::string nombre, int dia, int mes, int anyo)  
        : fecha_nacimiento(new Fecha(dia, mes, anyo)) {}  
  
    ~Persona() {  
        delete fecha_nacimiento;  
    }  
  
private:  
    std::string nombre;  
    Fecha *fecha_nacimiento;  
};
```

← Método destructor

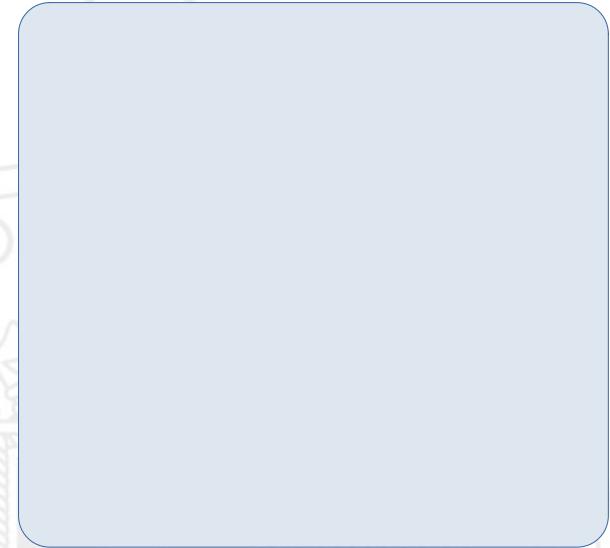
Ejemplo de uso

```
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?

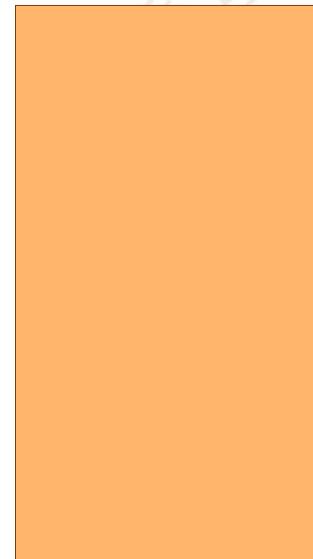
- Cuando se pasa una instancia a una función como parámetro **por valor**, se crea una copia de dicha instancia.
- **¿Cómo se realiza la copia?** Copiando el valor de cada uno de los atributos de la instancia “origen” a la instancia “destino”.

```
void modificar_copia(Persona p) { ... }

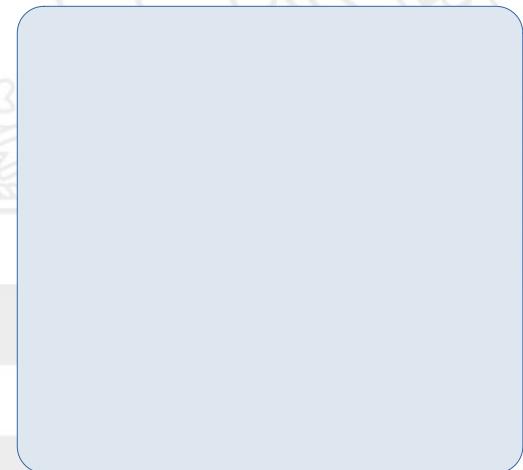
int main() {
    Persona david("David", 15, 3, 1979);
    david.imprimir();
    modificar_copia(david);
    david.imprimir();

    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

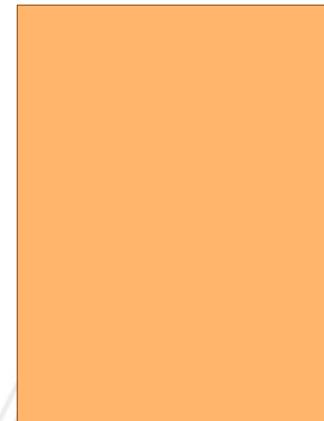
```
Persona p1("David", 15, 3, 1979);
Persona p2 = p1;
```

Sí se llama al
constructor de copia

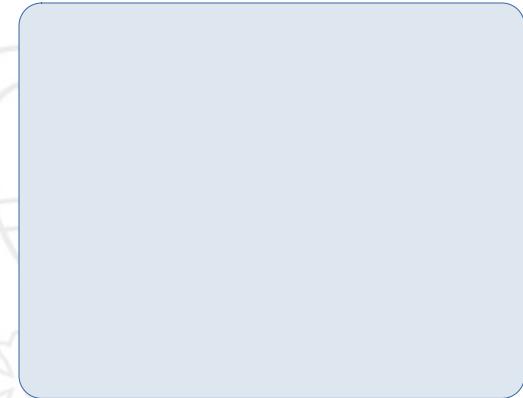
Volviendo a nuestro ejemplo...

```
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

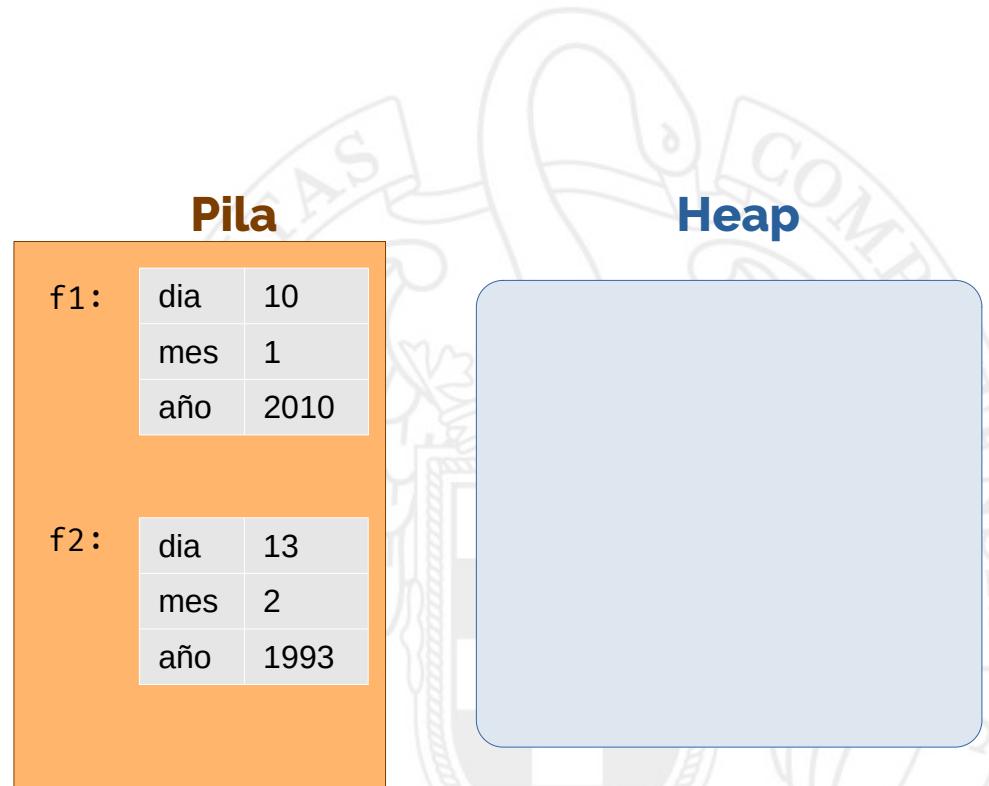
Recordatorio: clase Fecha

```
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

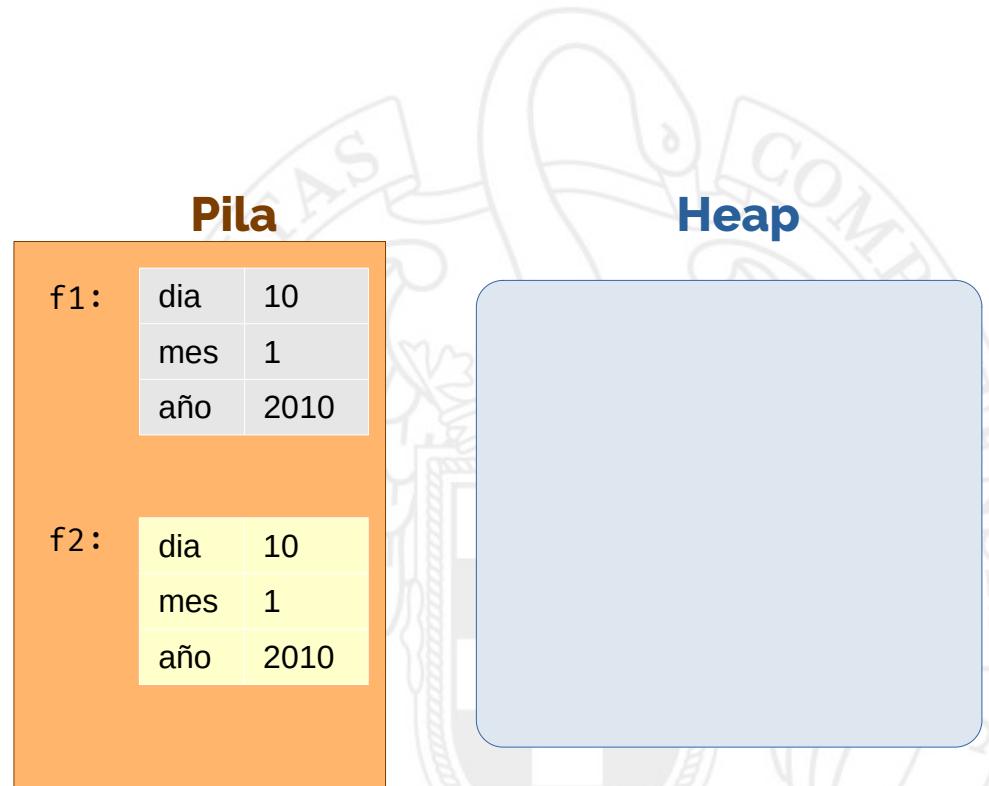
```
Fecha f1(10, 1, 2010);  
Fecha f2(13, 2, 1993);  
  
f2 = f1;
```



Asignar un objeto Fecha a otro

```
Fecha f1(10, 1, 2010);  
Fecha f2(13, 2, 1993);
```

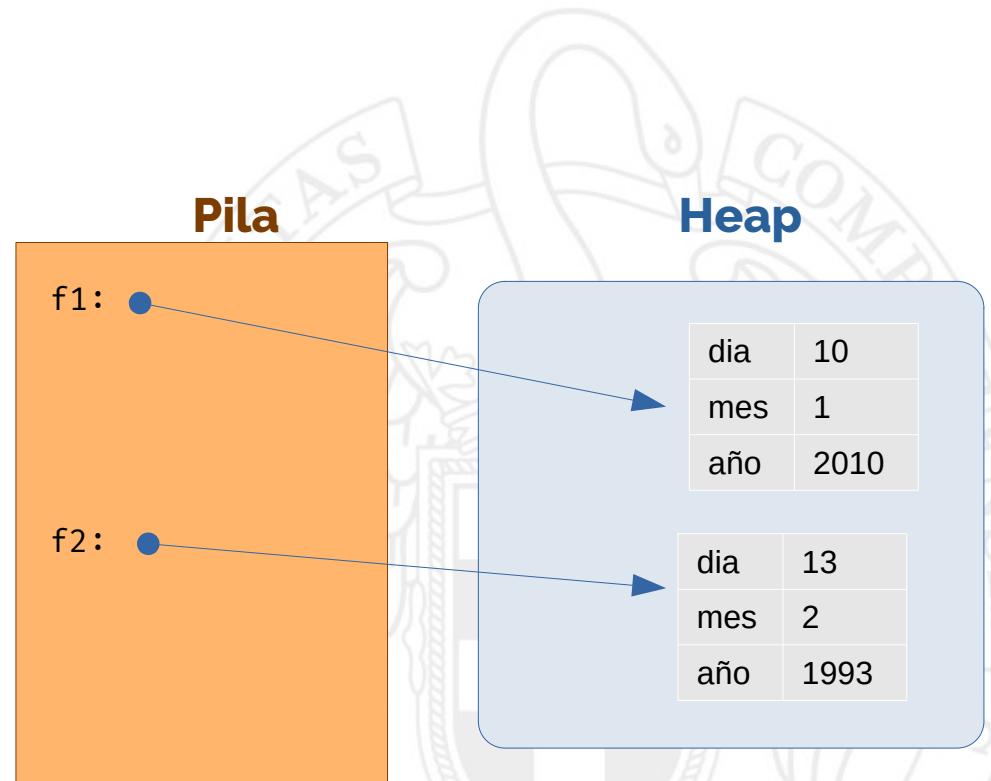
```
f2 = f1;
```



Asignar un objeto Fecha a otro

```
Fecha *f1 = new Fecha(10, 1, 2010);  
Fecha *f2 = new Fecha(13, 2, 1993);
```

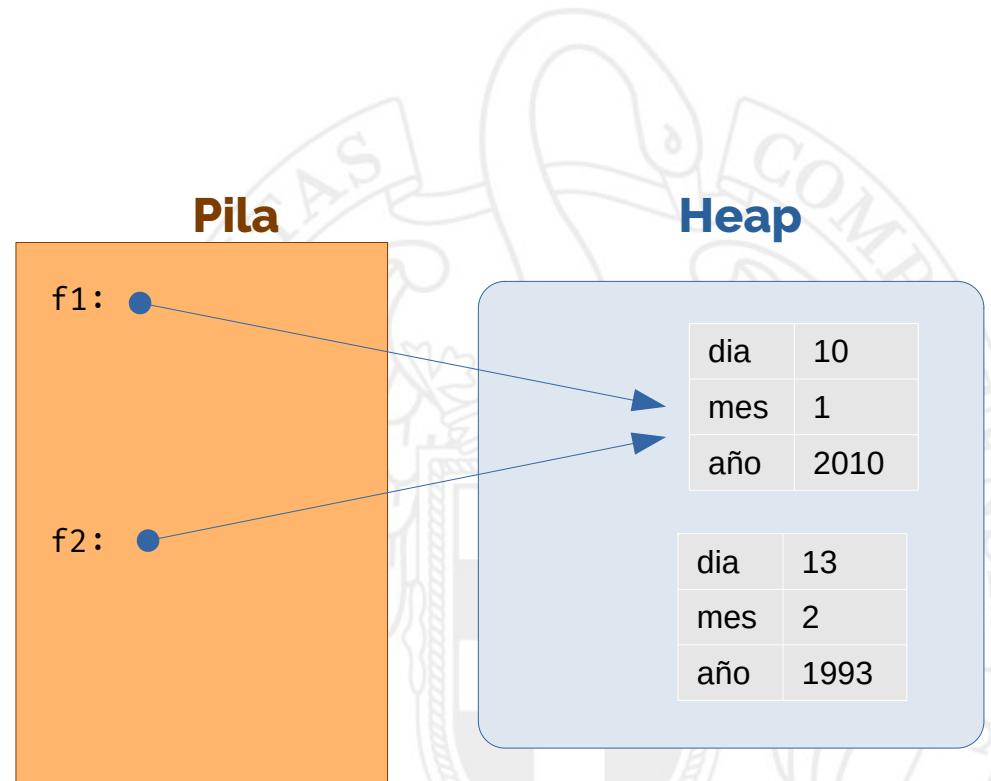
```
f2 = f1;
```



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

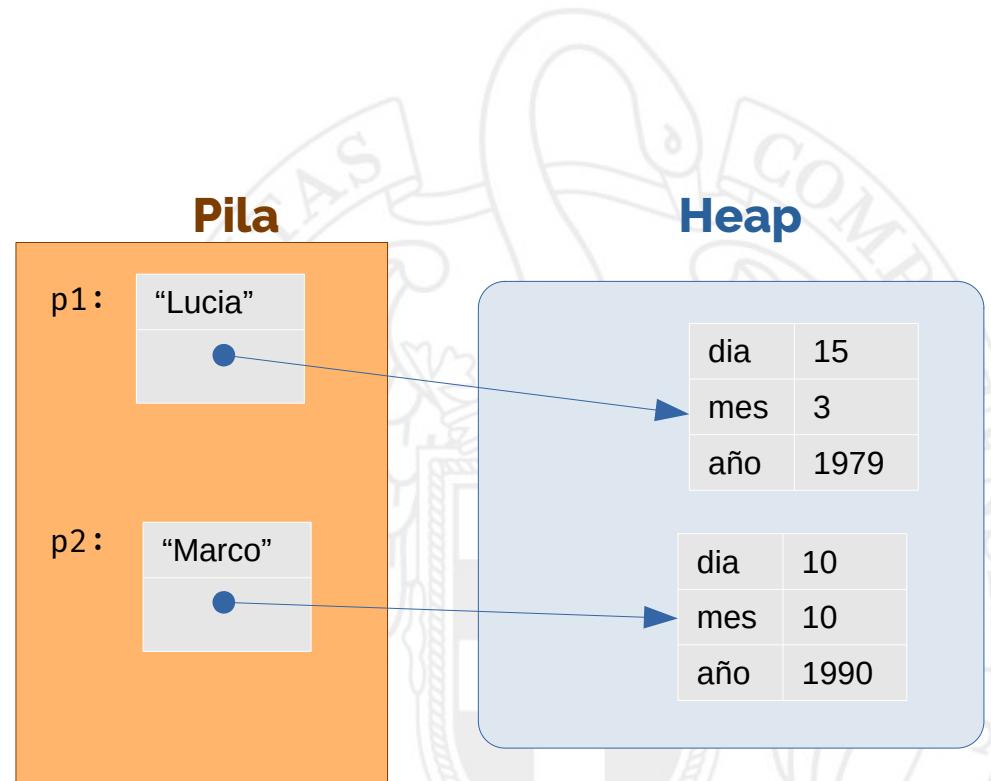
```
class Persona {  
public:  
    Persona(std::string nombre,  
            int dia,  
            int mes,  
            int anyo);  
  
    ~Persona() {  
        delete fecha_nacimiento;  
    }  
  
    ...  
  
private:  
    std::string nombre;  
    Fecha *fecha_nacimiento;  
};
```



Asignar un objeto Persona a otro

```
Persona p1("Lucía", 15, 3, 1979);  
Persona p2("Marco", 10, 10, 1990);
```

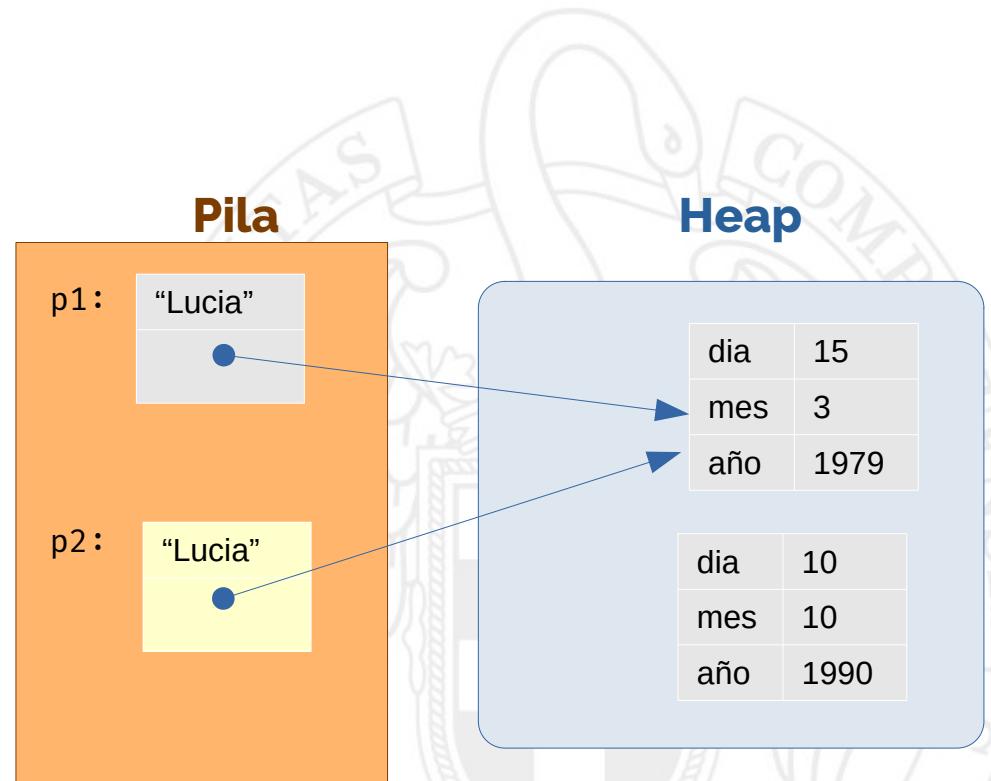
```
p2 = p1;
```



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

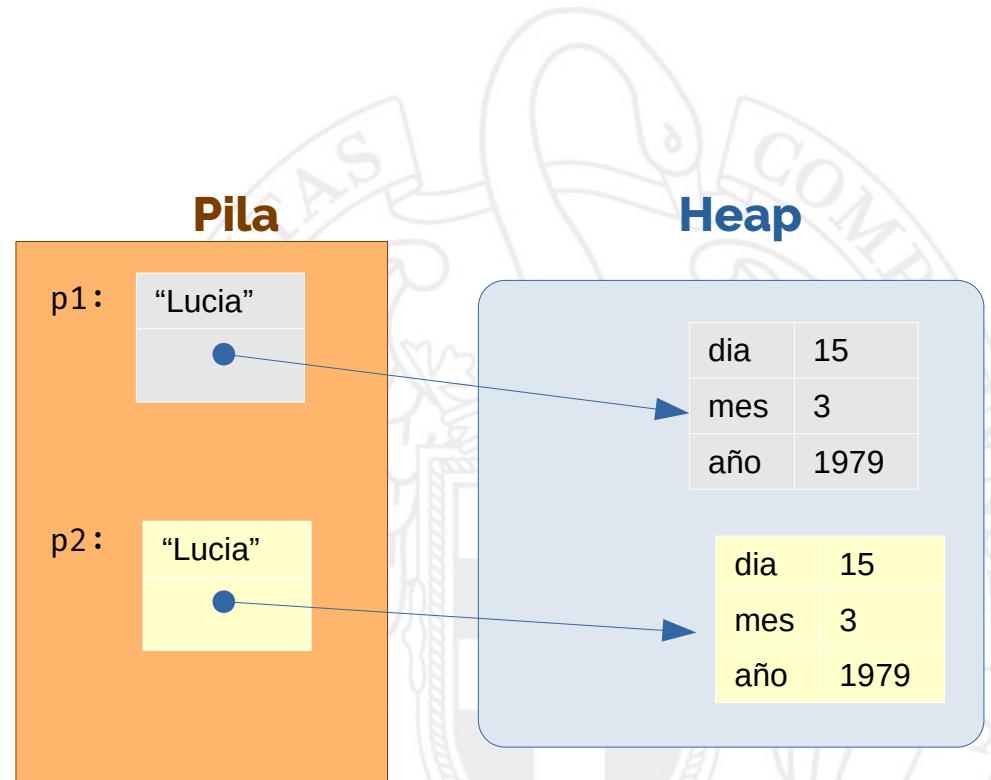
```
class Persona {  
public:  
    ...  
  
    void operator=(const Persona &other) {  
        nombre = other.nombre;  
        fecha_nacimiento→set_dia(other.fecha_nacimiento→get_dia());  
        fecha_nacimiento→set_mes(other.fecha_nacimiento→get_mes());  
        fecha_nacimiento→set_anyo(other.fecha_nacimiento→get_anyo());  
    }  
  
    ...  
  
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;
```



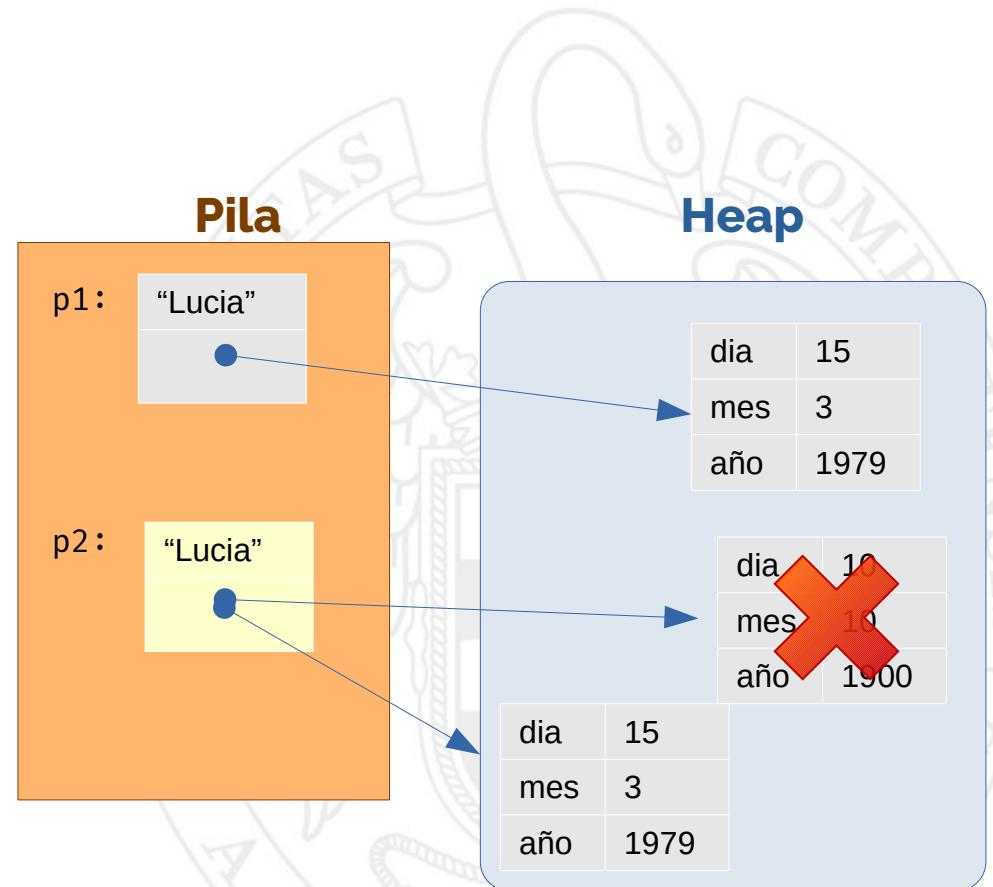
Otra posibilidad

```
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;
```

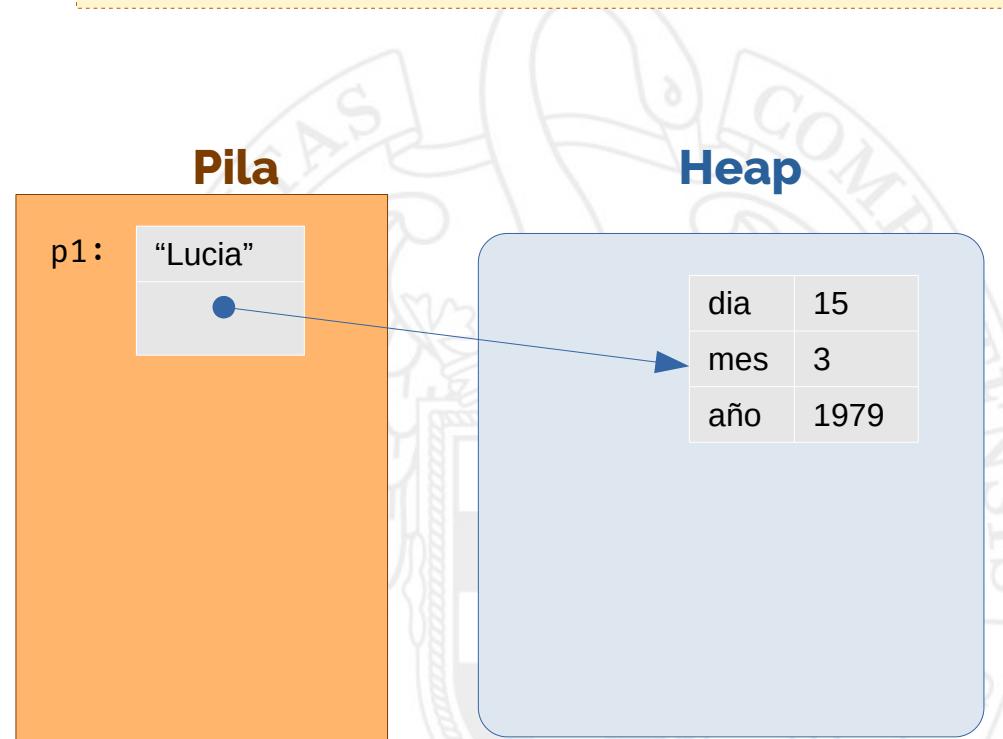


El problema de la autoasignación

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;
```

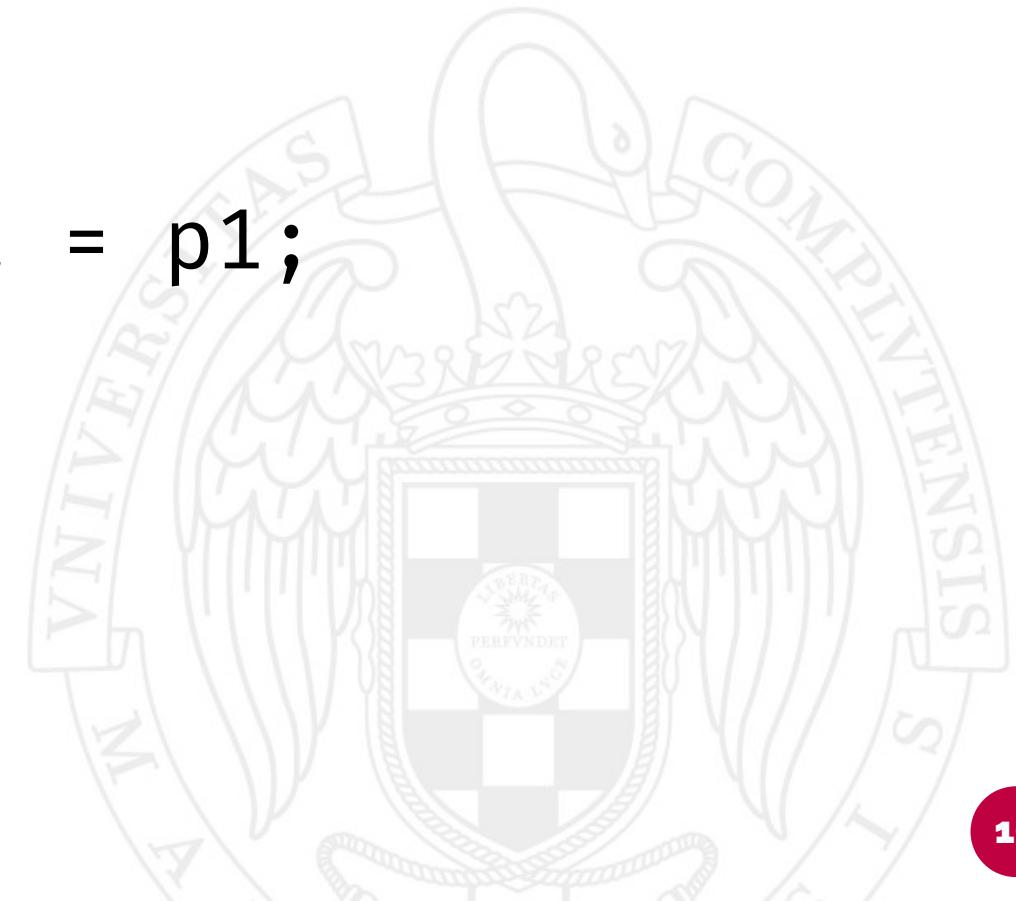
x = y = z = 0;

¿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;  
};
```

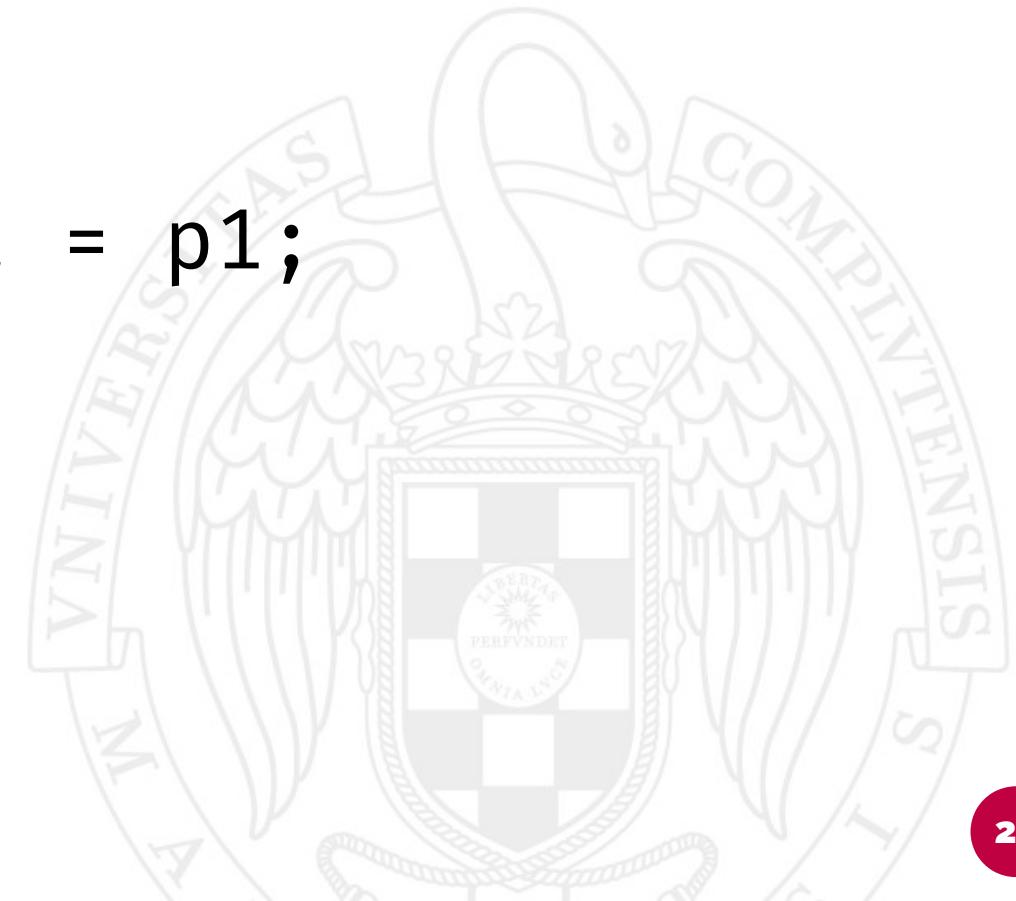


¿Podemos hacer lo mismo con objetos Persona?

```
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*

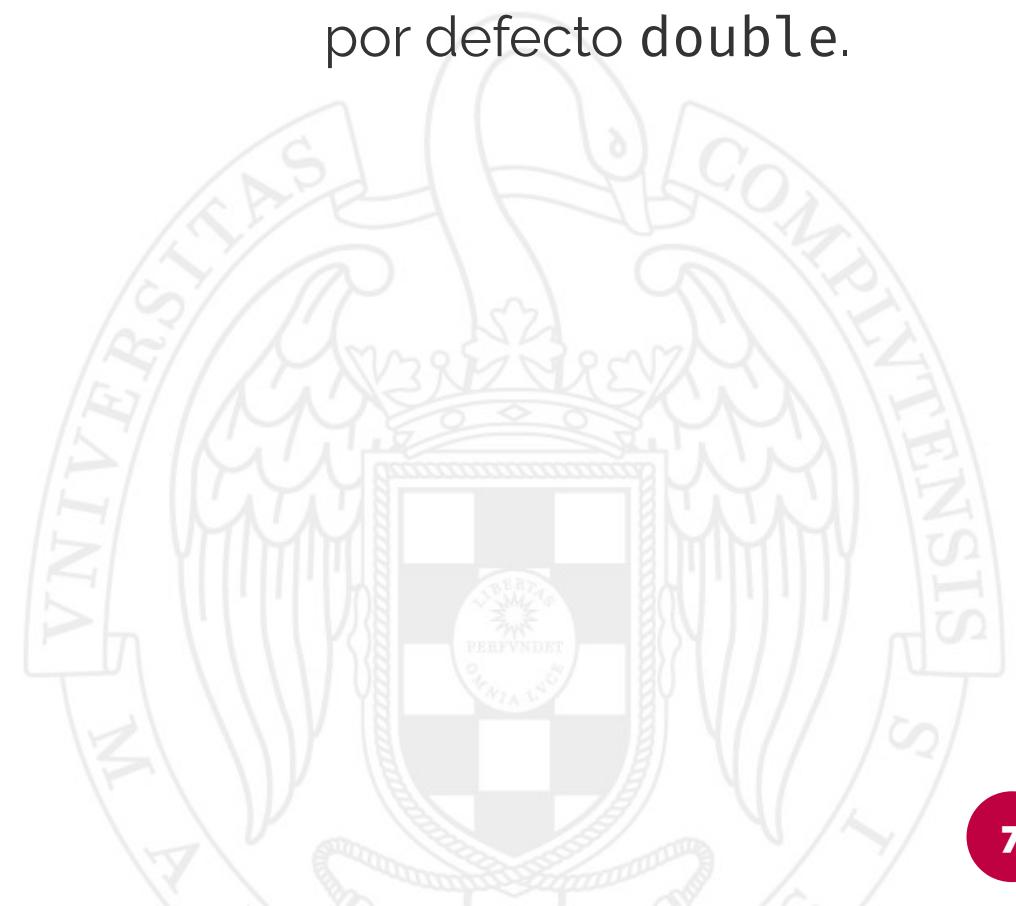
Complejo z4(2.0, -3.0); 

- ... 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:

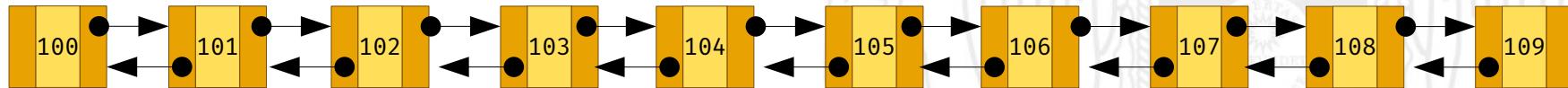
- `source_begin`, `source_end` son iteradores de entrada.
 - `destination_begin` es iterador de salida.
- Copia el intervalo de elementos delimitado por `source_begin` y `source_end` (excluyendo este último), a la posición apuntada por el iterador `destination_begin`.

Ejemplo

```
int main() {
    vector<int> origen;
    list<int> destino;

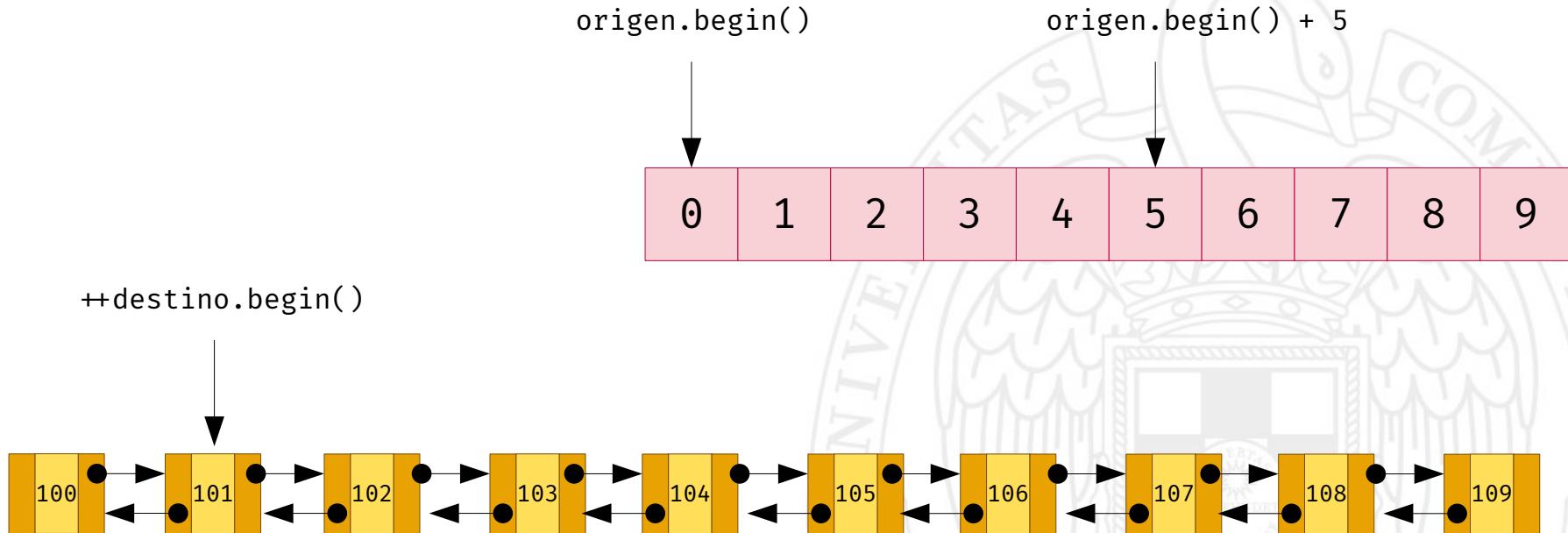
    for (int i = 0; i < 10; i++) {
        origen.push_back(i);
        destino.push_back(100 + i);
    }
    ...
}
```

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---



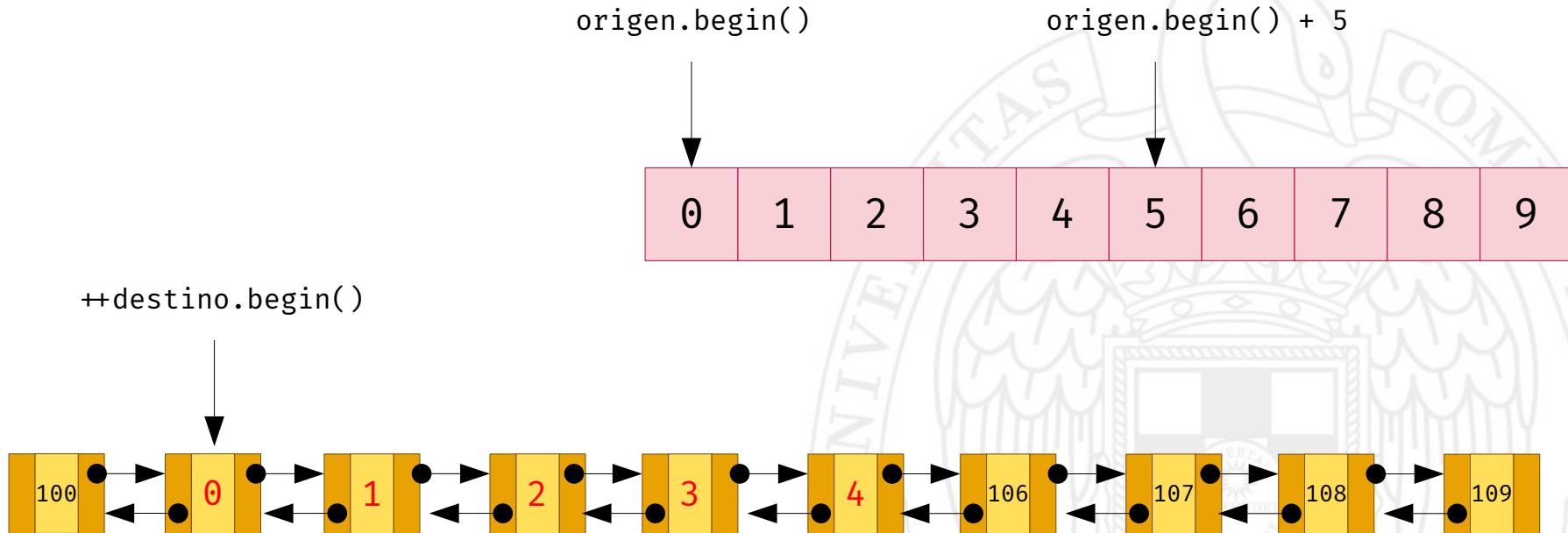
Ejemplo

```
copy(origen.begin(), origen.begin() + 5, ++destino.begin());
```



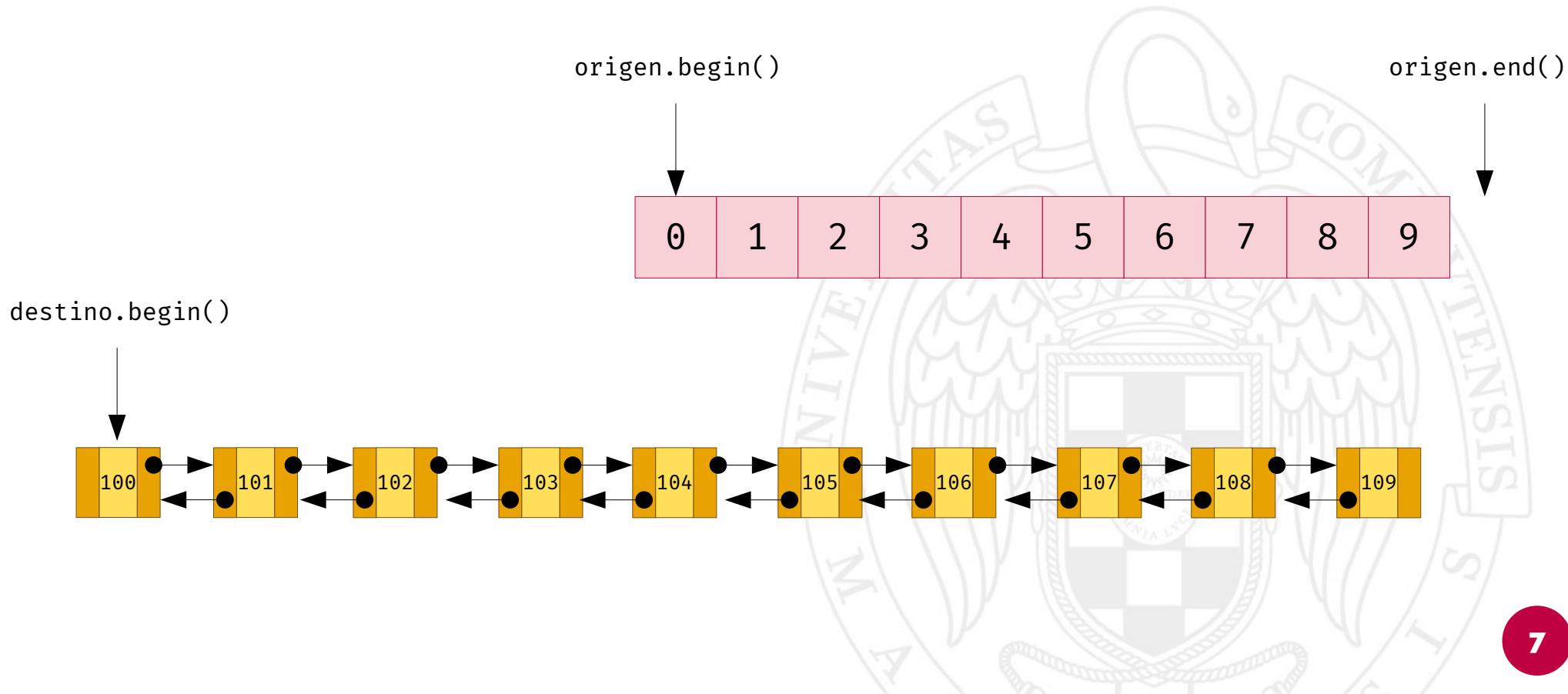
Ejemplo

```
copy(origen.begin(), origen.begin() + 5, ++destino.begin());
```



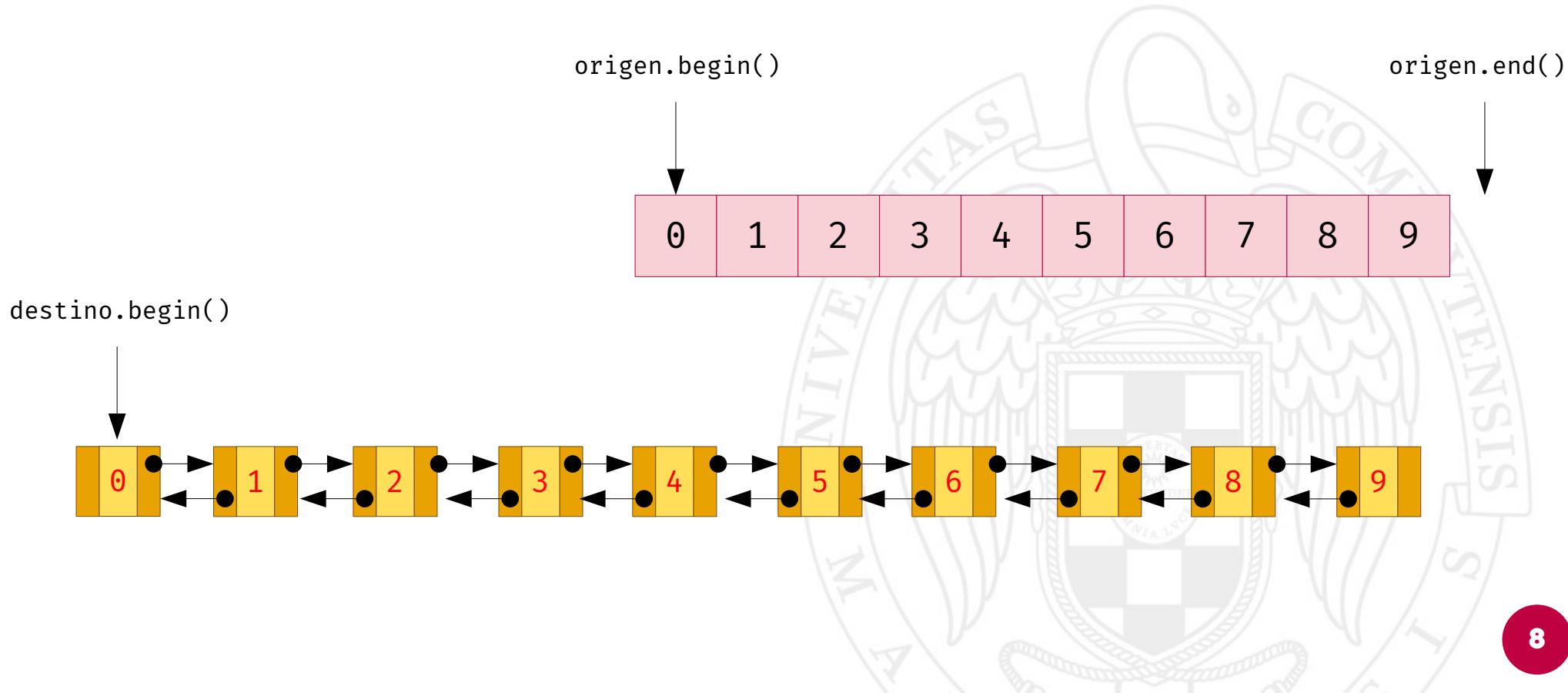
Ejemplo

```
copy(origen.begin(), origen.end(), destino.begin());
```



Ejemplo

```
copy(origen.begin(), origen.end(), destino.begin());
```

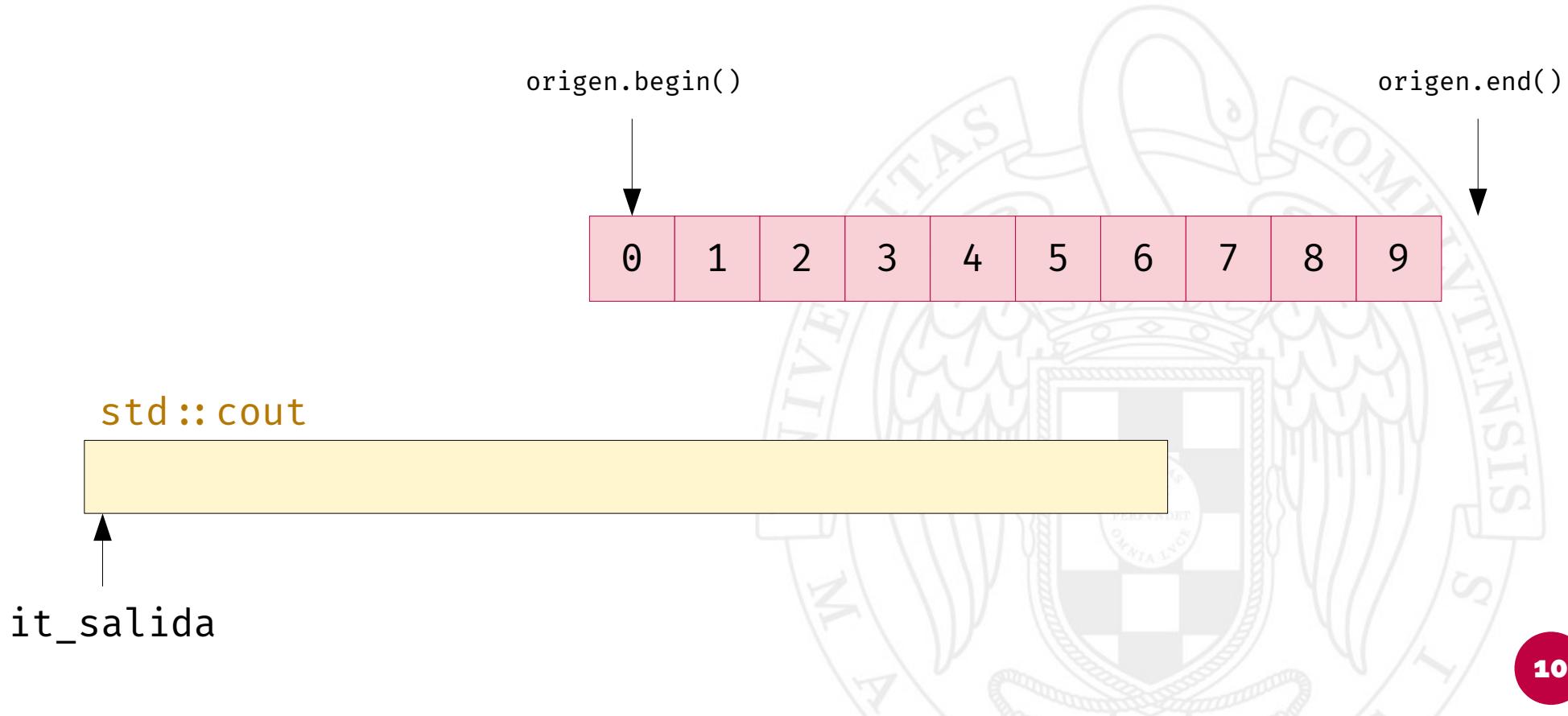


Utilidad de función `copy()`

- 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);
```

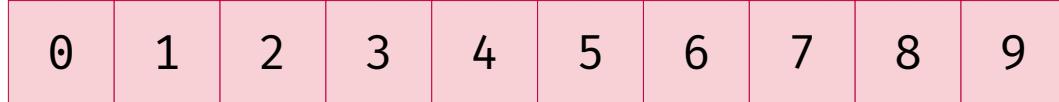


Otro ejemplo

```
ostream_iterator<int> it_salida(cout, " ");
copy(origen.begin(), origen.end(), it_salida);
```

origen.begin()

origen.end()



std::cout

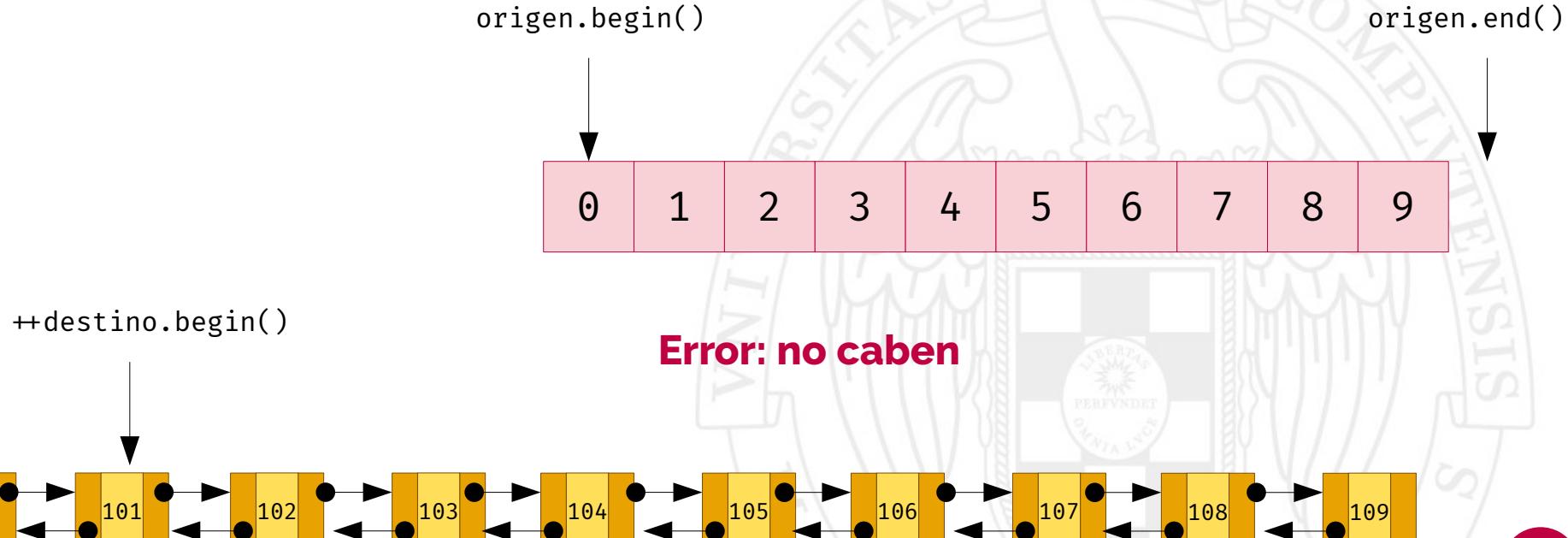
0_1_2_3_4_5_6_7_8_9_

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

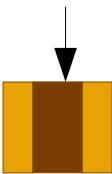
```
int main() {
    vector<int> origen;
    list<int> lista_destino;

    // 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

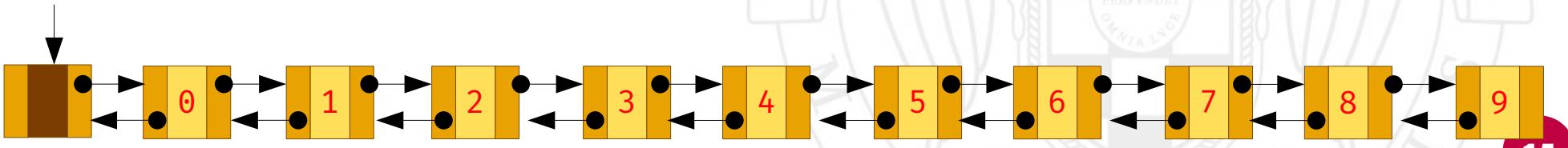
```
int main() {
    vector<int> origen;
    list<int> lista_destino;

    // 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.

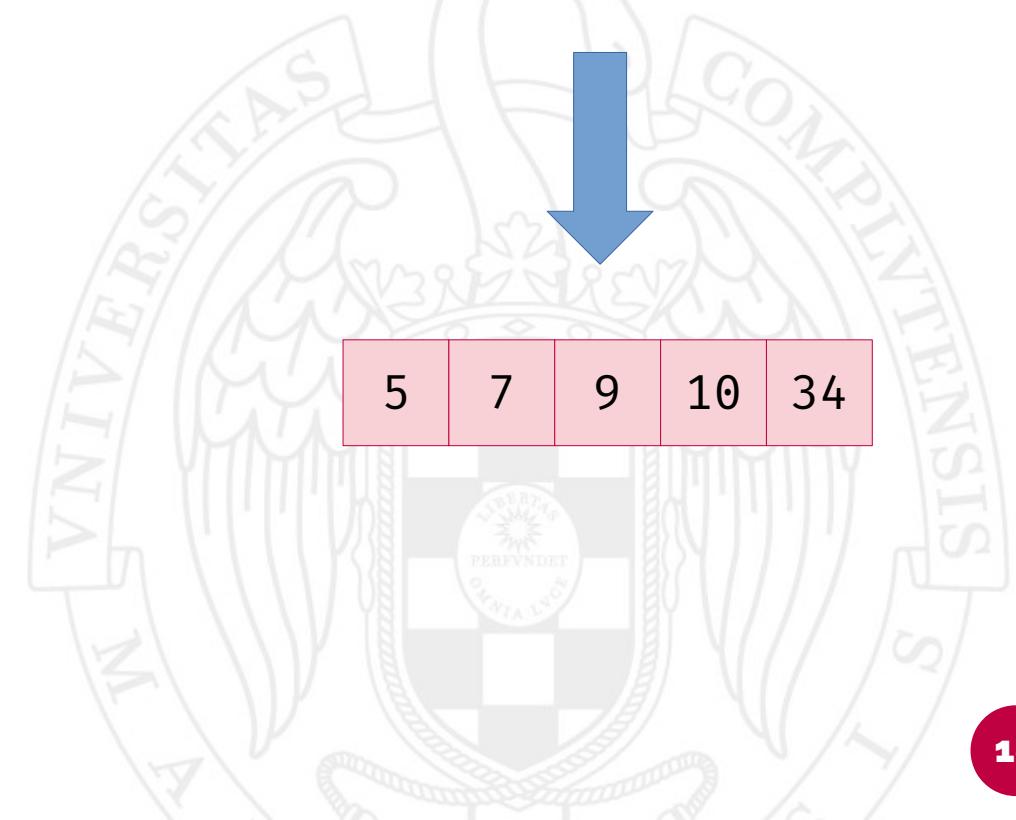
Ejemplo

```
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 *`.

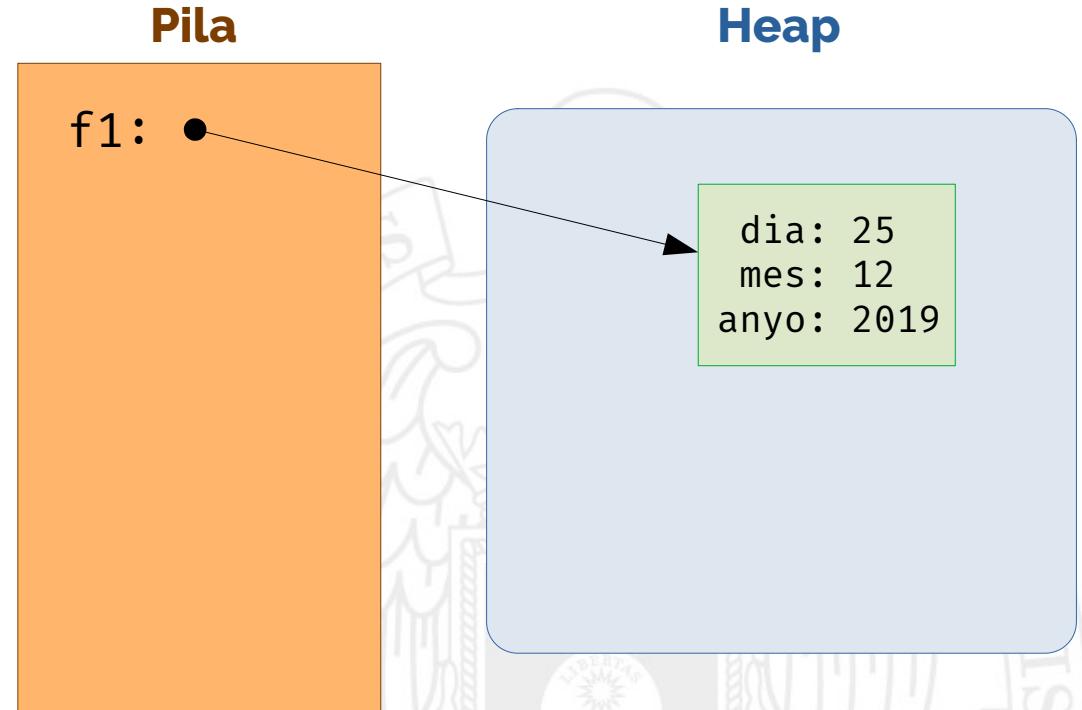
- Ejemplo: crear un objeto en el heap mediante un puntero exclusivo:

```
std::make_unique<Fecha>(25, 12, 2019)
```

Esto devuelve un objeto de tipo `std::unique_ptr<Fecha>`.

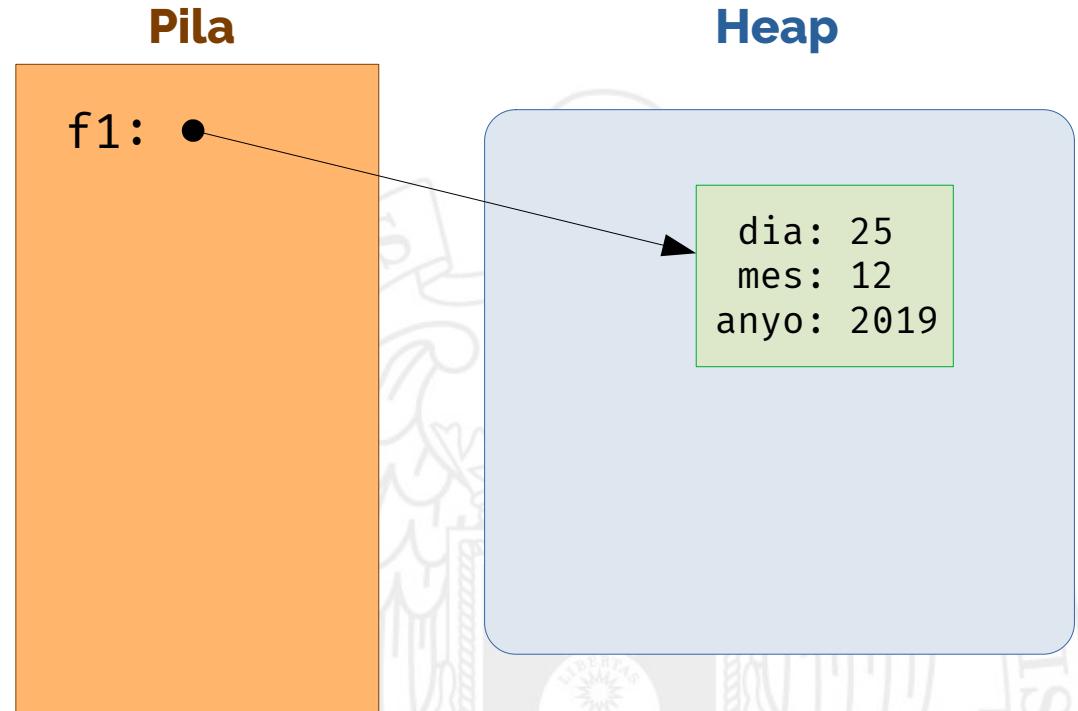
Ejemplo

```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);
```



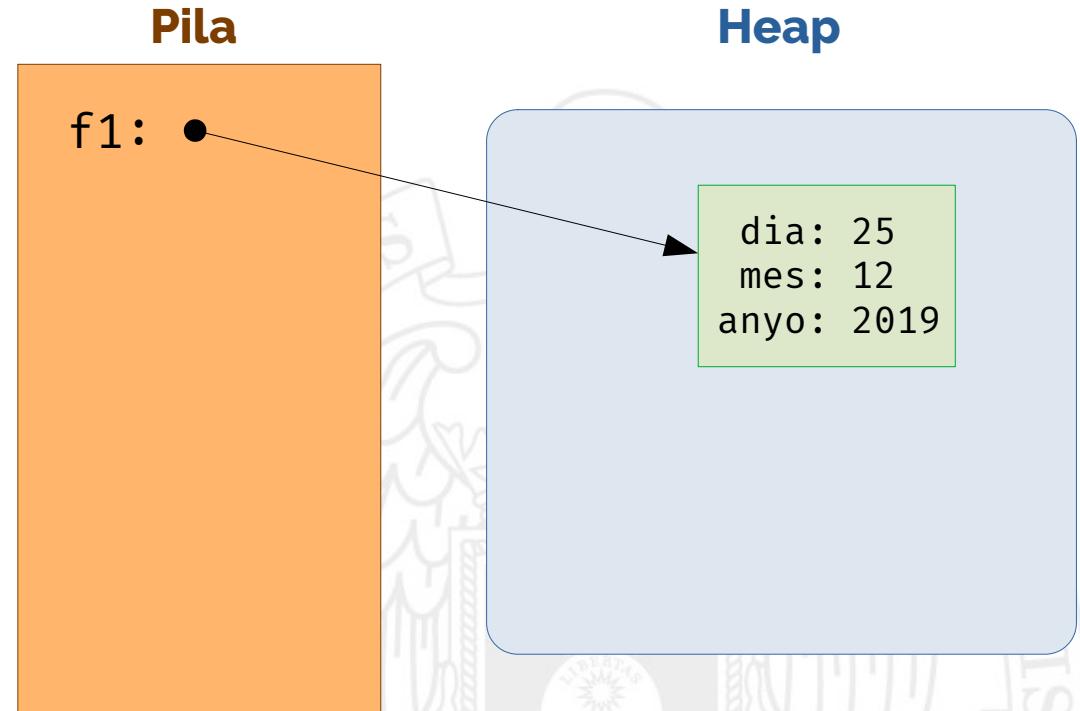
Ejemplo

```
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

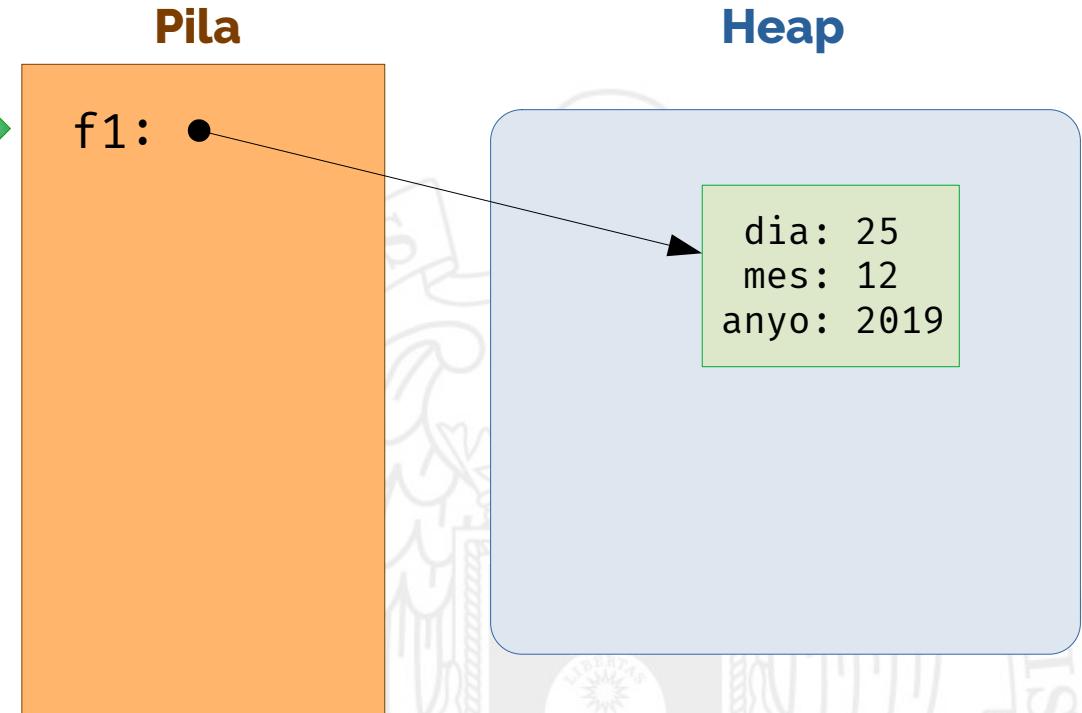
```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);  
  
std::unique_ptr<Fecha> f2 = f1; 
```



Un unique_ptr puede ser transferido

```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);
```

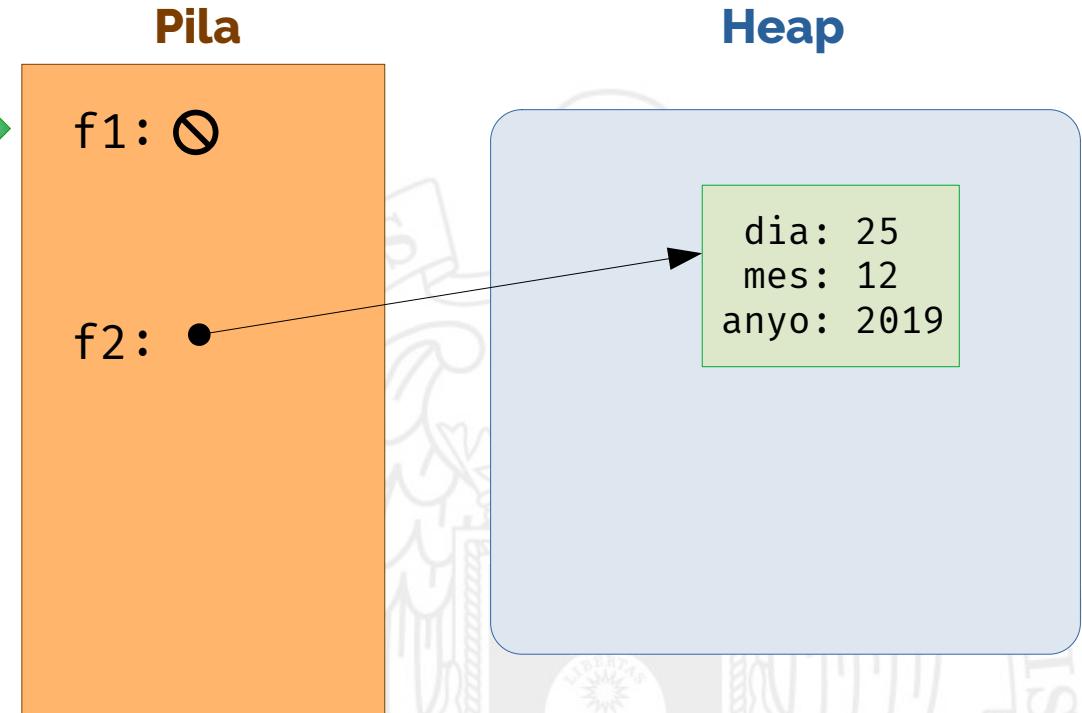
```
std::unique_ptr<Fecha> f2 = std::move(f1); ✓
```



Un unique_ptr puede ser transferido

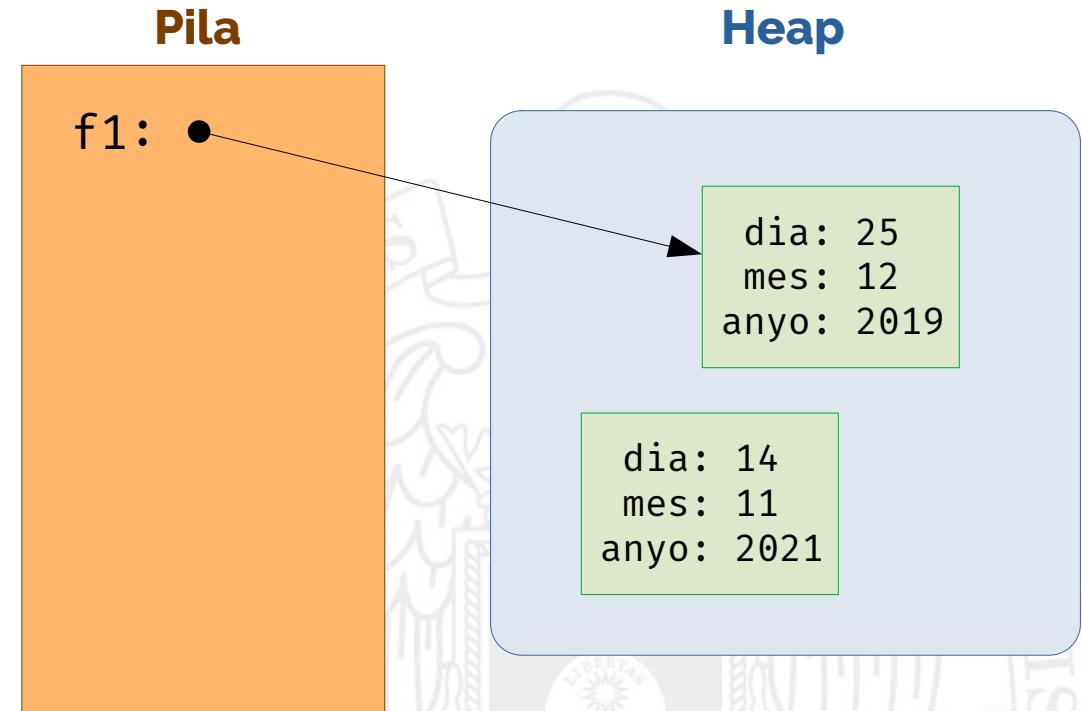
```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);
```

```
std::unique_ptr<Fecha> f2 = std::move(f1); ✓
```



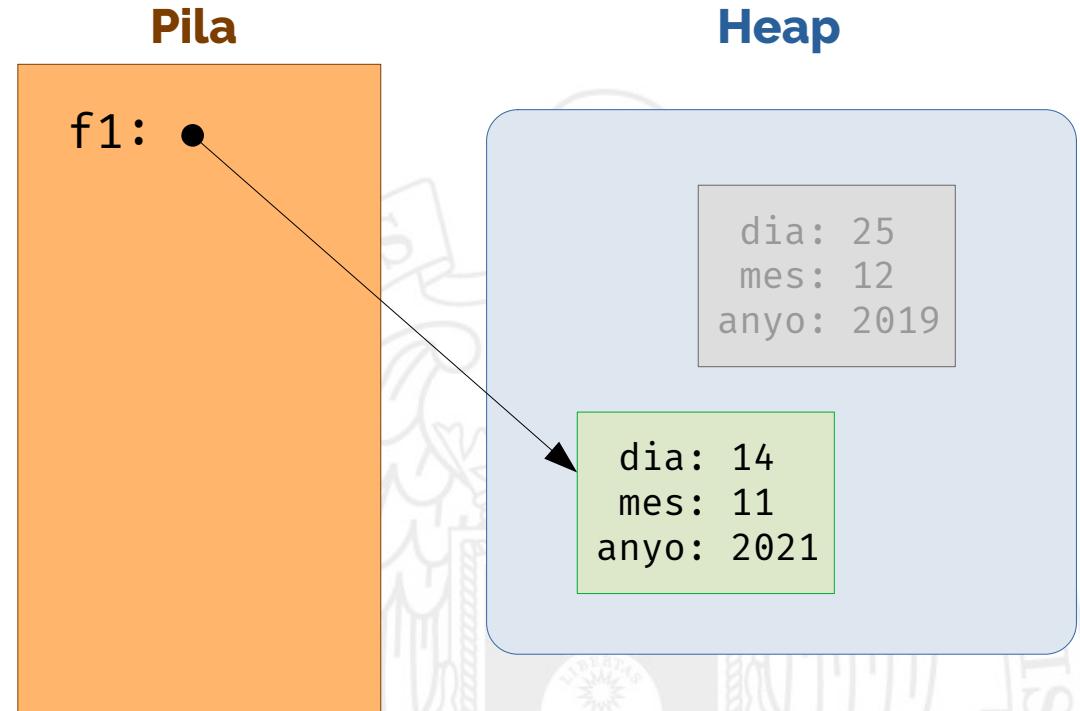
Reasignando un unique_ptr

```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);  
  
f1 = std::make_unique<Fecha>(14, 11, 2021);
```



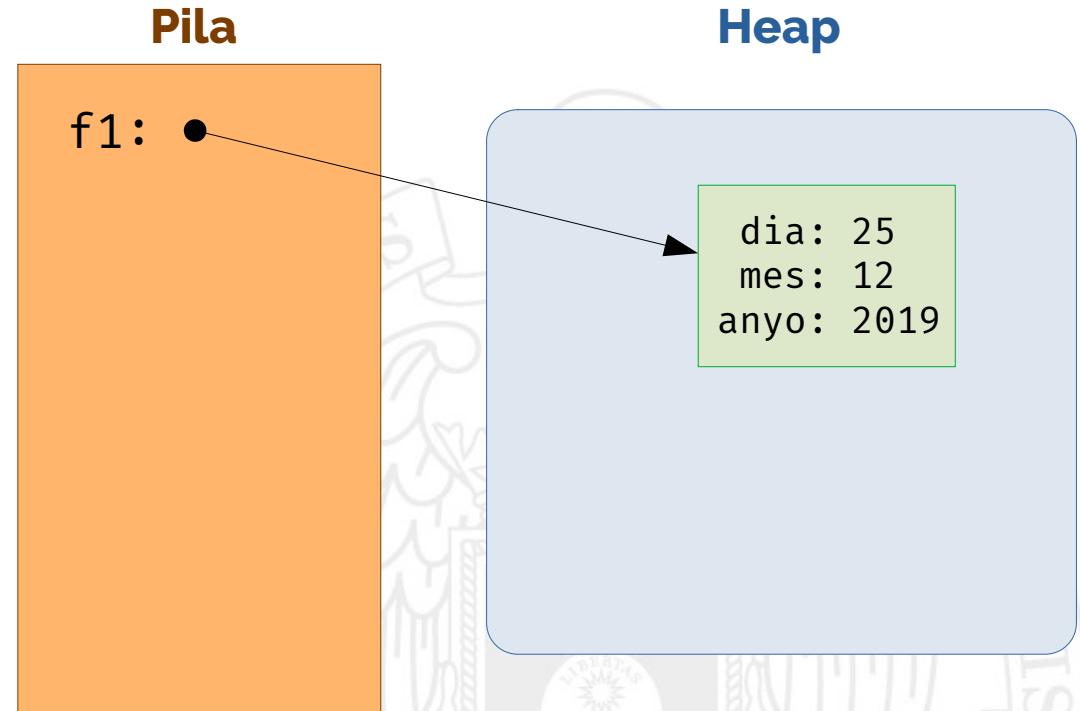
Reasignando un unique_ptr

```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);  
  
f1 = std::make_unique<Fecha>(14, 11, 2021);
```



Reasignando un unique_ptr

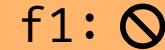
```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);  
  
f1 = nullptr;
```



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:

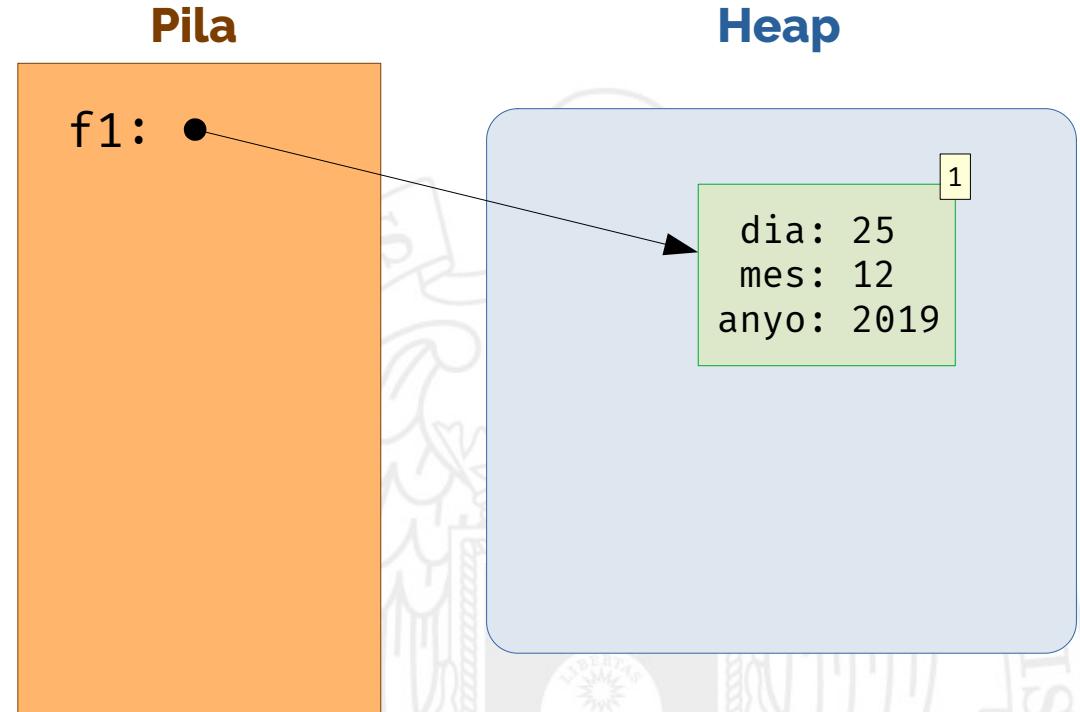
```
std :: make_shared<Fecha>(25, 12, 2019)
```

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.

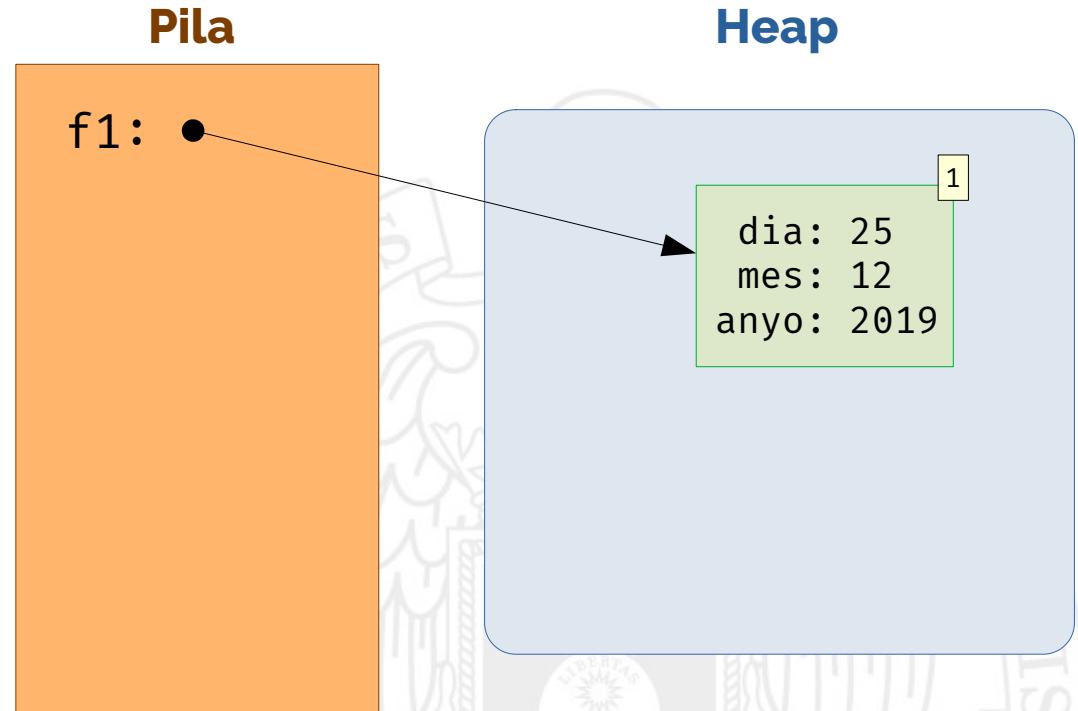
Ejemplo

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);
```



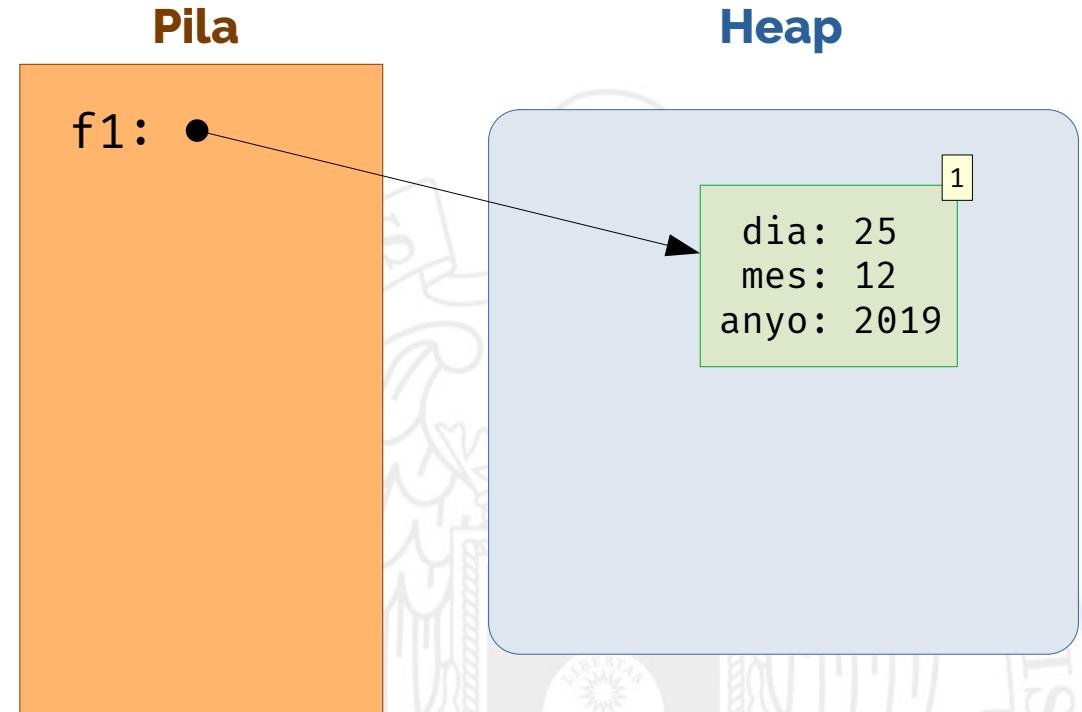
Ejemplo

```
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;  
}
```



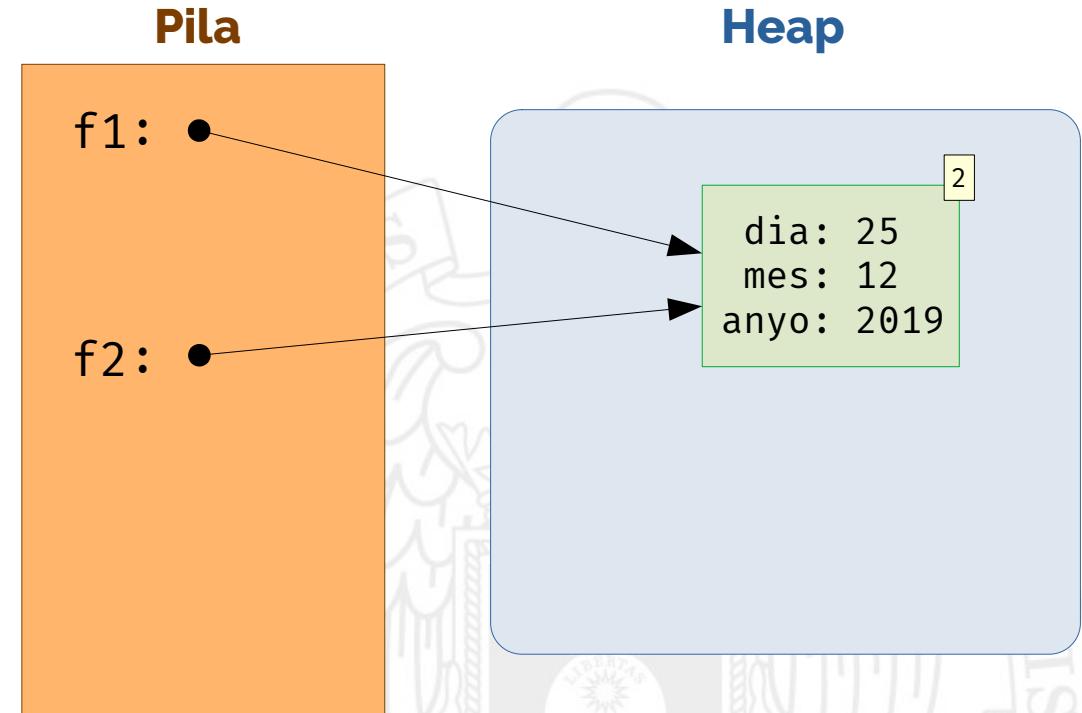
Copia de un shared_ptr

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
std::shared_ptr<Fecha> f2 = f1;
```



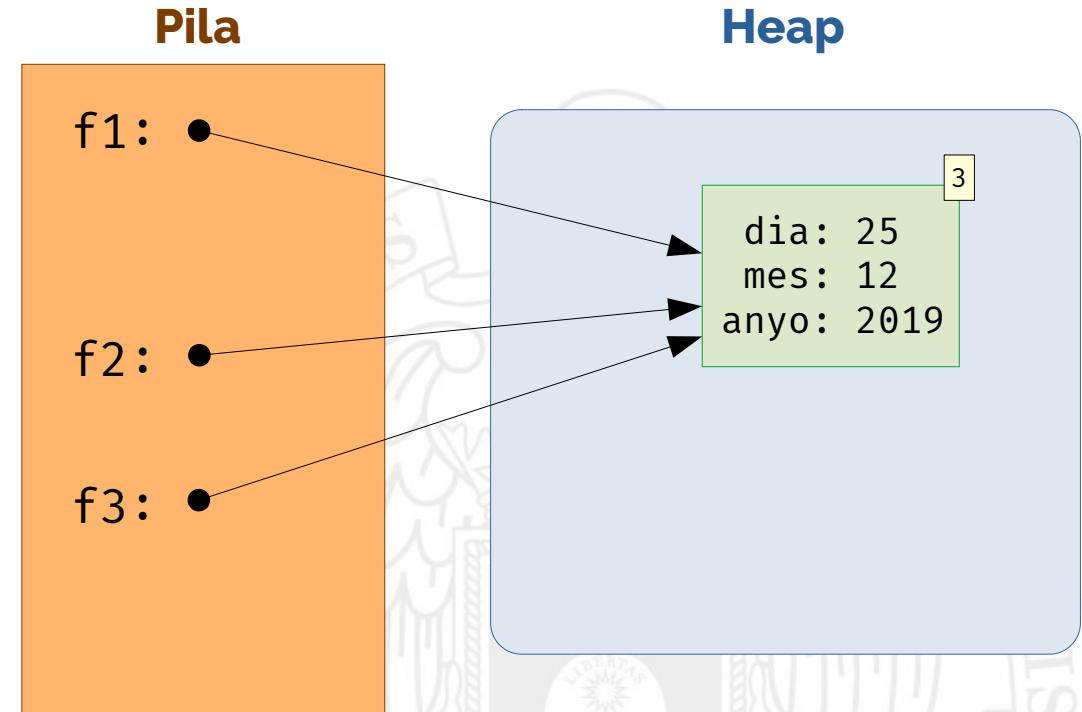
Copia de un shared_ptr

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
std::shared_ptr<Fecha> f2 = f1;
```



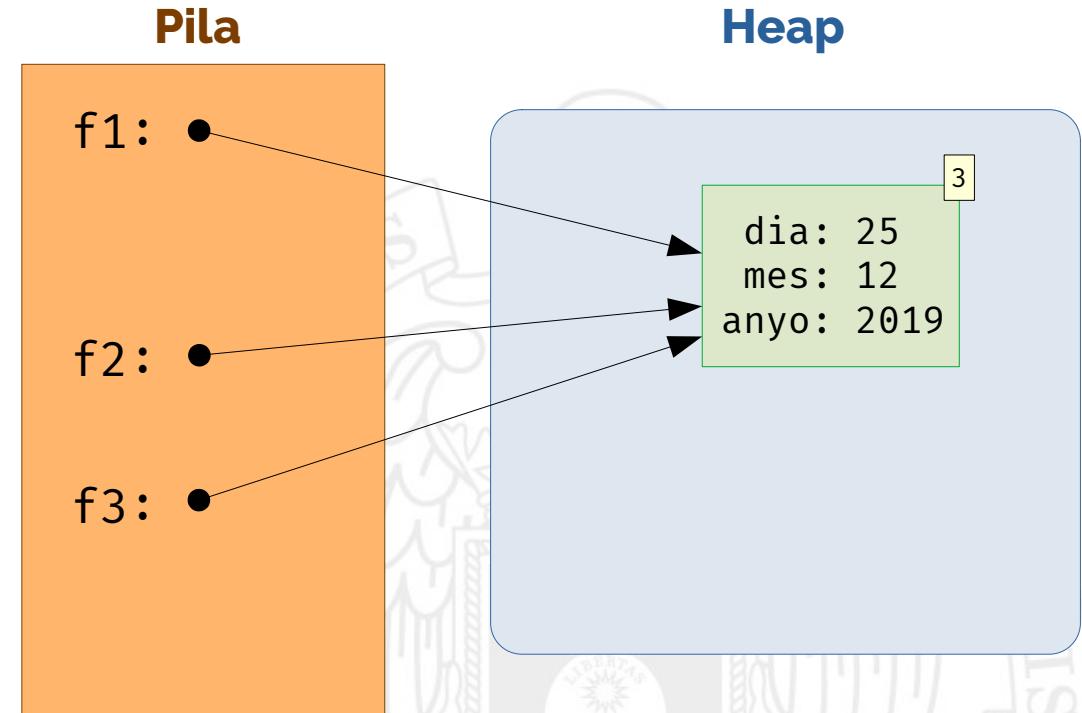
Copia de un shared_ptr

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
std::shared_ptr<Fecha> f2 = f1;  
std::shared_ptr<Fecha> f3 = f2;
```



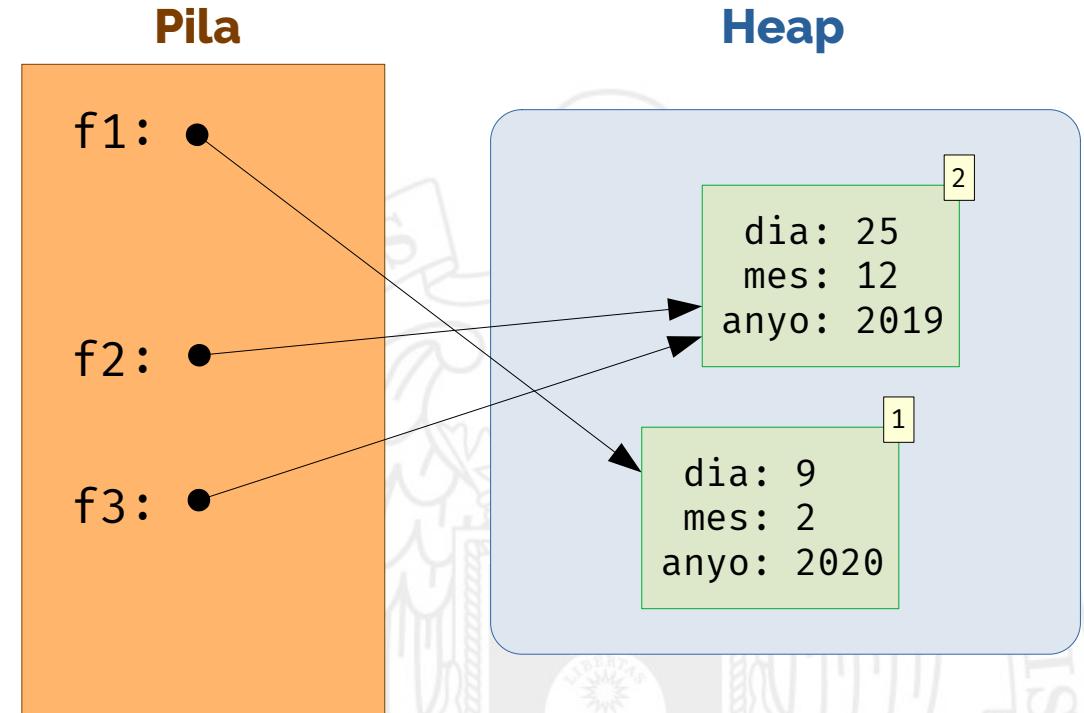
Copia de un shared_ptr

```
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);
```



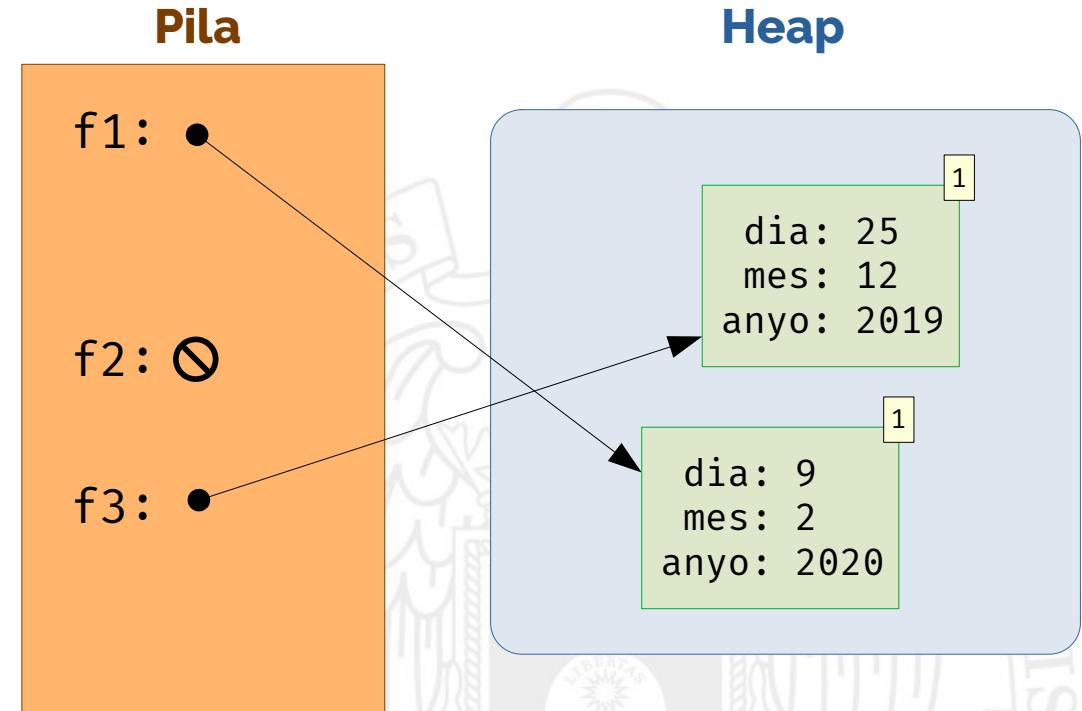
Copia de un shared_ptr

```
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);
```



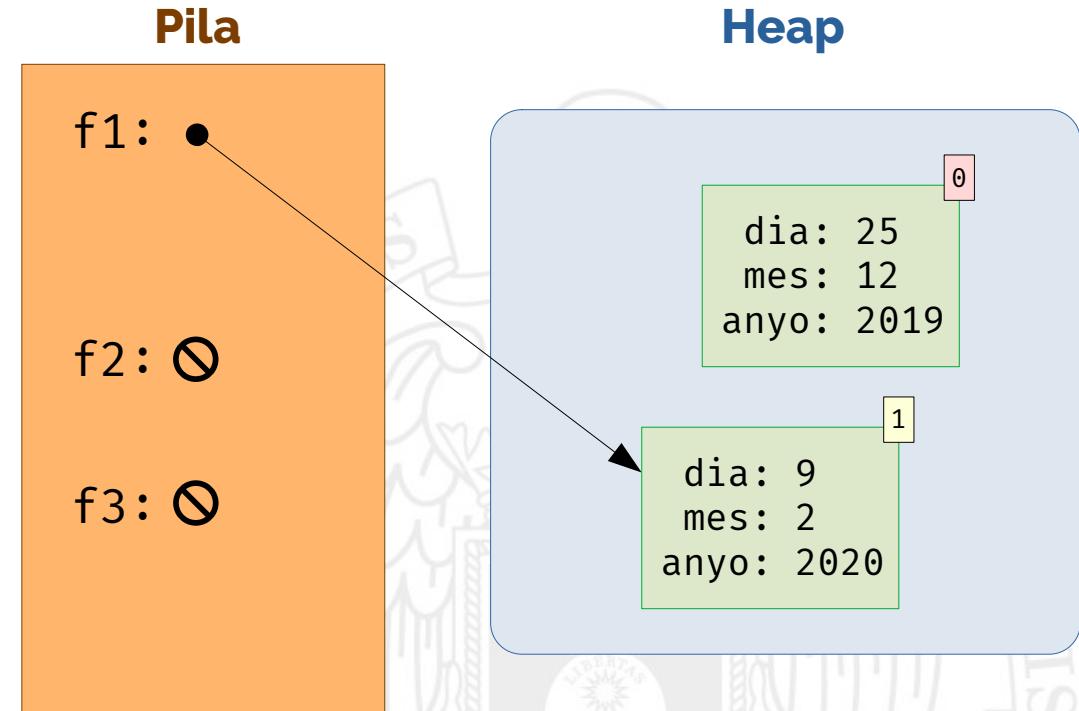
Copia de un shared_ptr

```
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;
```



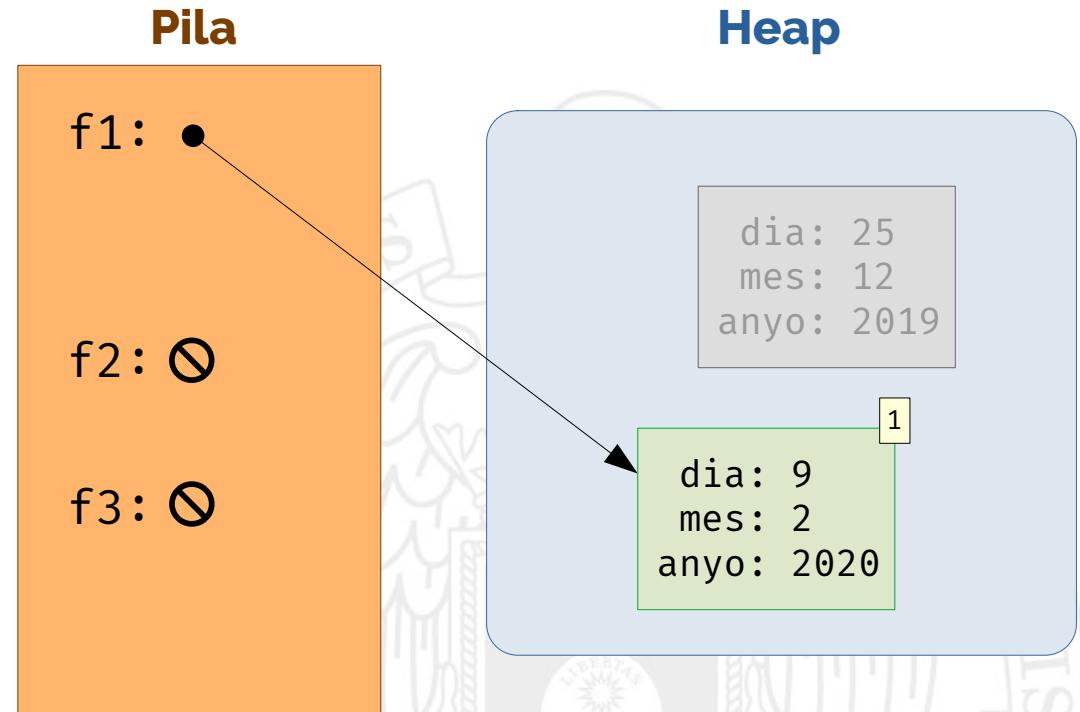
Copia de un shared_ptr

```
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;
```

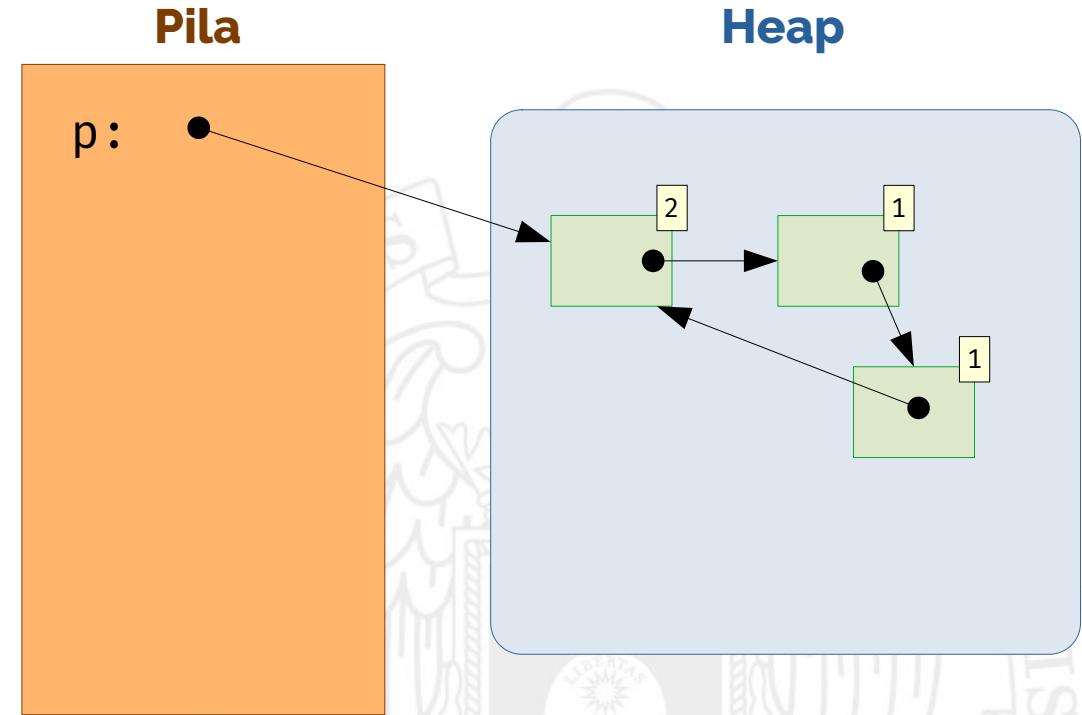


Copia de un shared_ptr

```
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!



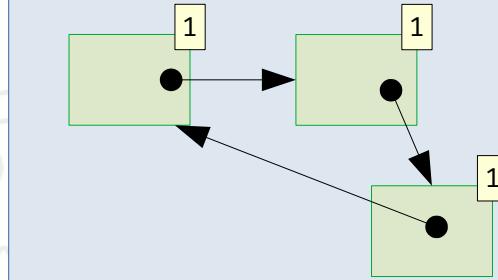
¡Cuidado con las referencias circulares!

```
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

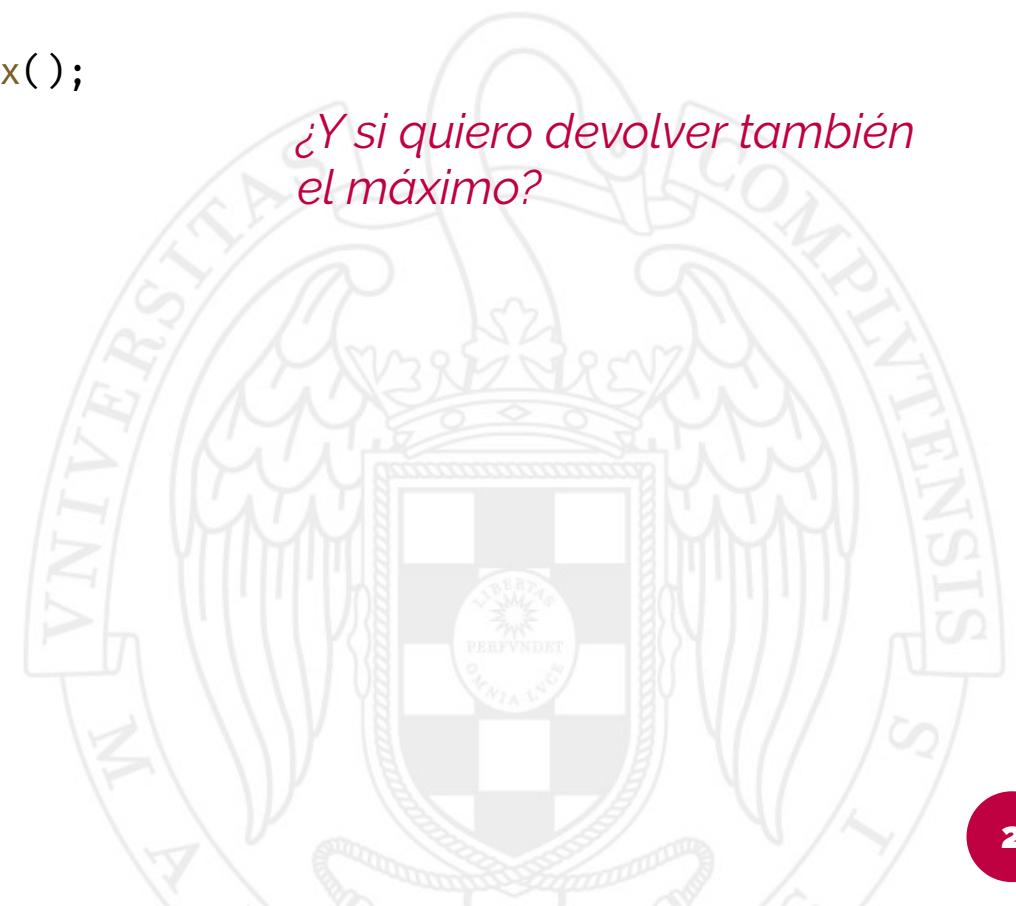
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

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;  
}
```

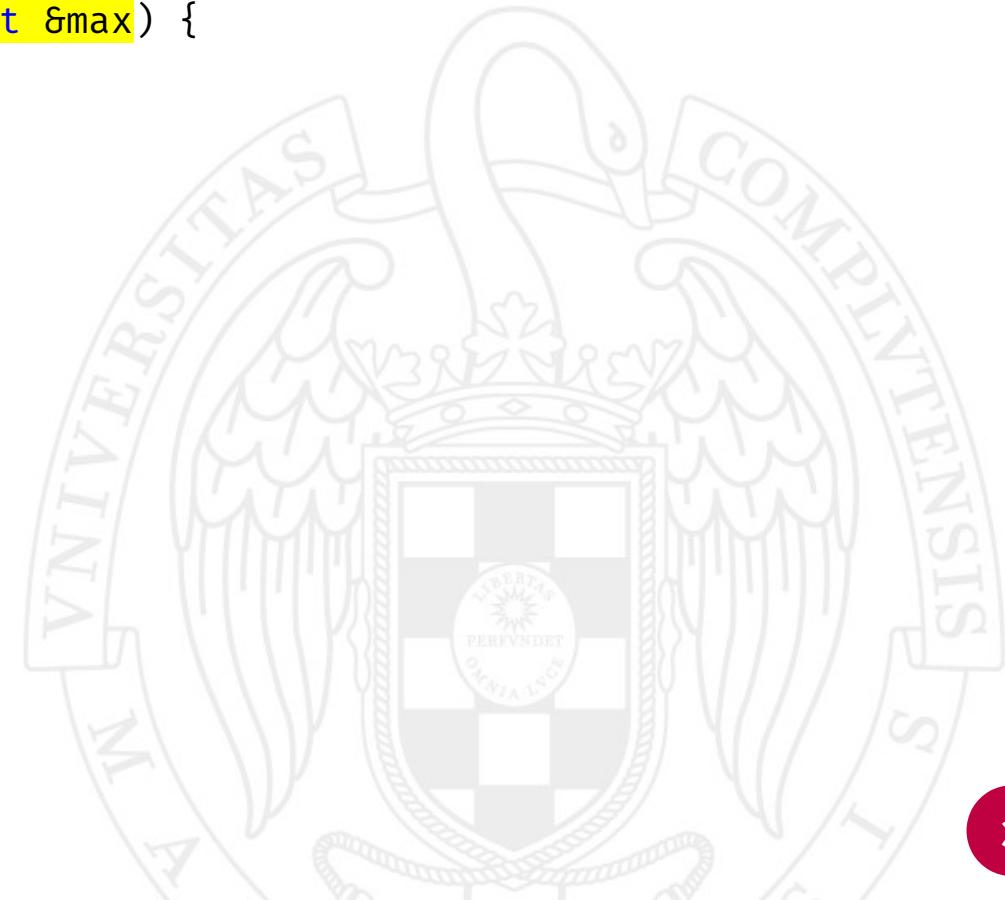
¿Y si quiero devolver también el máximo?



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.

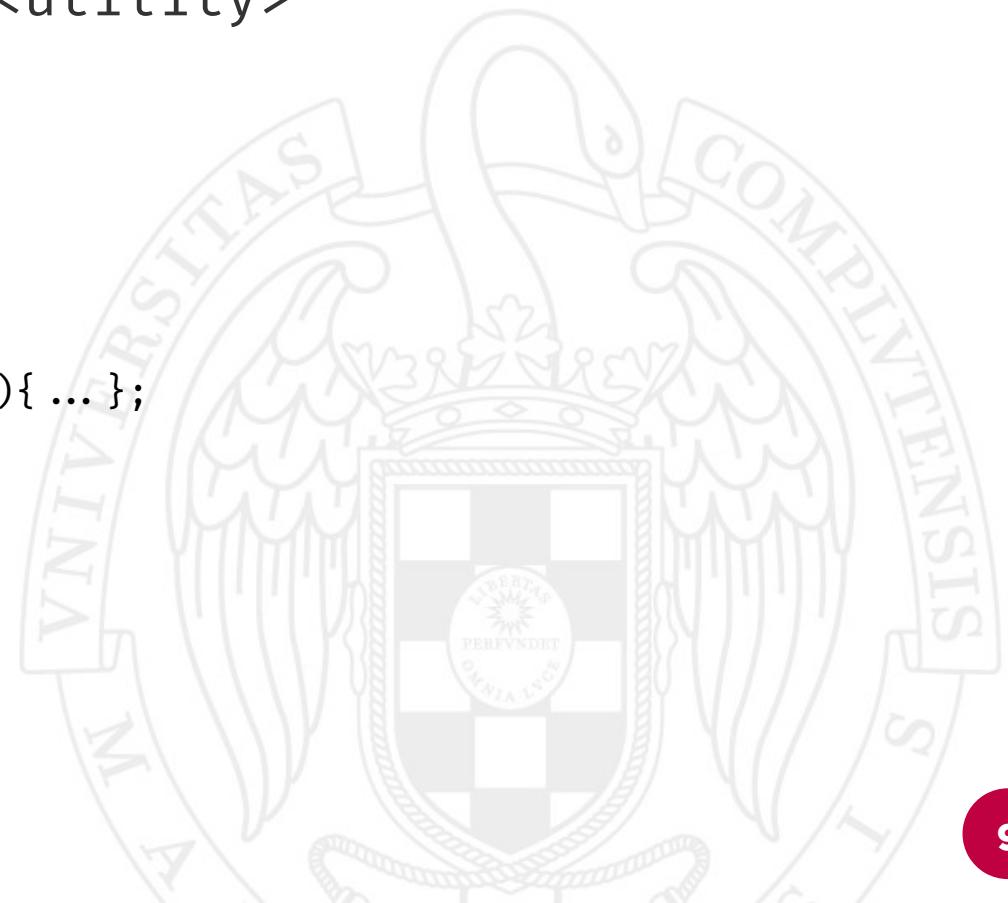
Pares – std :: pair

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

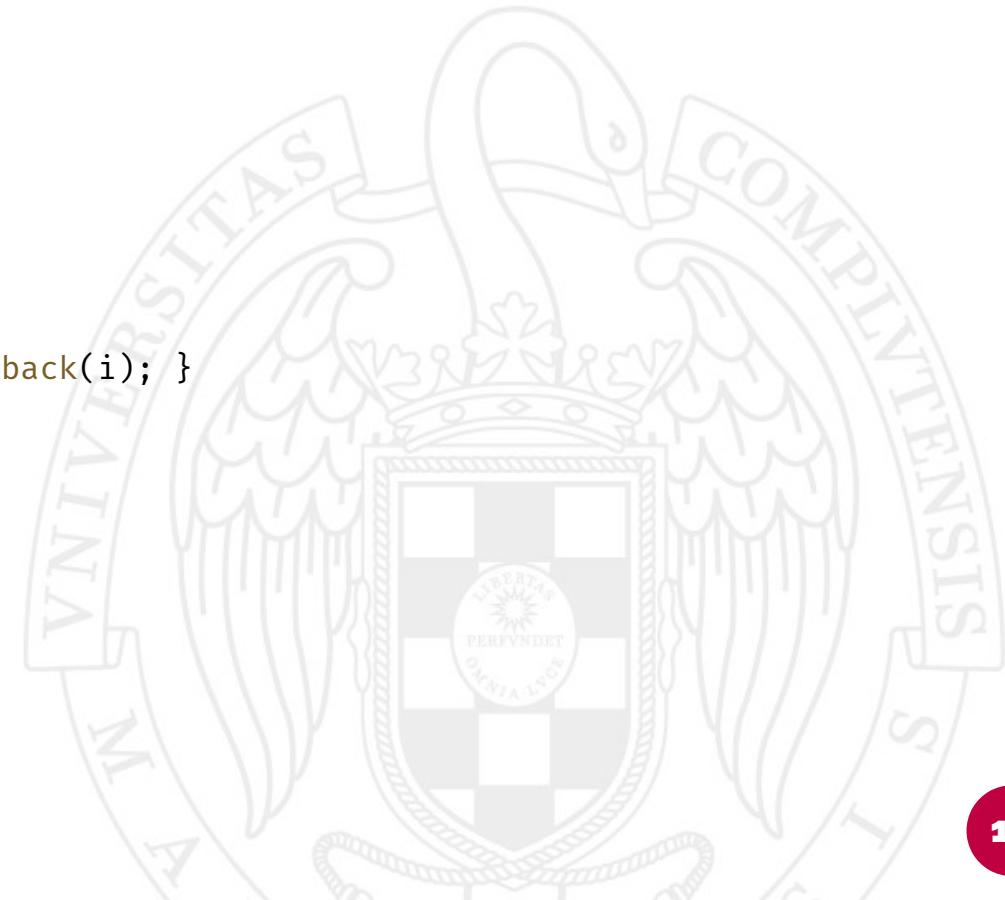
```
bool es_multiplo_de_dos(int x) {
    return x % 2 == 0;
}

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
```

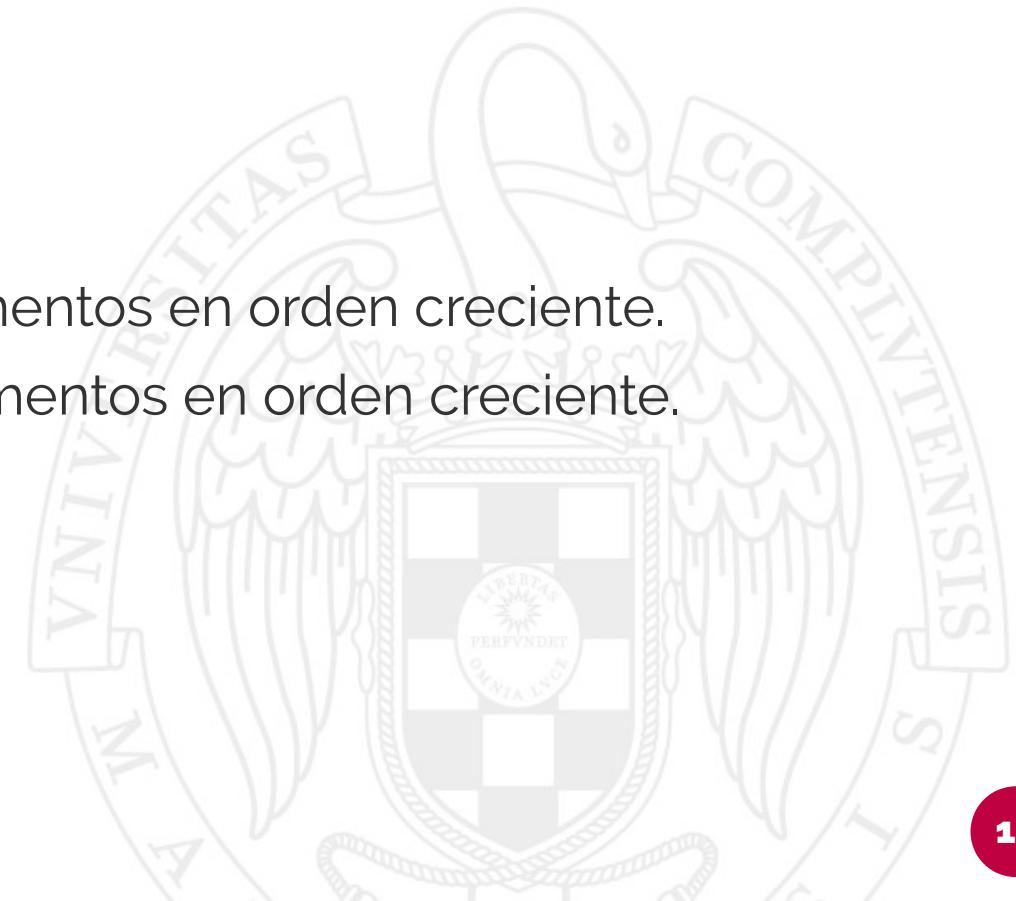
Otras funciones de orden superior

- `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
```

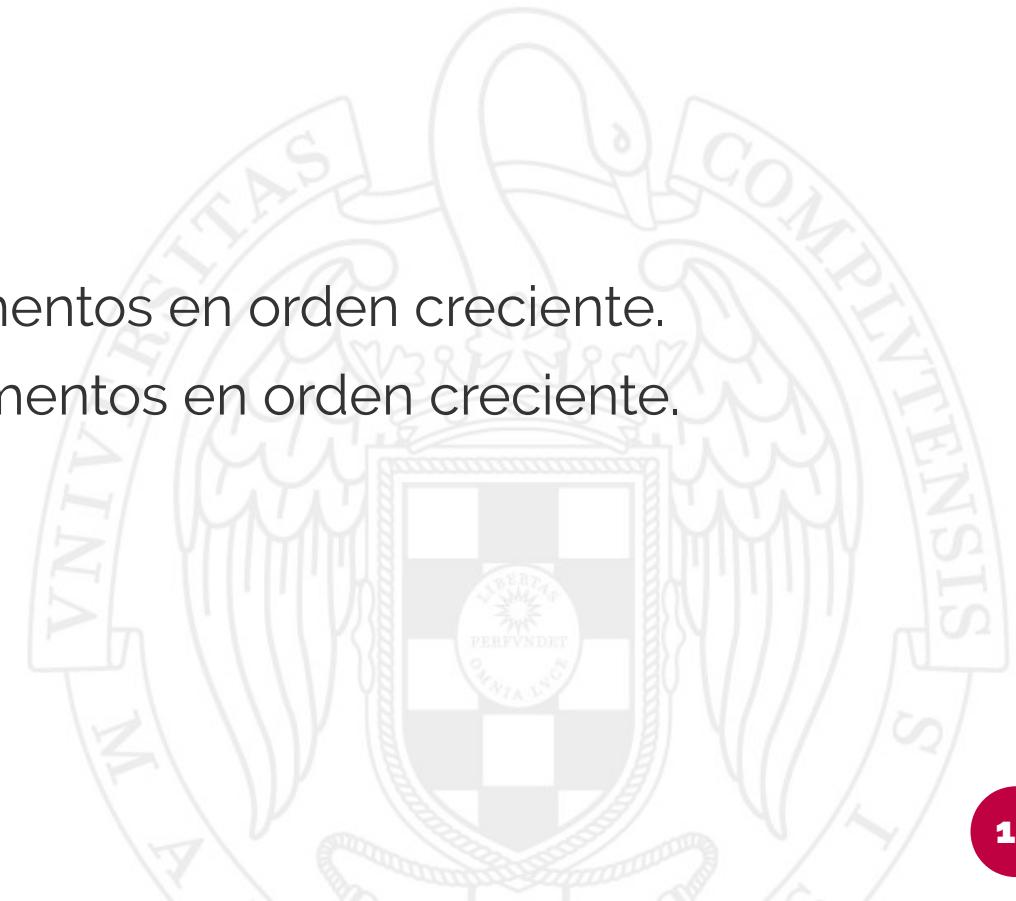
Otras funciones de orden superior

- `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**.

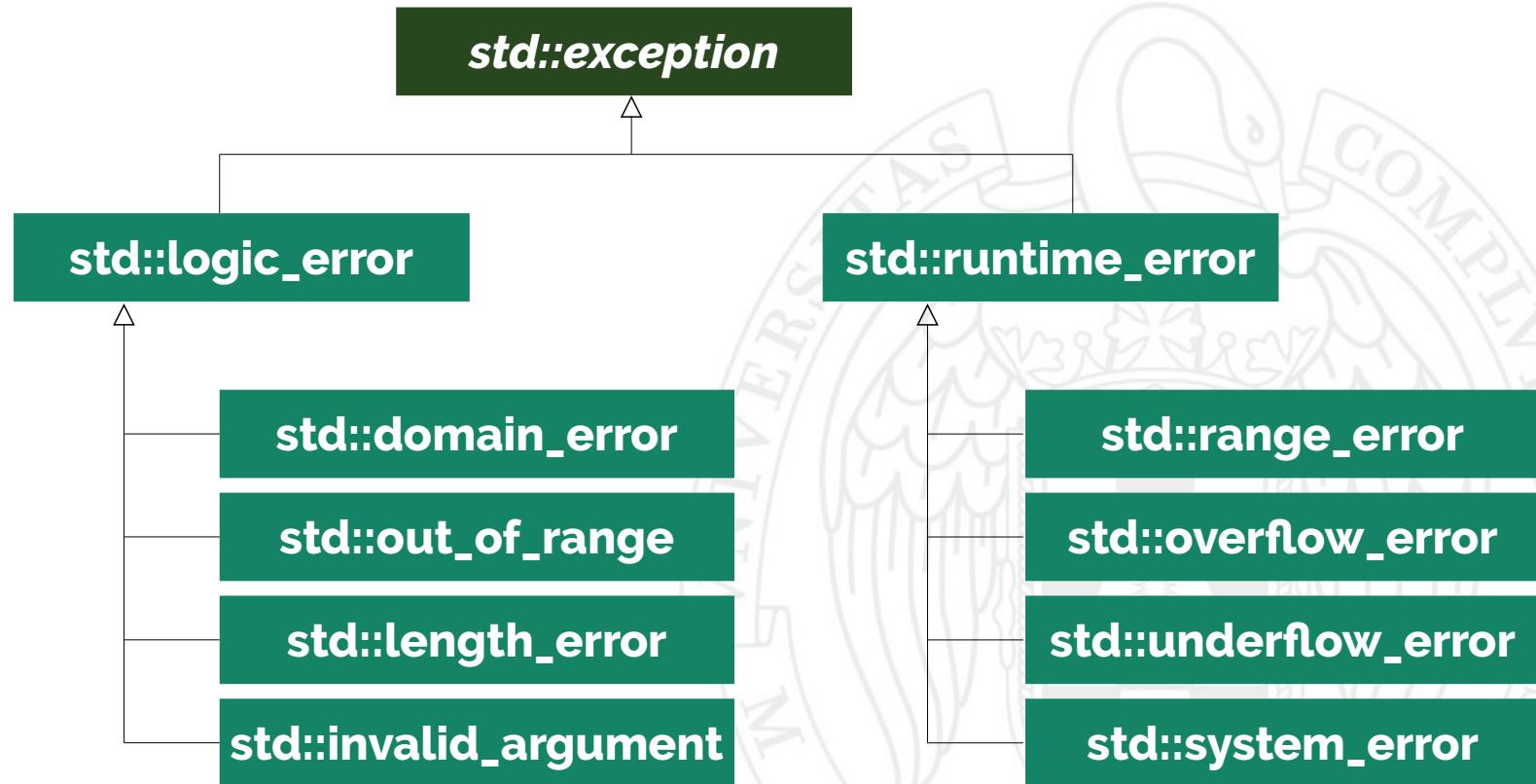
```
try {  
    dividir(1, 0);  
} catch (division_por_cero &e) {  
    std::cout << "División por cero!" << std::endl;  
}
```

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 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

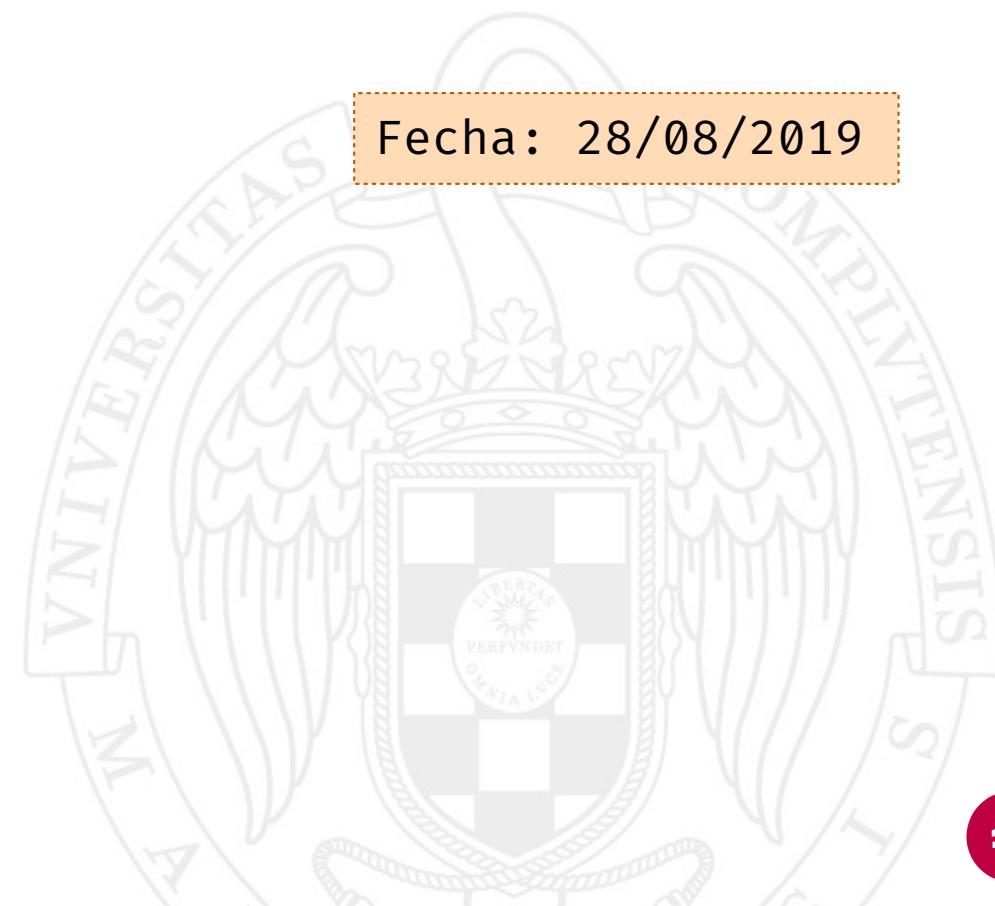
```
void Fecha::imprimir() {  
    std::cout << std::setfill('0') << std::setw(2) << dia << "/"  
        << std::setw(2) << mes << "/"  
        << std::setw(4) << anyo;  
}
```

```
#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

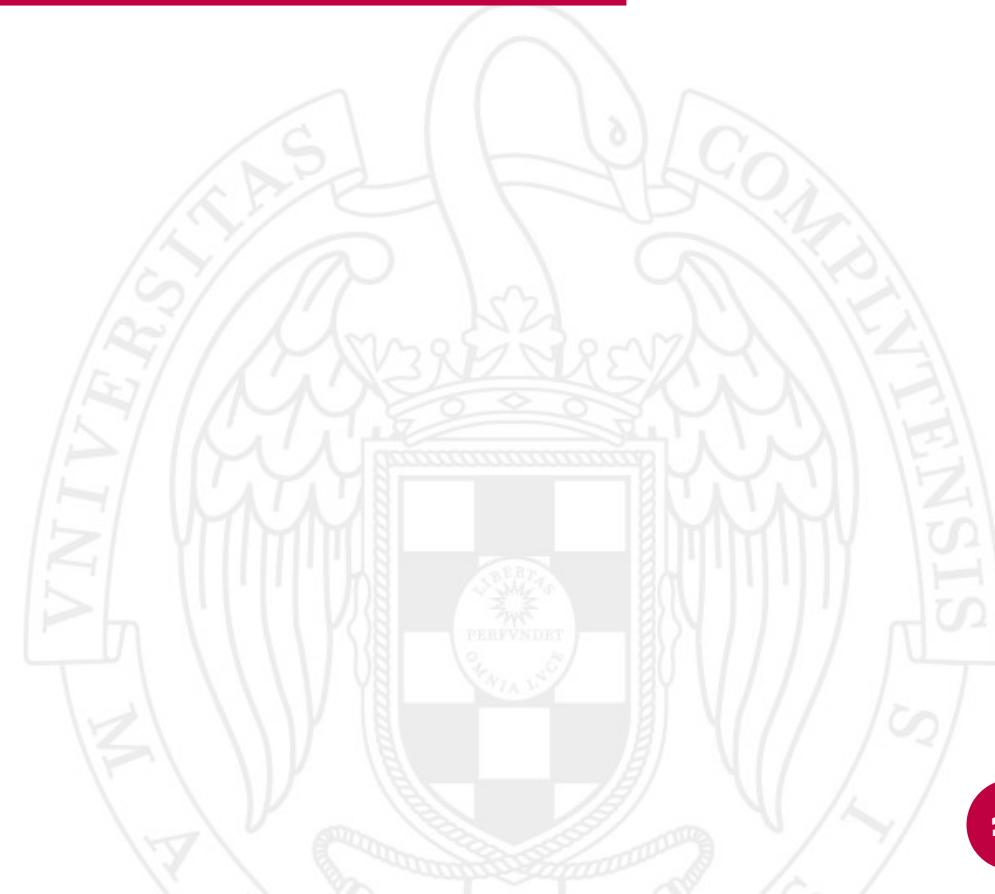
```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo) {  
        this->dia = dia;  
        this->mes = mes;  
        this->anyo = anyo;  
    }  
  
    Fecha(int anyo): Fecha(1, 1, anyo) {  
        // vacío  
    }  
  
    // ...  
private:  
    int dia;  
    int mes;  
    int 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

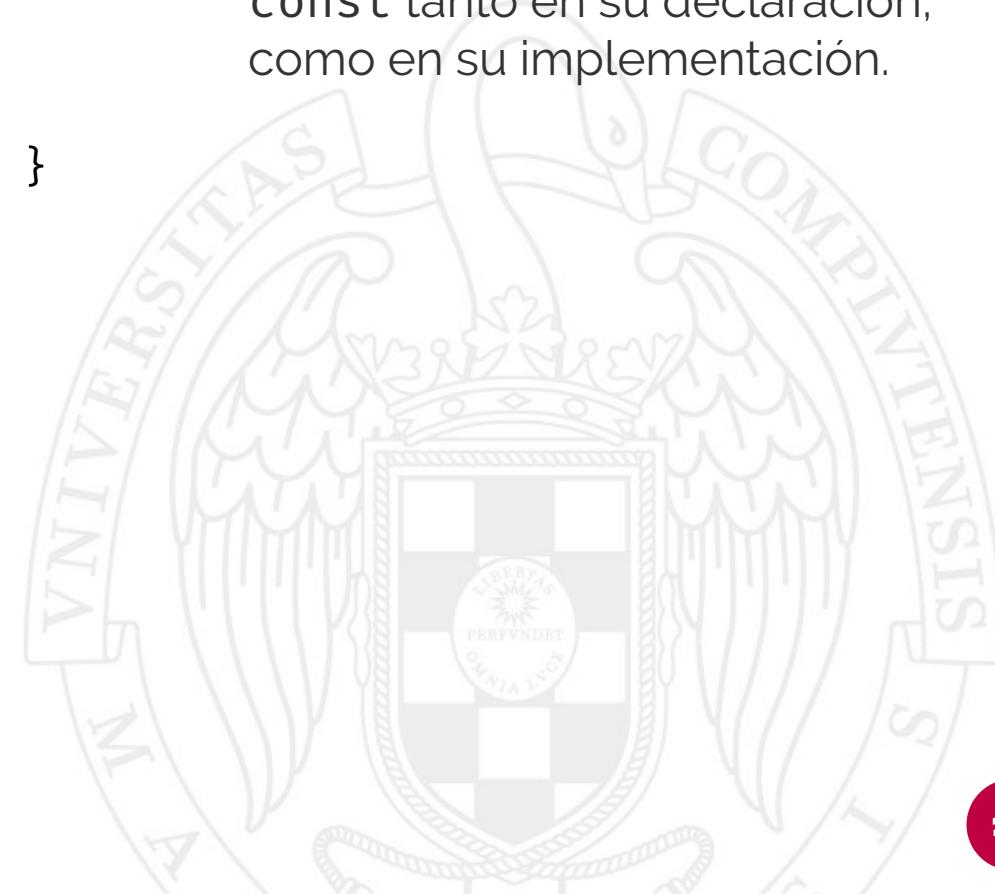
```
class Fecha {  
public:  
    ...  
    int get_dia() const { return dia; }  
    int get_mes() const { return mes; }  
    int get_anyo() const { return anyo; }  
    void imprimir() const;  
    ...  
  
private:  
    ...  
};
```

- 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*



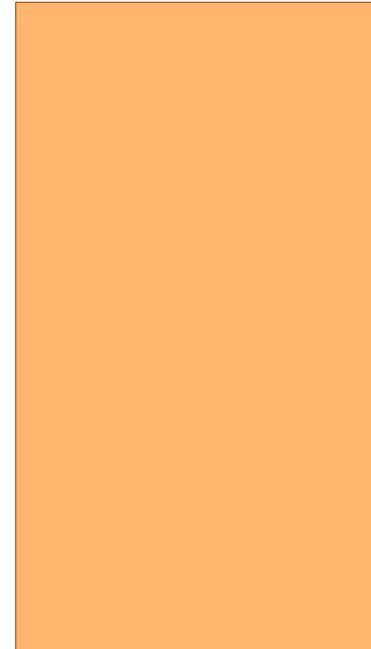
Regiones de memoria

- **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.

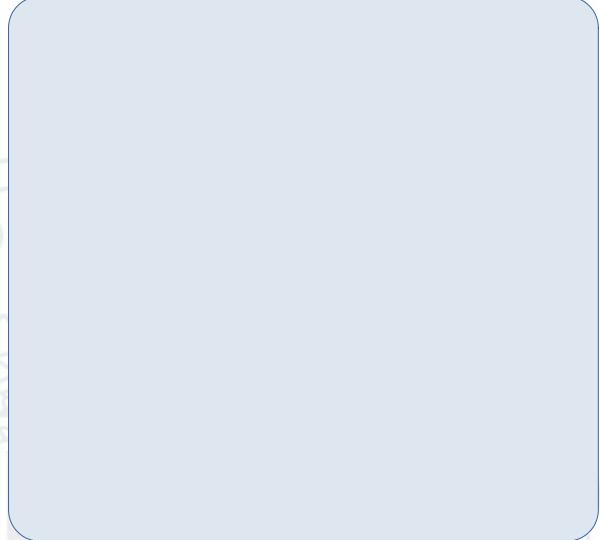
Regiones de memoria

```
int main() {  
    int x = 3;  
    int *y = new int;  
    *y = 3;  
    int *z = &x;  
  
    delete y;  
    return 0;  
}
```

Pila



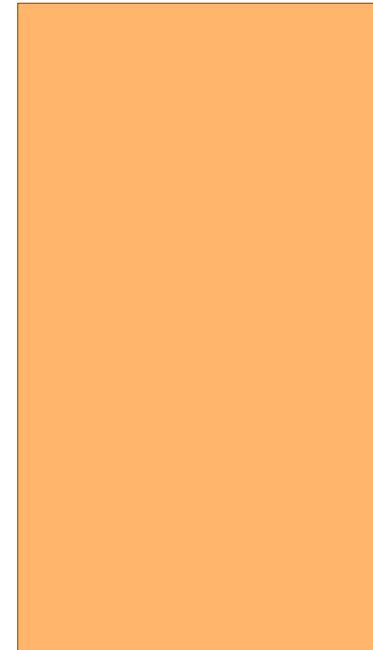
Heap



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

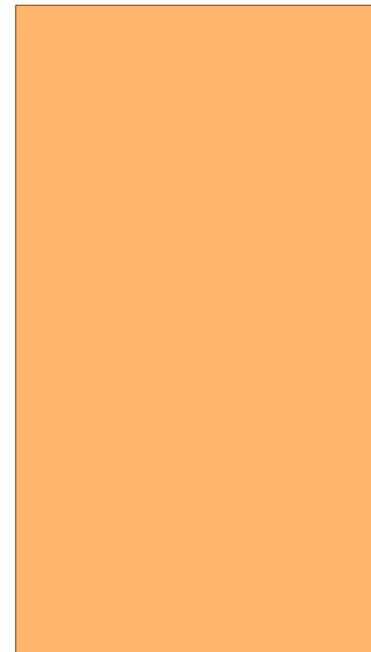
```
int main() {
    Fecha f1(28, 8, 2038);
    Fecha *f2 = new Fecha(10, 6, 2010);

    std::cout << "Fecha 1: ";
    f1.imprimir();
    std::cout << std::endl;

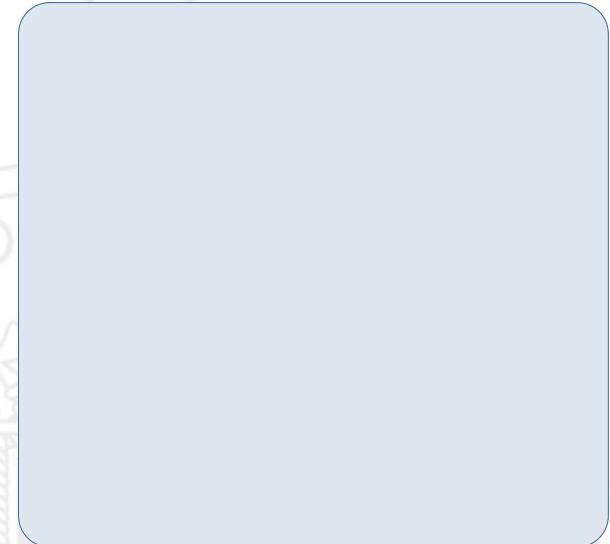
    std::cout << "Fecha 2: ";
    f2->imprimir();
    std::cout << std::endl;

    delete f2;
    return 0;
}
```

Pila



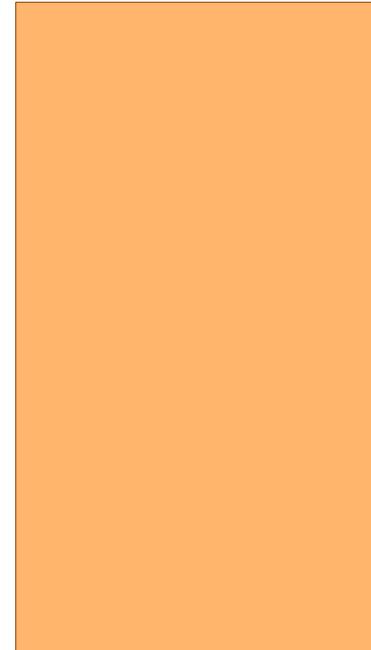
Heap



Arrays de objetos

```
int main() {  
    Fecha fs[3] =  
        { {2010}, {2011}, {2012} };  
  
    return 0;  
}
```

Pila



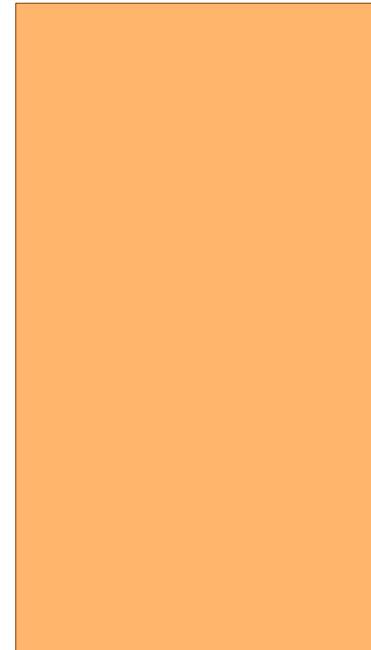
Heap



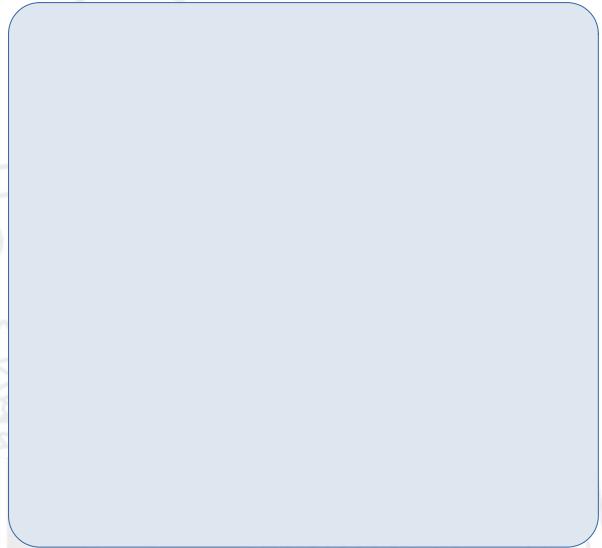
Arrays de punteros a objetos

```
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



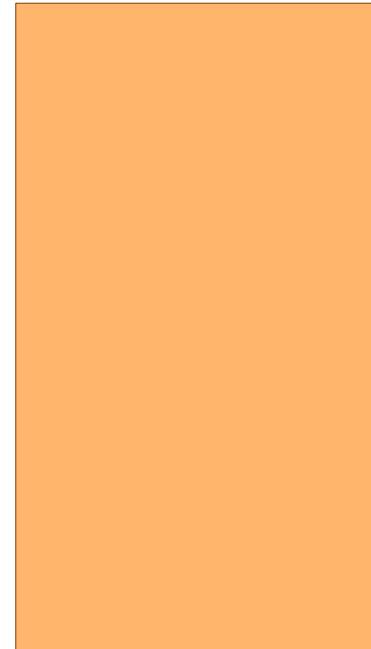
Heap



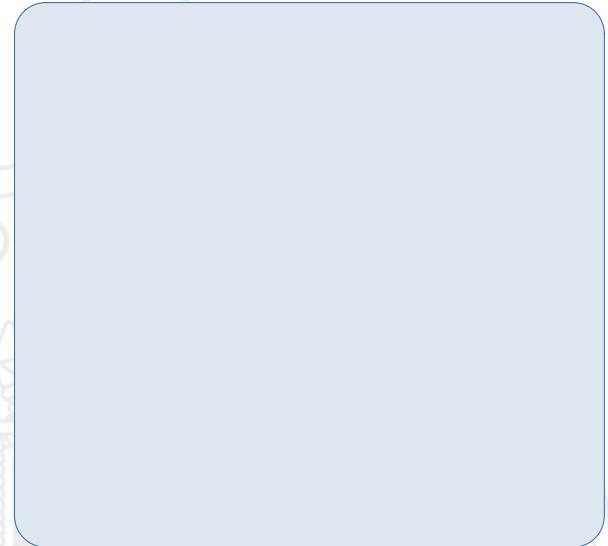
Arrays dinámicos de punteros a objetos

```
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



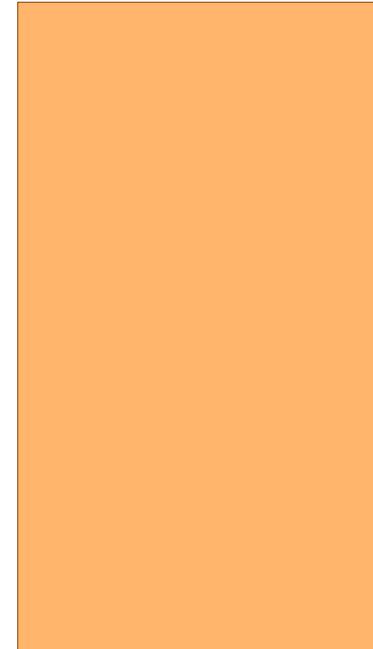
Heap



Arrays dinámicos de objetos

```
int main() {  
    Fecha *fs = new Fecha[3];  
  
    delete[] fs;  
    return 0;  
}
```

Pila



Heap



Añadiendo un constructor por defecto

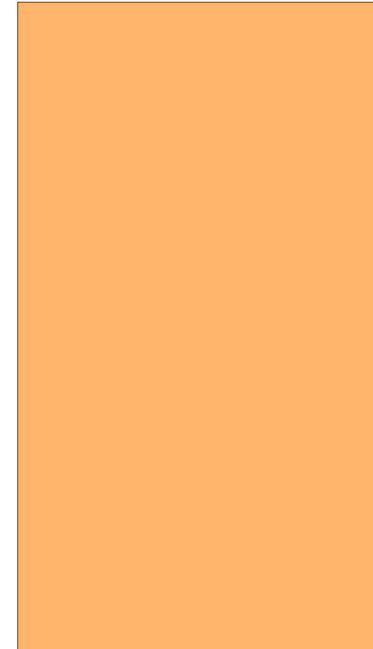
```
class Fecha {  
public:  
    Fecha(int dia, int mes, int anyo);  
    Fecha(int anyo);  
    Fecha(): Fecha(1, 1, 1900) { }  
  
    int get_dia() const;  
    void set_dia(int dia);  
    int get_mes() const;  
    void set_mes(int mes);  
    int get_anyo() const;  
    void set_anyo(int anyo);  
    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

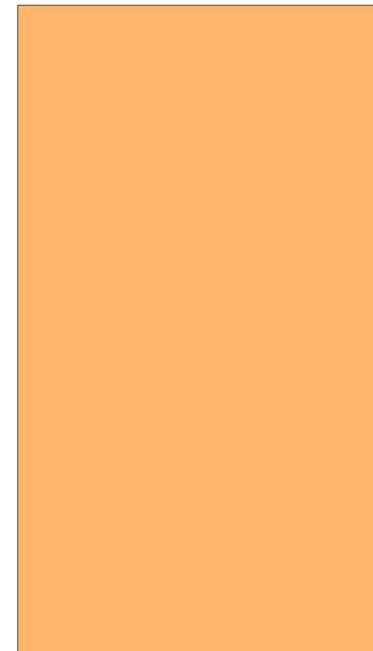


Compartición de objetos

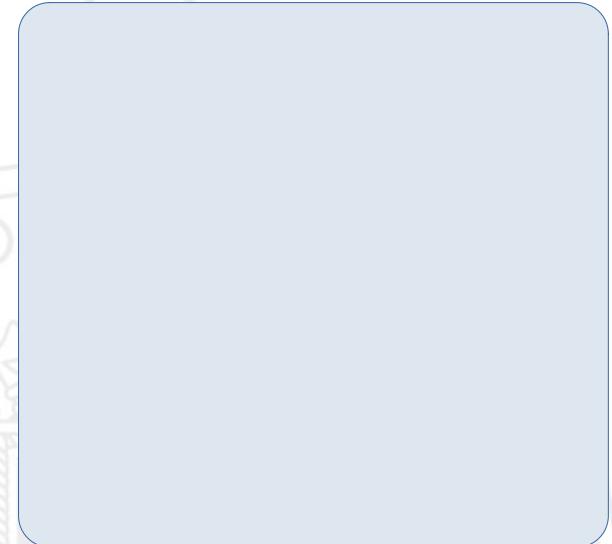
Compartición de punteros

```
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

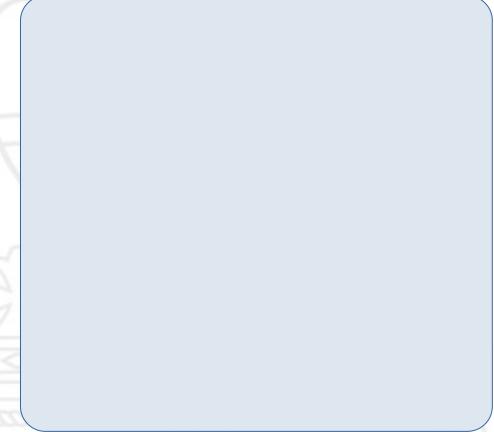


- **Todos los objetos viven en el heap.**
- La pila solo almacena valores básicos o punteros a objetos.

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;  
};
```



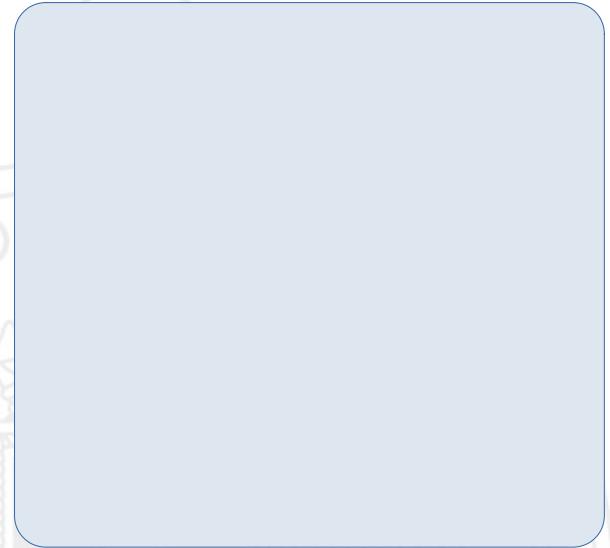
Ejemplo de uso

```
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;  
};
```

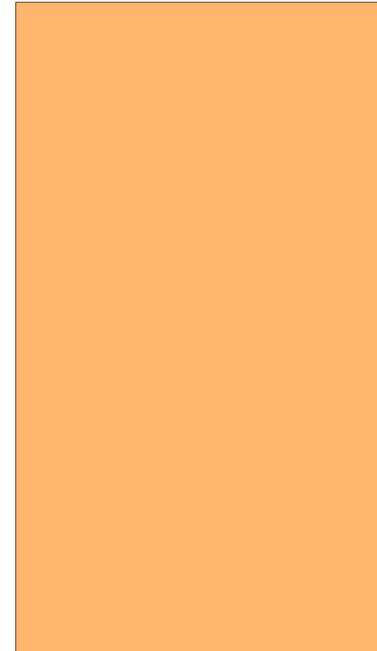


Ejemplo de uso

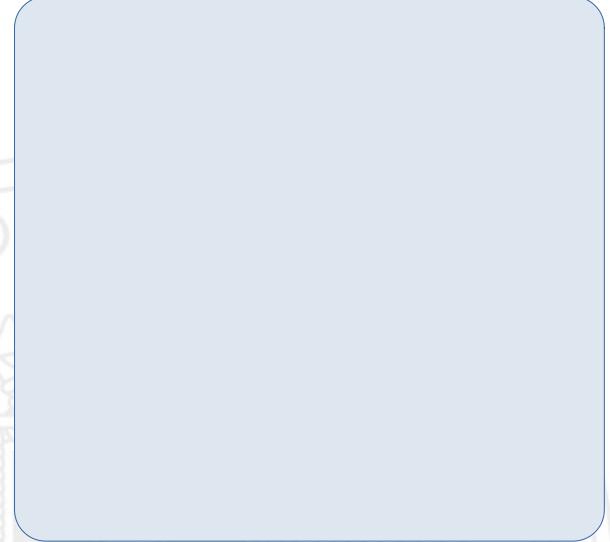
```
void ejemplo() {  
    Persona p("David", 15, 3, 1979);  
}
```

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

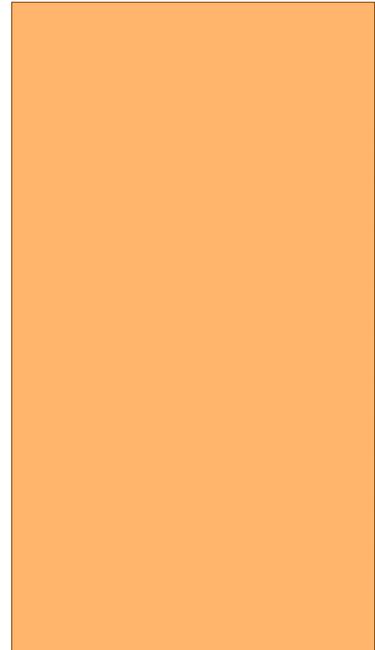
```
class Persona {  
public:  
    Persona(std::string nombre, int dia, int mes, int anyo)  
        : fecha_nacimiento(new Fecha(dia, mes, anyo)) {}  
  
    ~Persona() {  
        delete fecha_nacimiento;  
    }  
  
private:  
    std::string nombre;  
    Fecha *fecha_nacimiento;  
};
```

← Método destructor

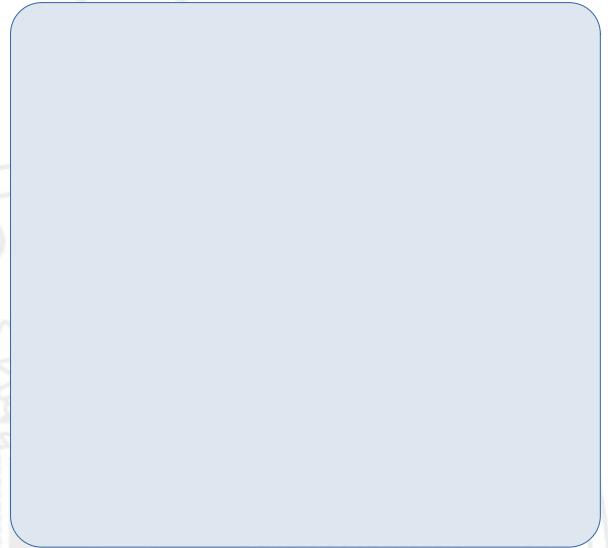
Ejemplo de uso

```
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?

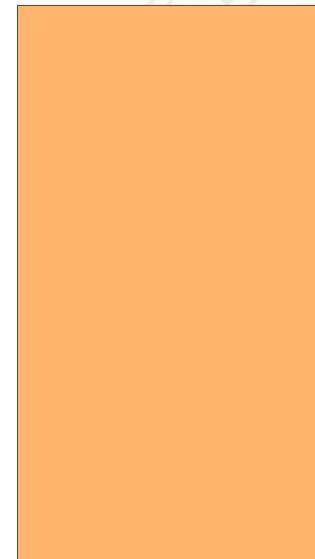
- Cuando se pasa una instancia a una función como parámetro **por valor**, se crea una copia de dicha instancia.
- **¿Cómo se realiza la copia?** Copiando el valor de cada uno de los atributos de la instancia “origen” a la instancia “destino”.

```
void modificar_copia(Persona p) { ... }

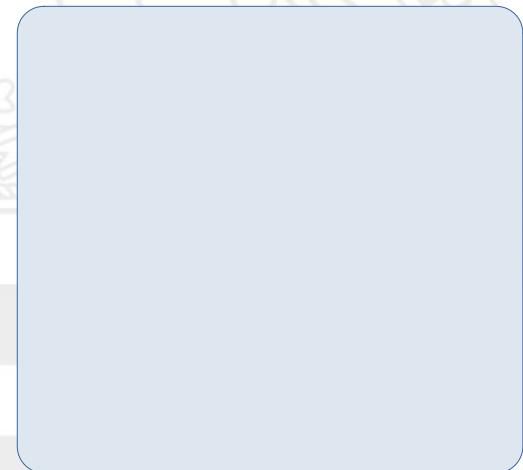
int main() {
    Persona david("David", 15, 3, 1979);
    david.imprimir();
    modificar_copia(david);
    david.imprimir();

    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

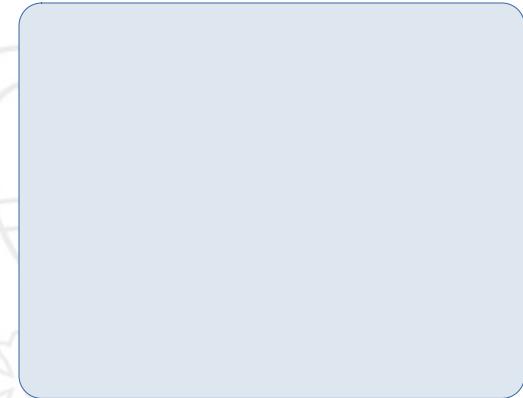
Volviendo a nuestro ejemplo...

```
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

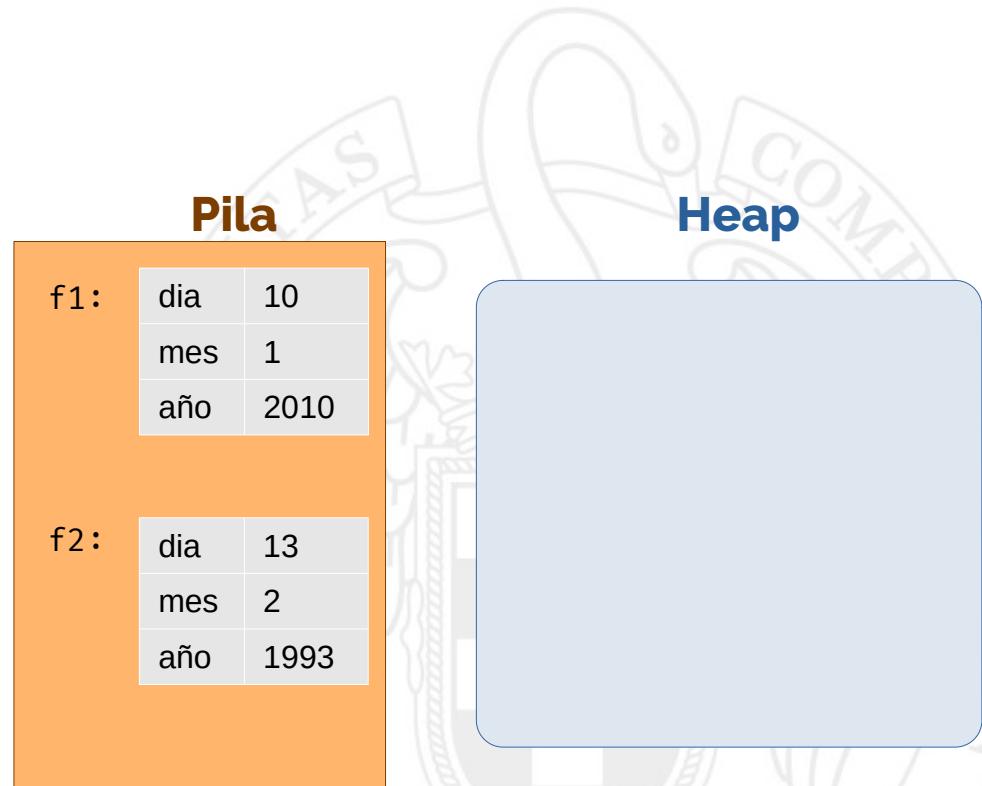
Recordatorio: clase Fecha

```
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

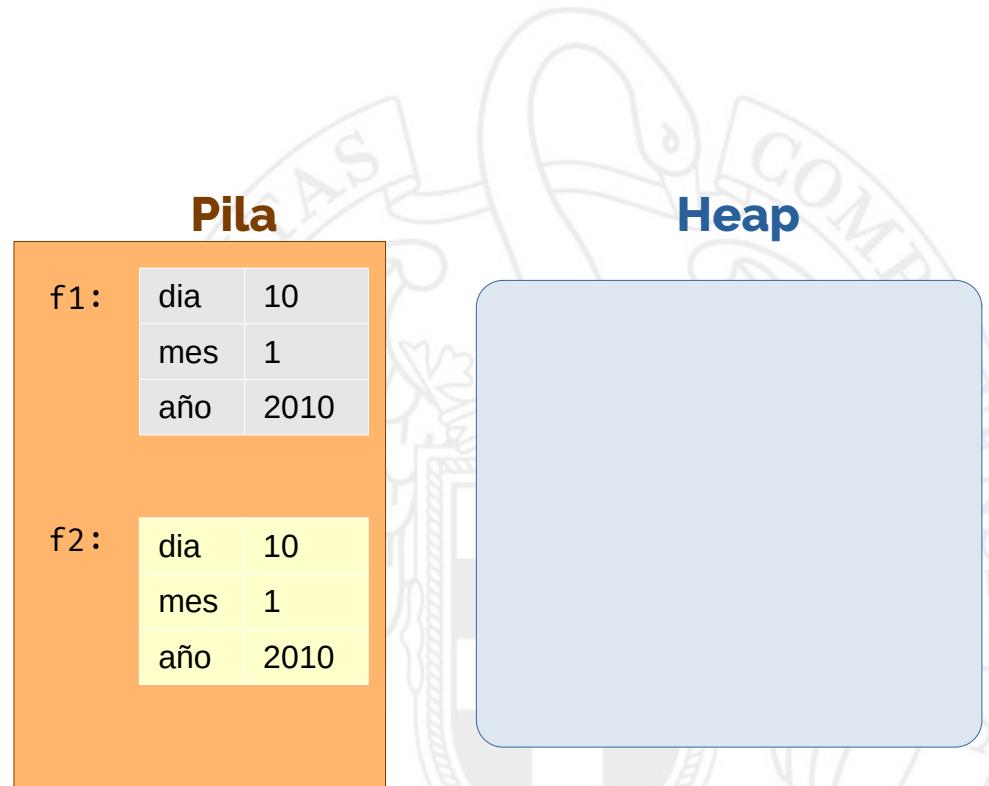
```
Fecha f1(10, 1, 2010);  
Fecha f2(13, 2, 1993);  
  
f2 = f1;
```



Asignar un objeto Fecha a otro

```
Fecha f1(10, 1, 2010);  
Fecha f2(13, 2, 1993);
```

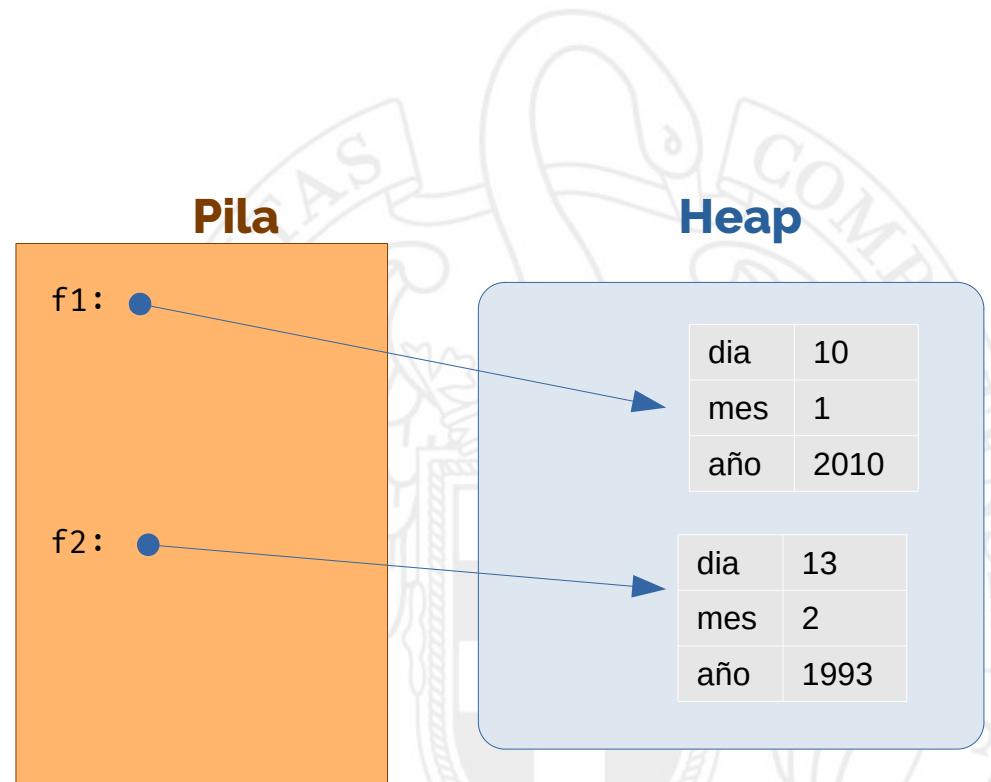
```
f2 = f1;
```



Asignar un objeto Fecha a otro

```
Fecha *f1 = new Fecha(10, 1, 2010);  
Fecha *f2 = new Fecha(13, 2, 1993);
```

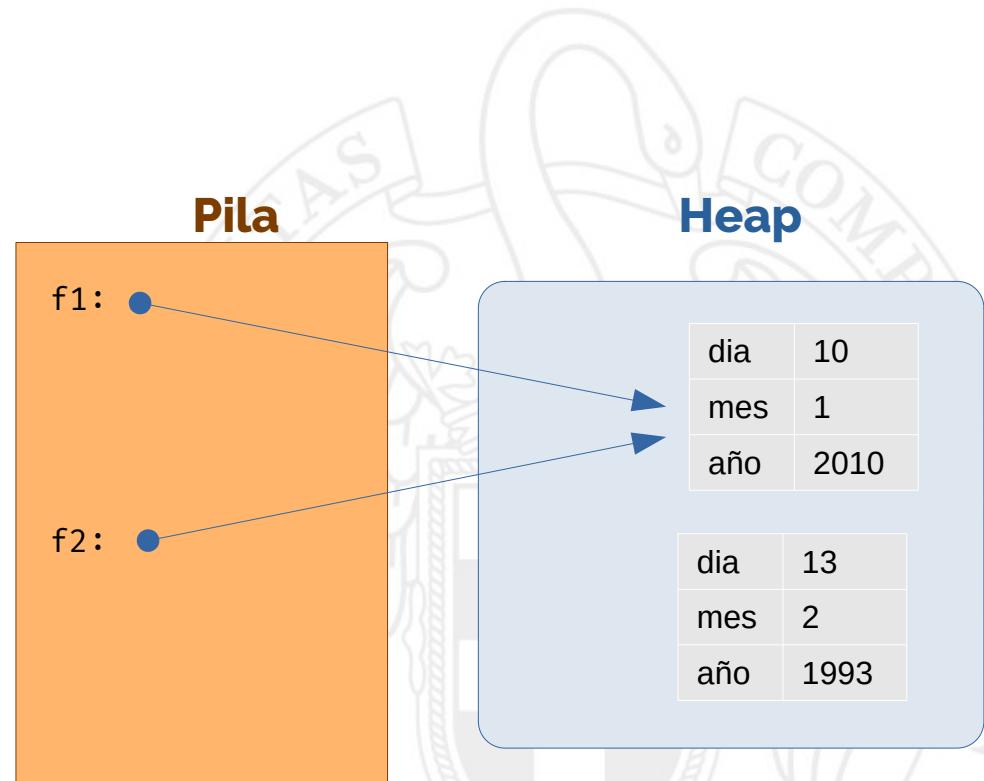
```
f2 = f1;
```



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

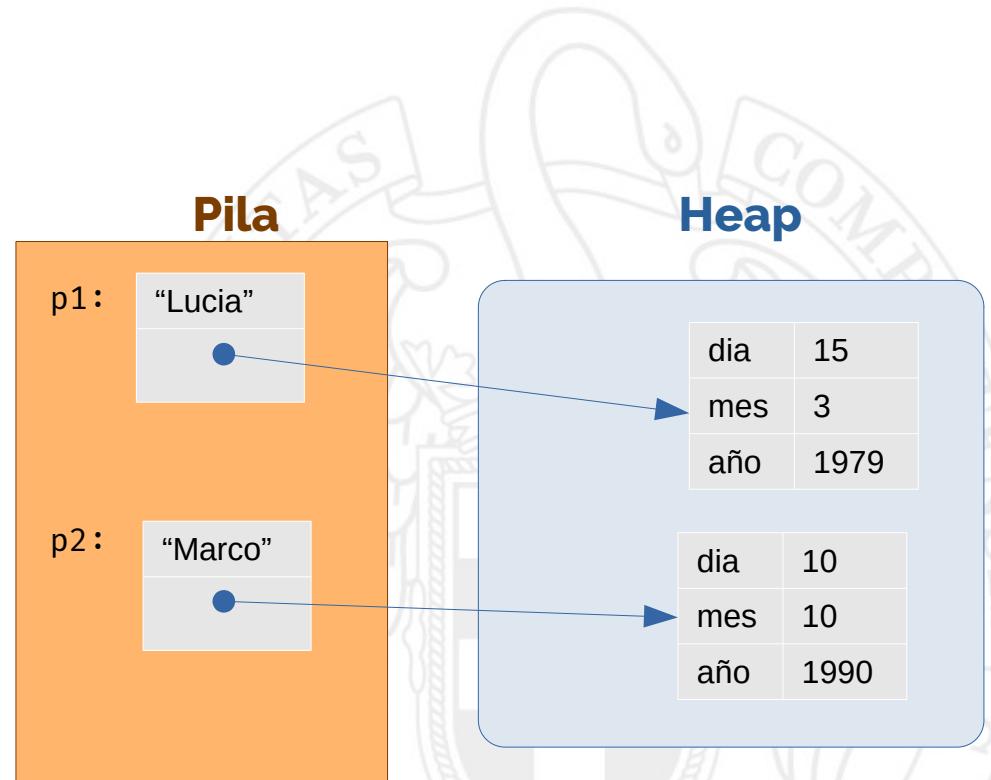
```
class Persona {  
public:  
    Persona(std::string nombre,  
            int dia,  
            int mes,  
            int anyo);  
  
    ~Persona() {  
        delete fecha_nacimiento;  
    }  
  
    ...  
  
private:  
    std::string nombre;  
    Fecha *fecha_nacimiento;  
};
```



Asignar un objeto Persona a otro

```
Persona p1("Lucía", 15, 3, 1979);  
Persona p2("Marco", 10, 10, 1990);
```

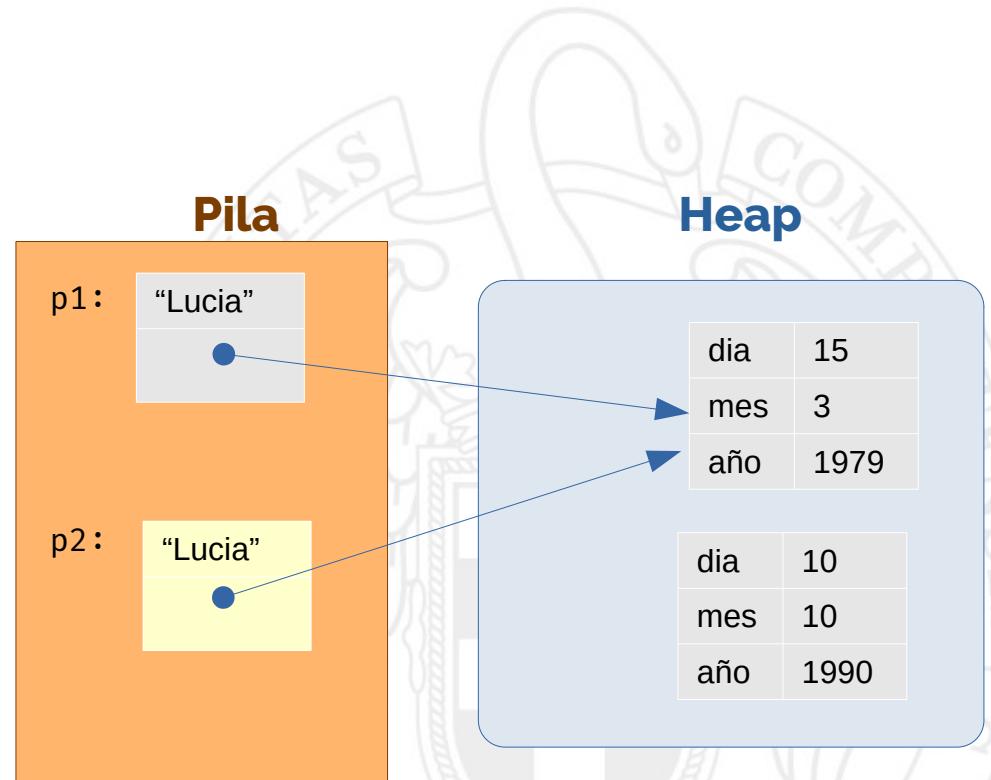
```
p2 = p1;
```



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

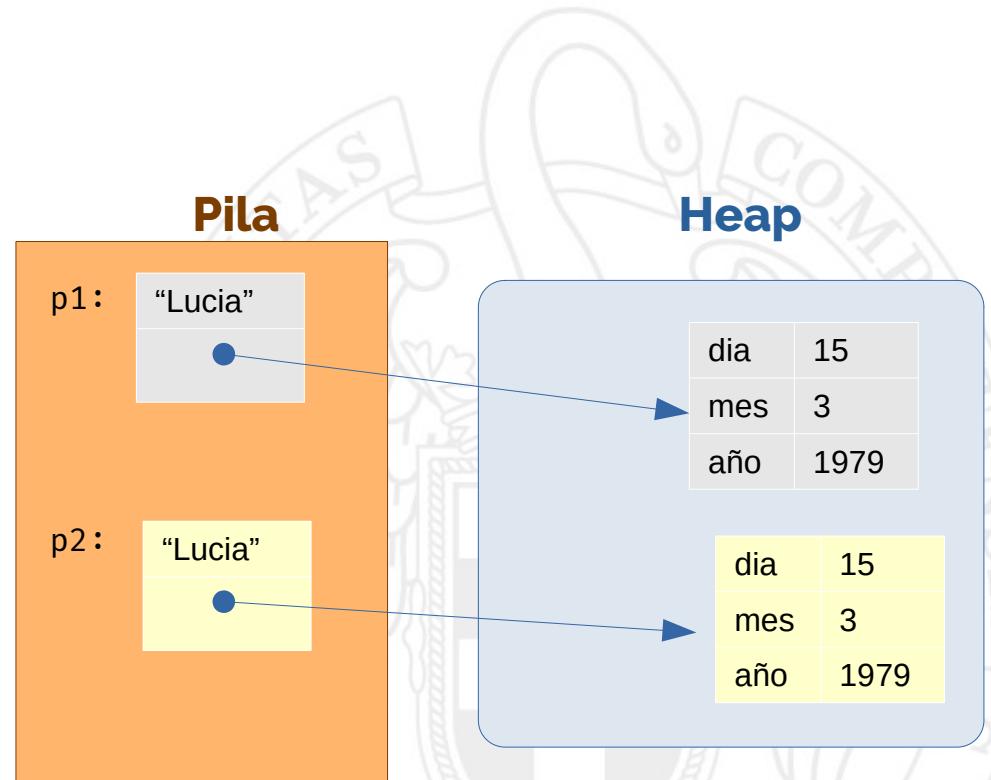
equivale a

p2.operator=(p1)

Asignar un objeto Persona a otro

```
Persona p1("Lucía", 15, 3, 1979);  
Persona p2("Marco", 10, 10, 1990);
```

```
p2 = p1;
```



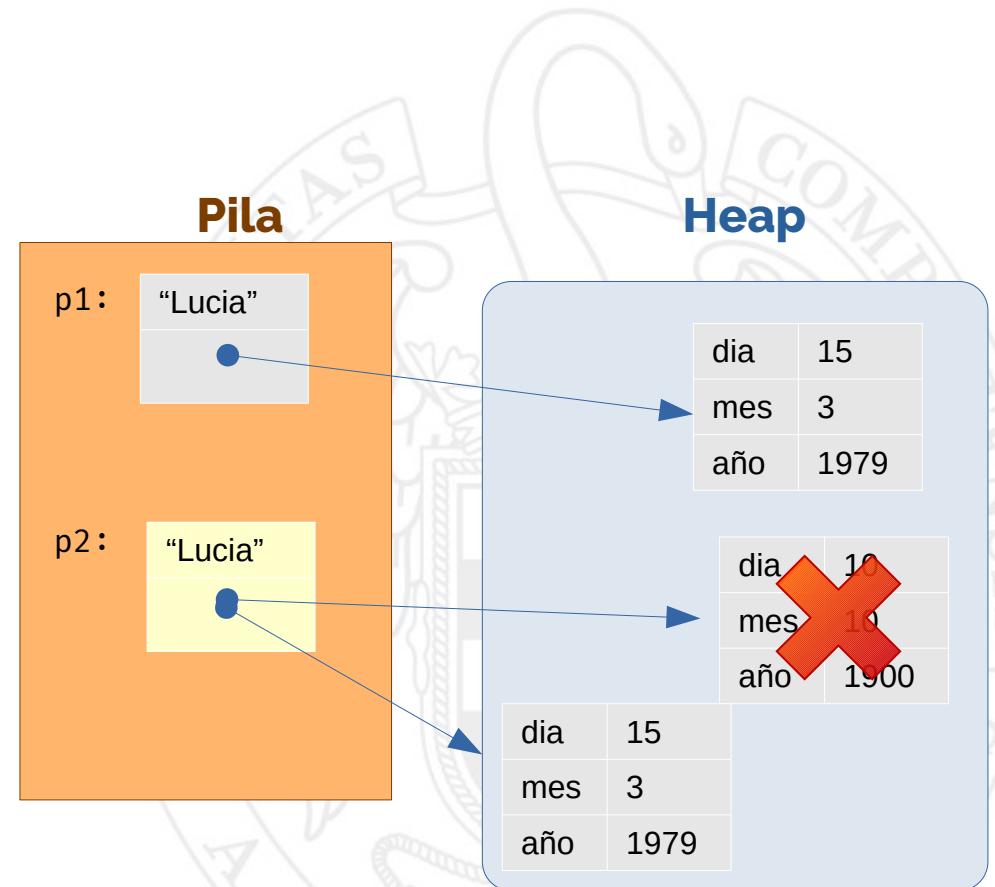
Otra posibilidad

```
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;
```

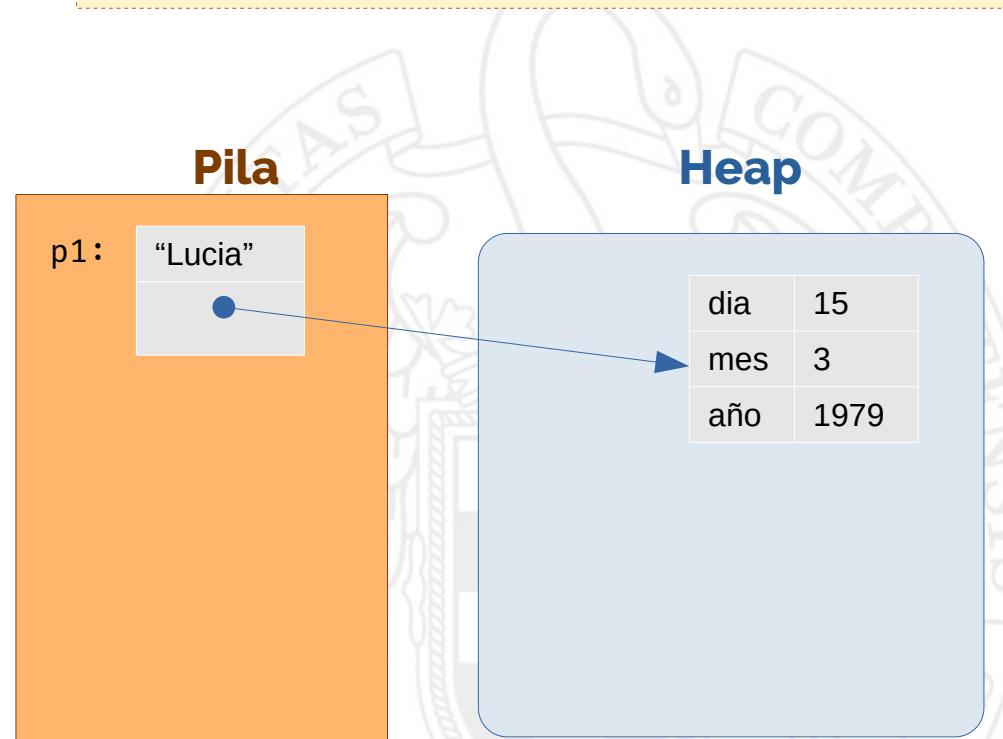


El problema de la autoasignación

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;
```

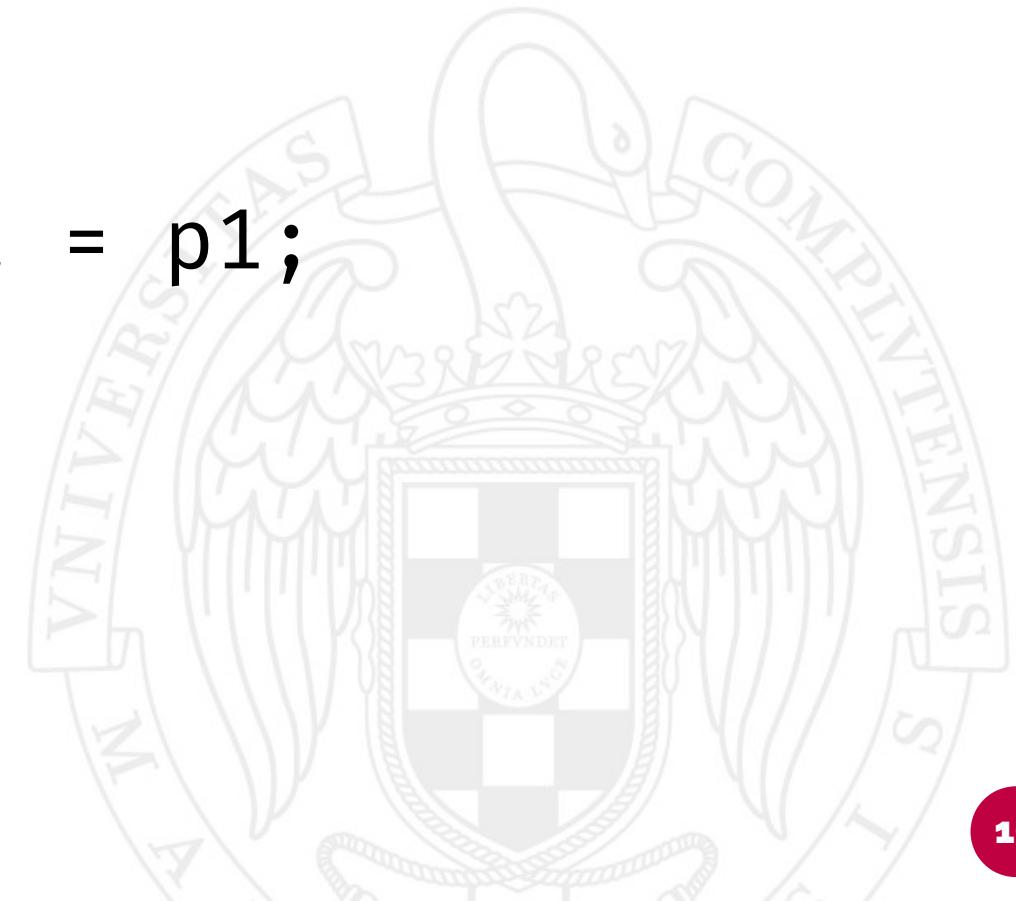
x = y = z = 0;

¿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;  
};
```

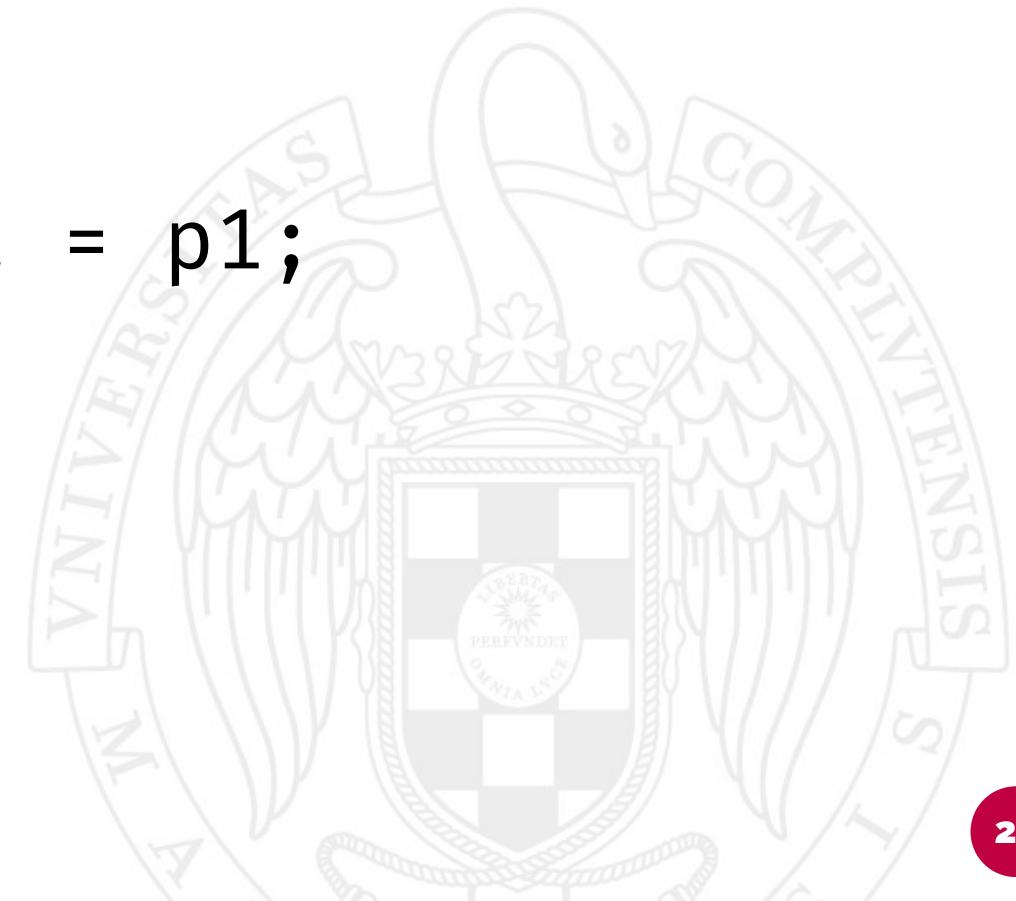


¿Podemos hacer lo mismo con objetos Persona?

```
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*

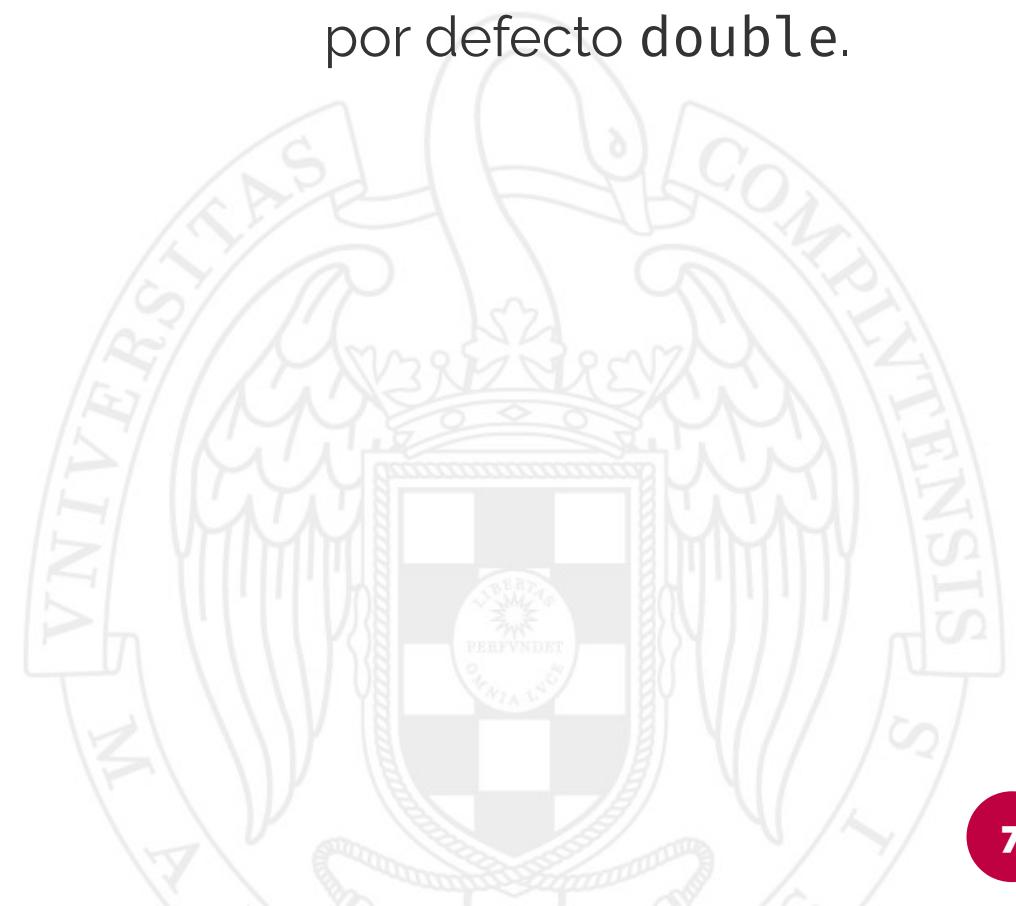
Complejo z4(2.0, -3.0); 

- ... 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

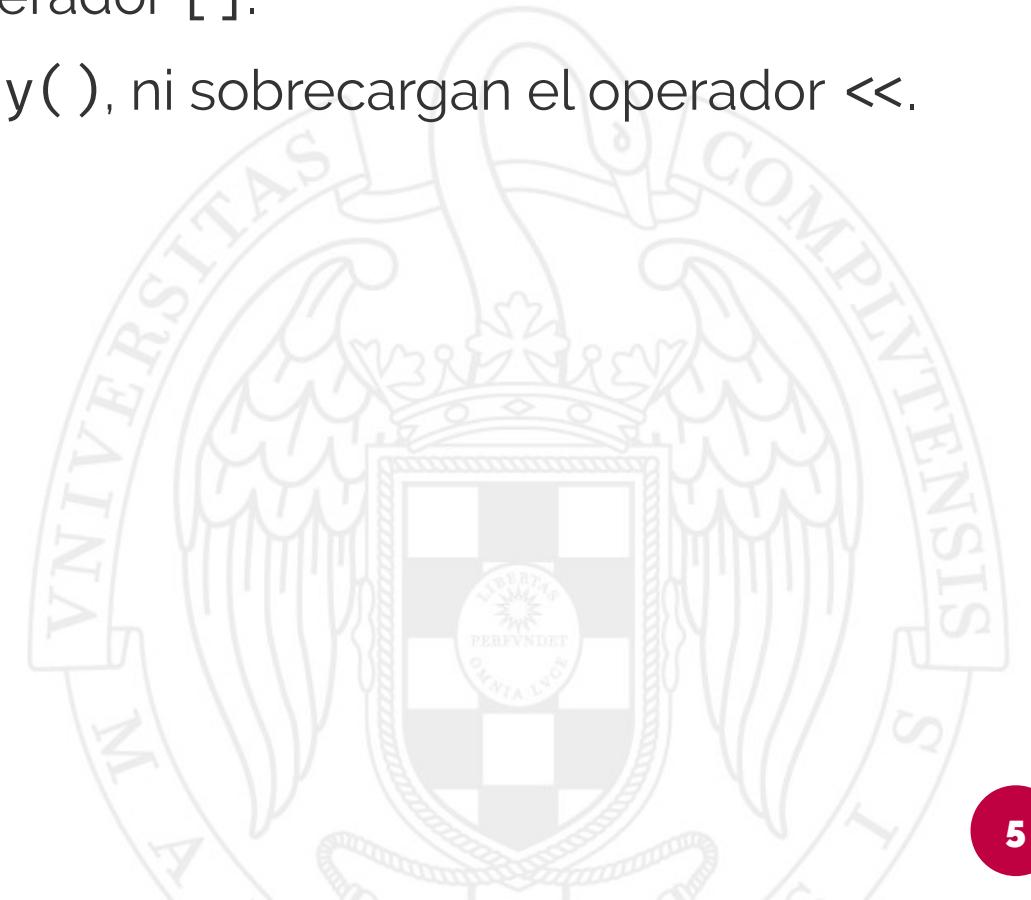
Operaciones

- 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:

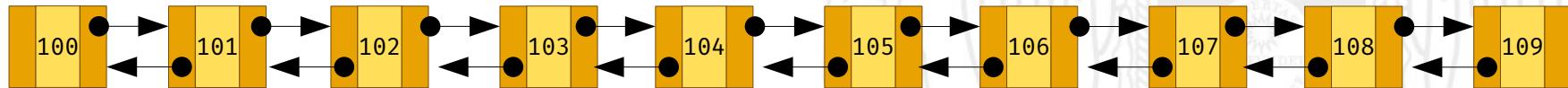
- `source_begin`, `source_end` son iteradores de entrada.
 - `destination_begin` es iterador de salida.
- Copia el intervalo de elementos delimitado por `source_begin` y `source_end` (excluyendo este último), a la posición apuntada por el iterador `destination_begin`.

Ejemplo

```
int main() {
    vector<int> origen;
    list<int> destino;

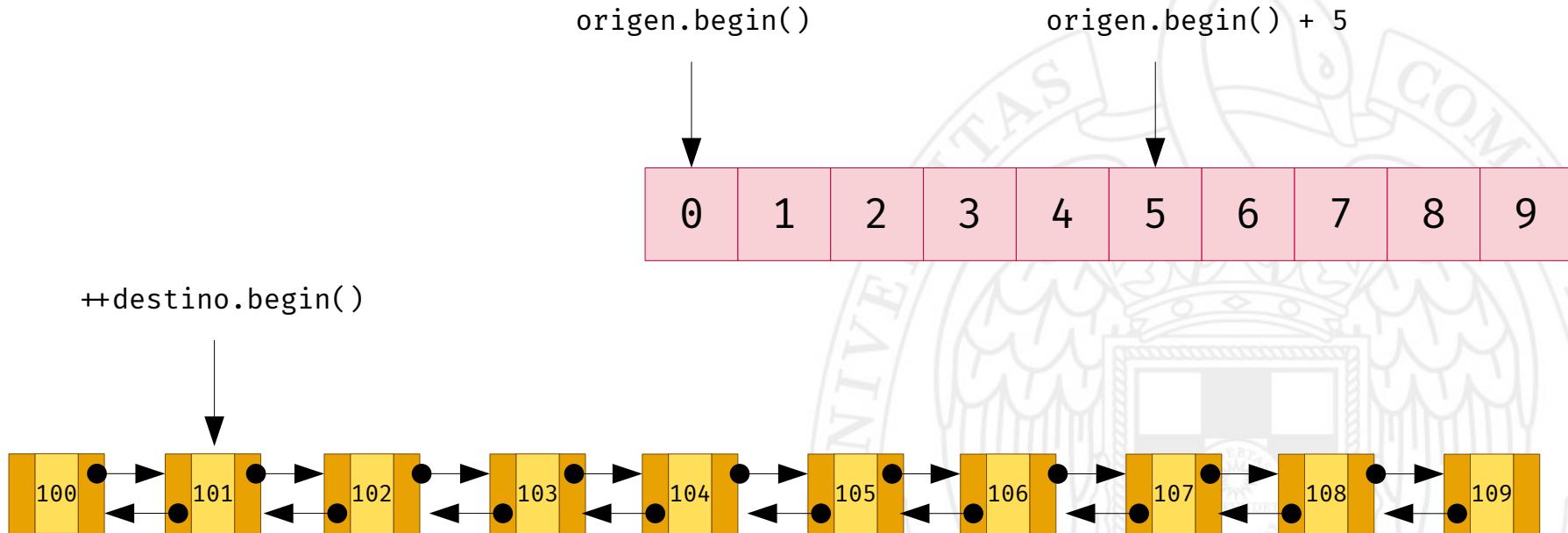
    for (int i = 0; i < 10; i++) {
        origen.push_back(i);
        destino.push_back(100 + i);
    }
    ...
}
```

0	1	2	3	4	5	6	7	8	9
---	---	---	---	---	---	---	---	---	---



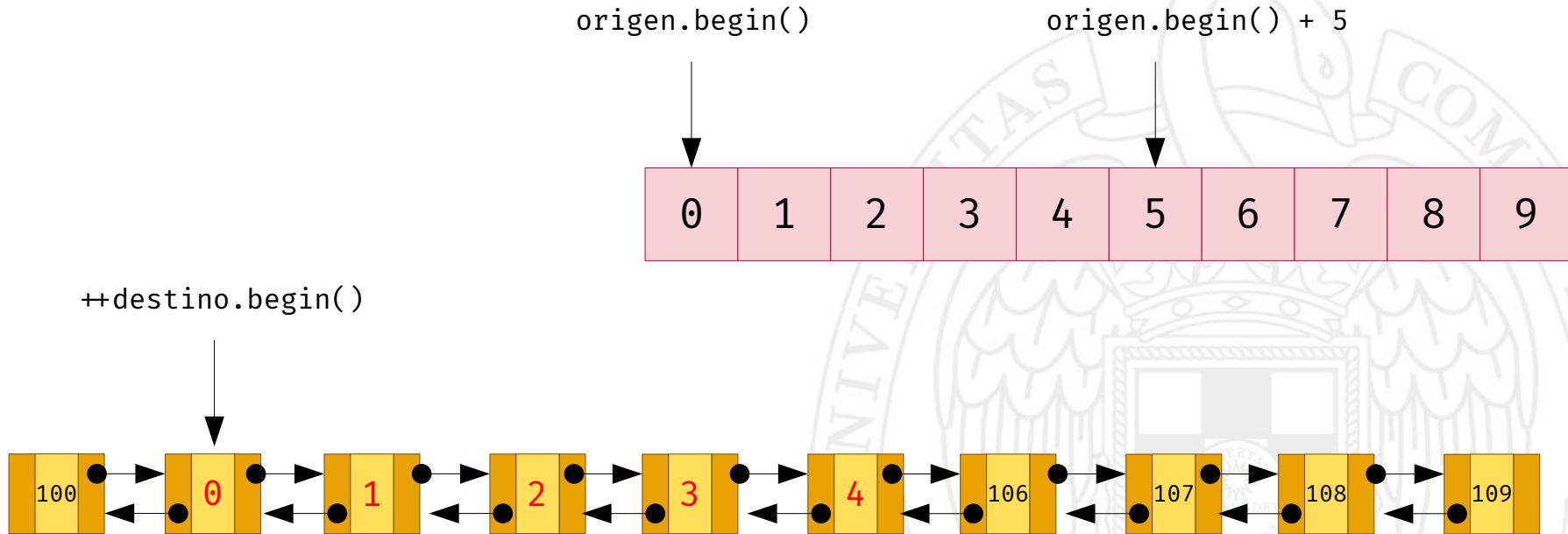
Ejemplo

```
copy(origen.begin(), origen.begin() + 5, ++destino.begin());
```



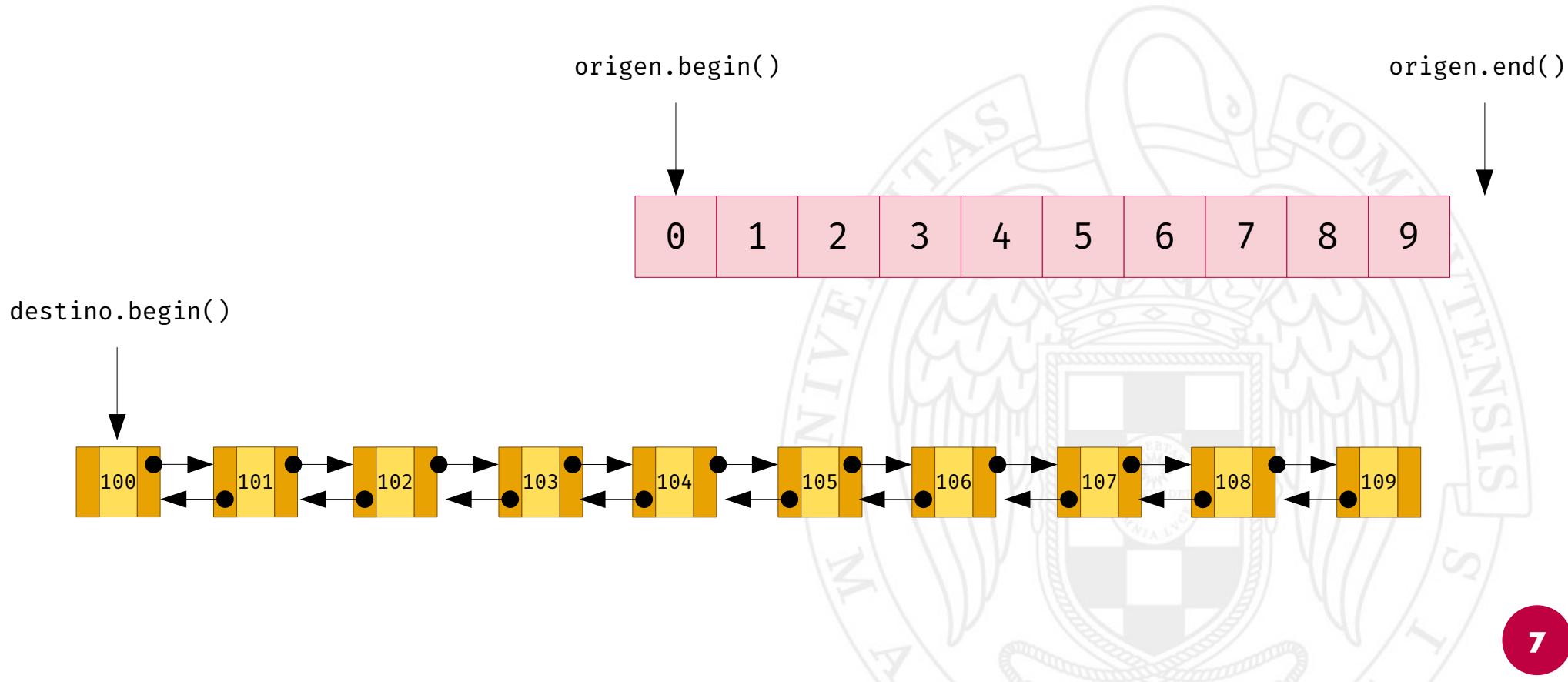
Ejemplo

```
copy(origen.begin(), origen.begin() + 5, ++destino.begin());
```



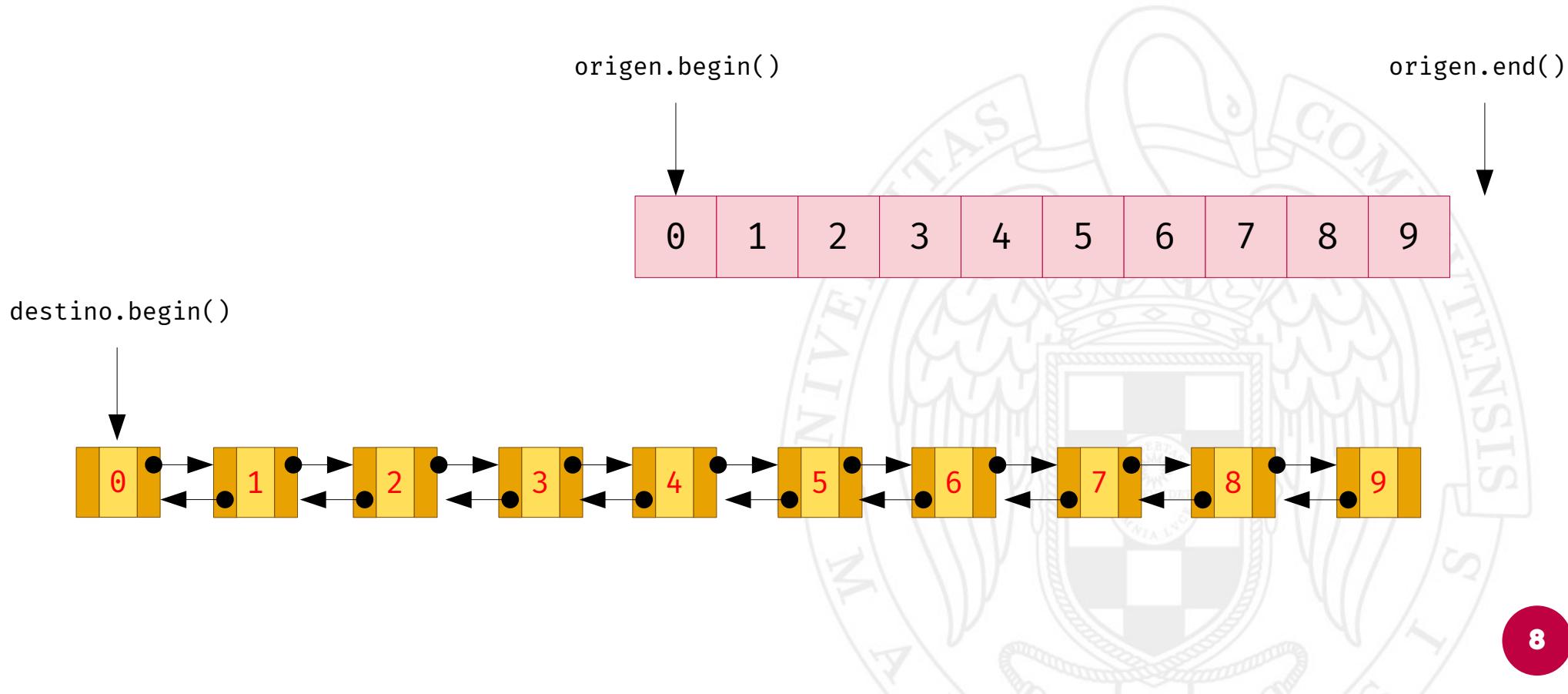
Ejemplo

```
copy(origen.begin(), origen.end(), destino.begin());
```



Ejemplo

```
copy(origen.begin(), origen.end(), destino.begin());
```

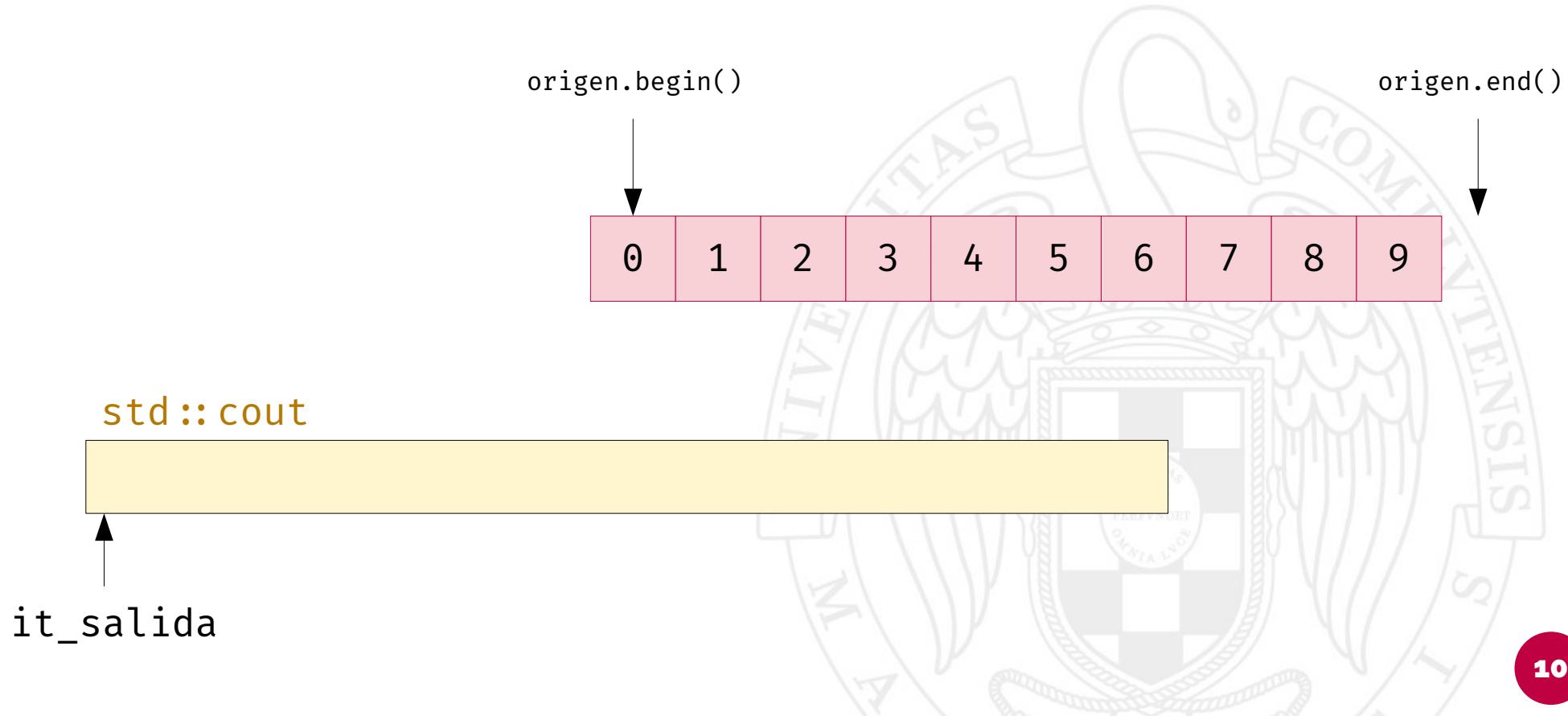


Utilidad de función `copy()`

- 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

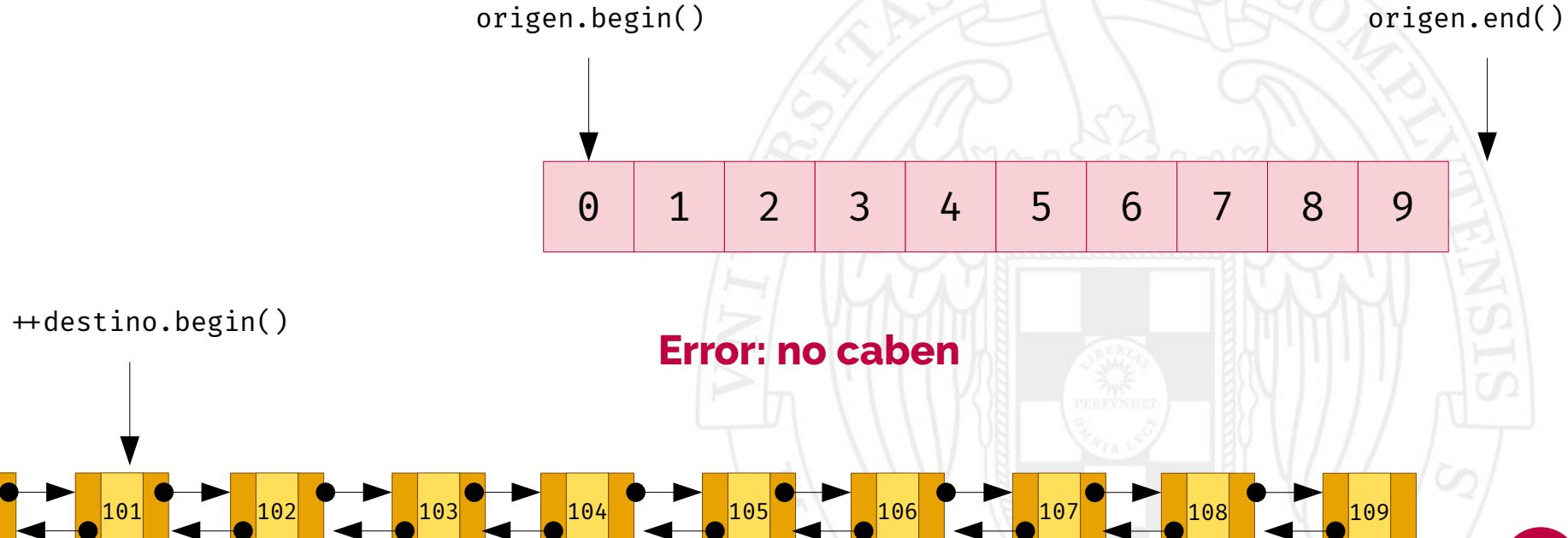
0_1_2_3_4_5_6_7_8_9_

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

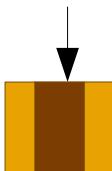
```
int main() {
    vector<int> origen;
    list<int> lista_destino;

    // 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

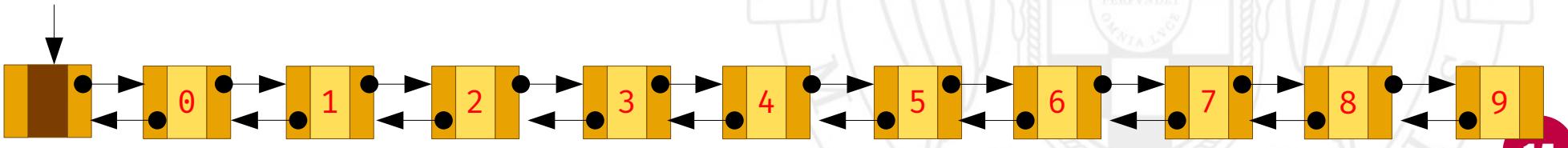
```
int main() {
    vector<int> origen;
    list<int> lista_destino;

    // 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.

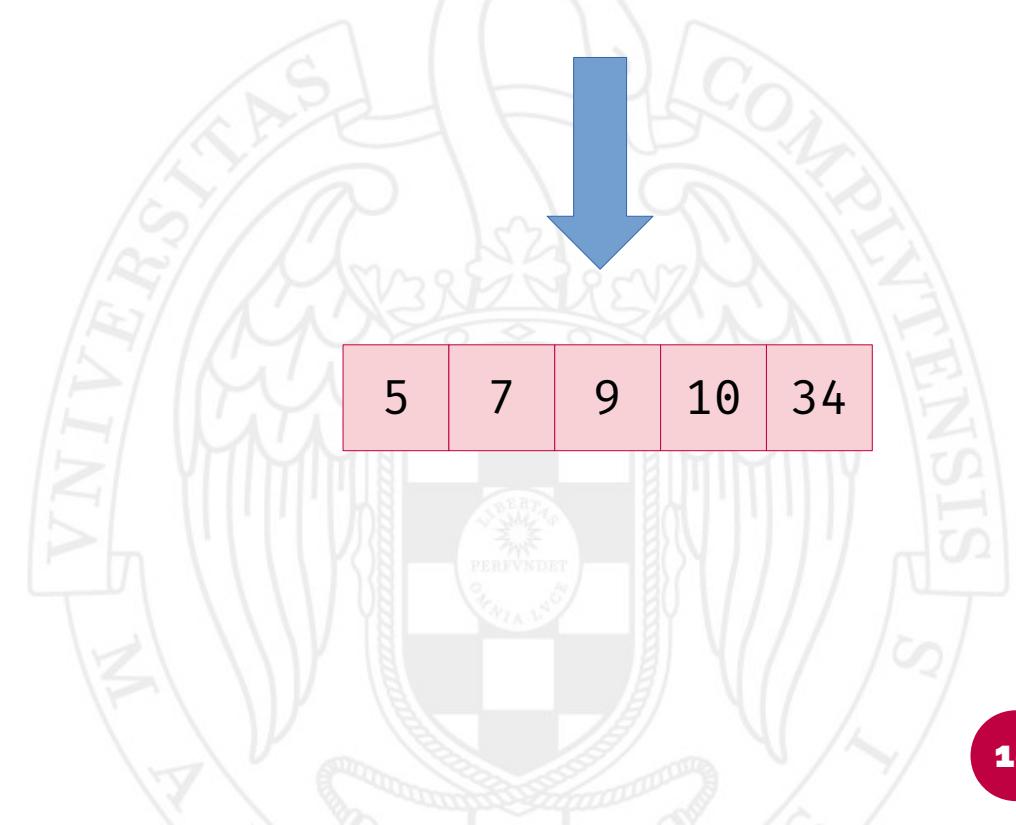
Ejemplo

```
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 *`.

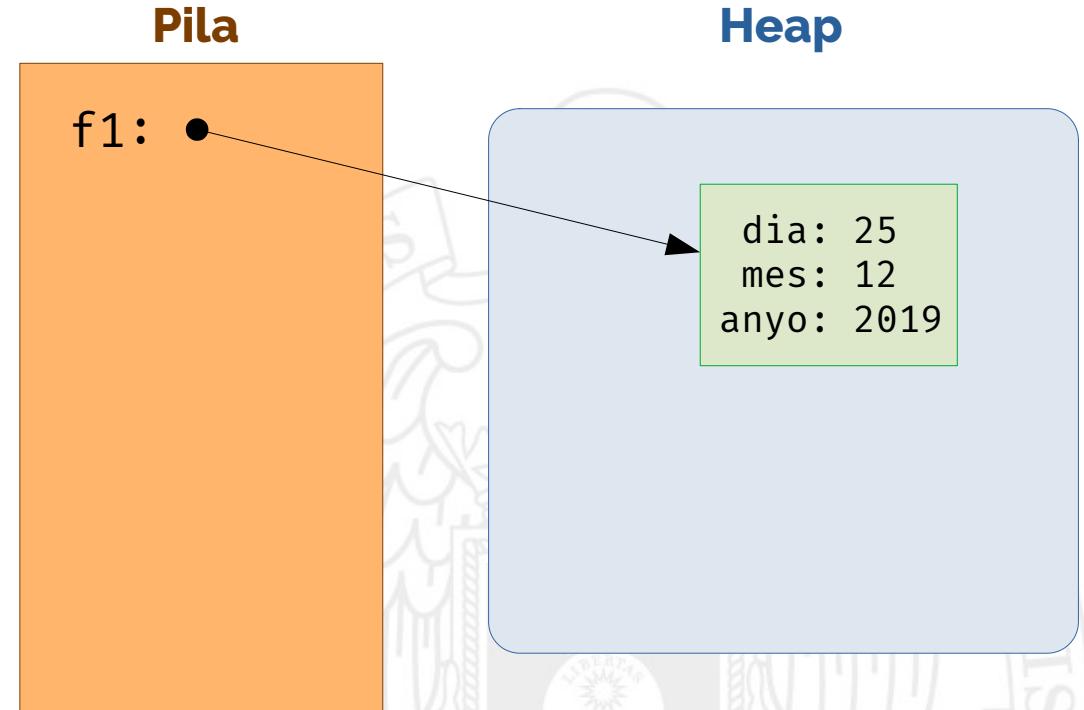
- Ejemplo: crear un objeto en el heap mediante un puntero exclusivo:

```
std::make_unique<Fecha>(25, 12, 2019)
```

Esto devuelve un objeto de tipo `std::unique_ptr<Fecha>`.

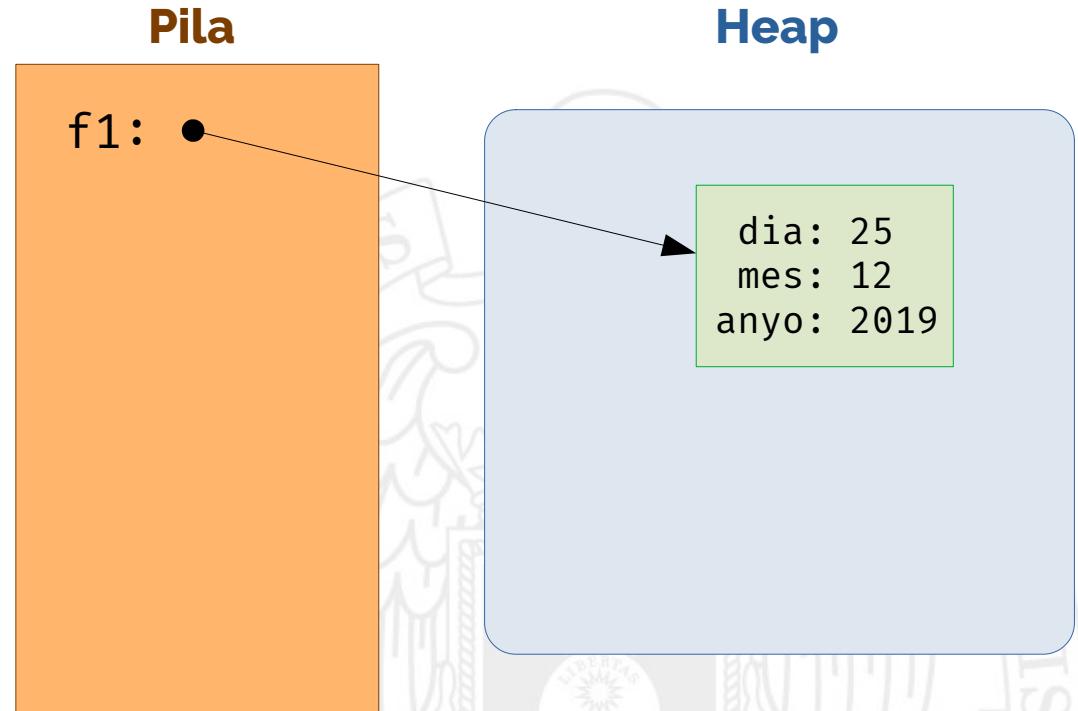
Ejemplo

```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);
```



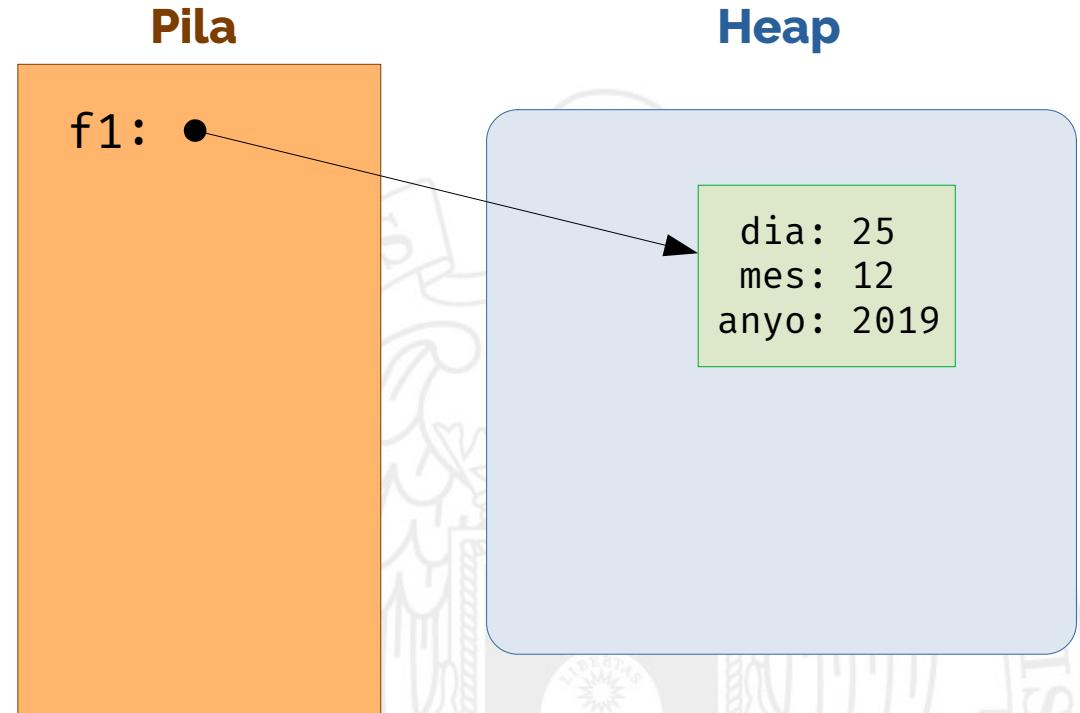
Ejemplo

```
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

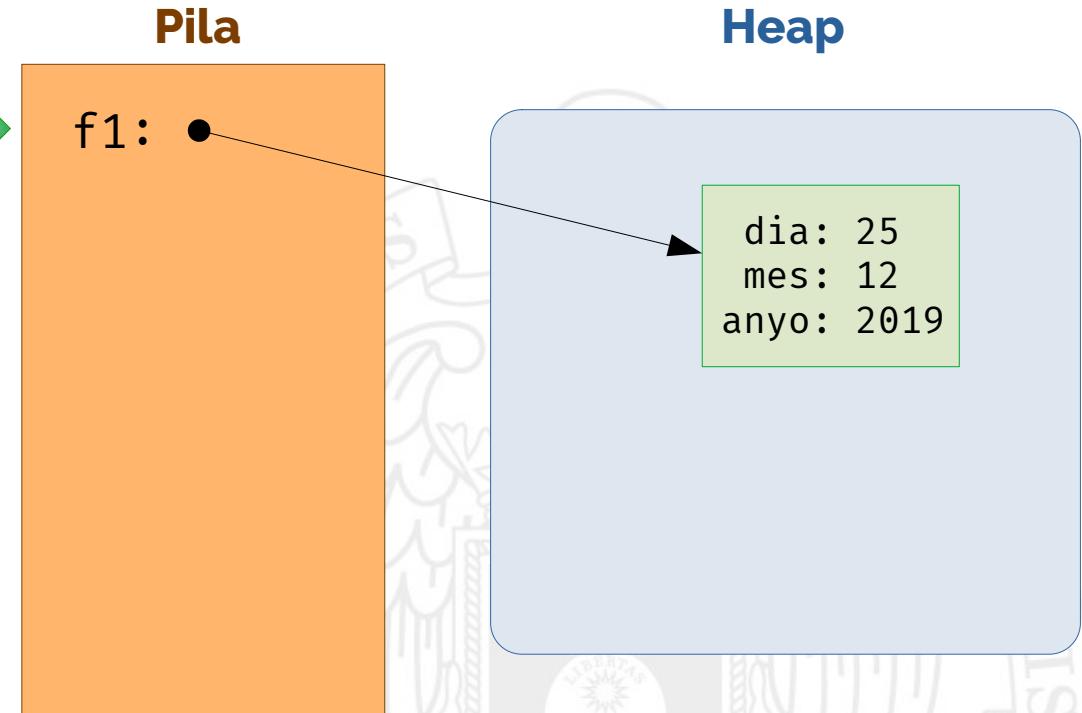
```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);  
  
std::unique_ptr<Fecha> f2 = f1; 
```



Un unique_ptr puede ser transferido

```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);
```

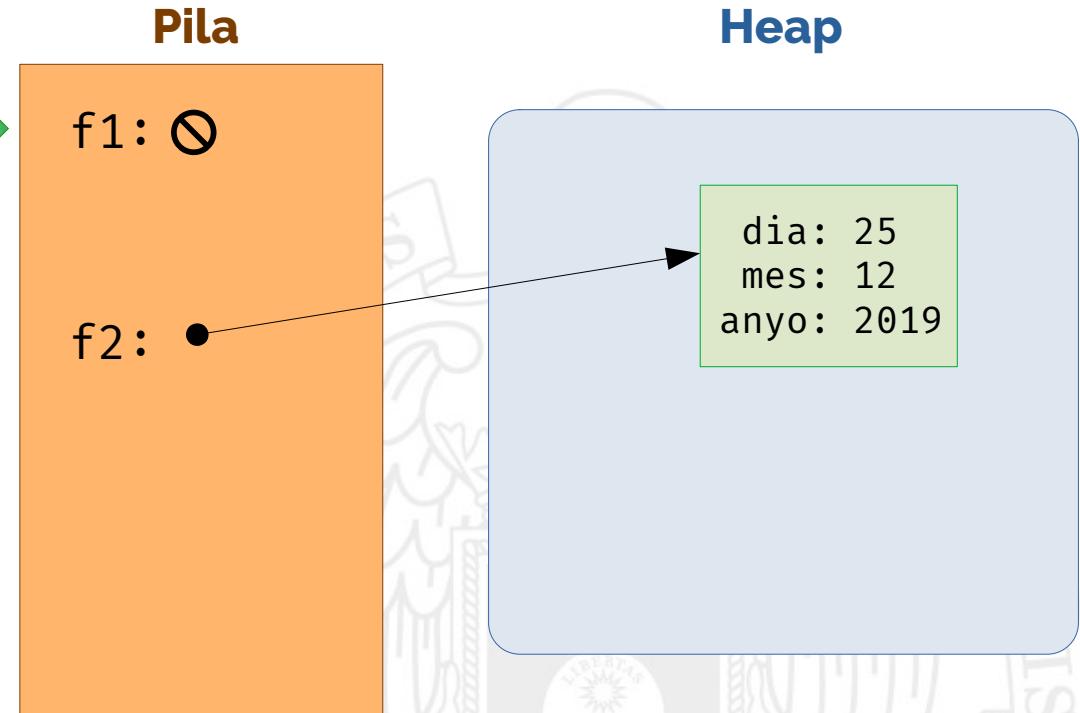
```
std::unique_ptr<Fecha> f2 = std::move(f1); ✓
```



Un unique_ptr puede ser transferido

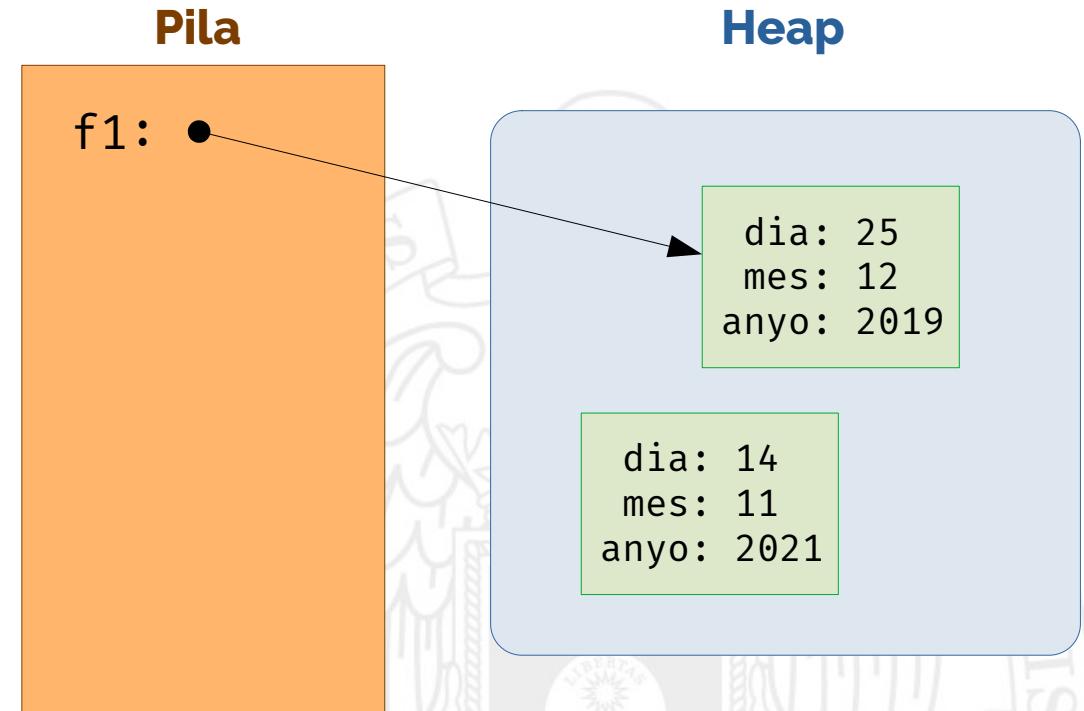
```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);
```

```
std::unique_ptr<Fecha> f2 = std::move(f1); ✓
```



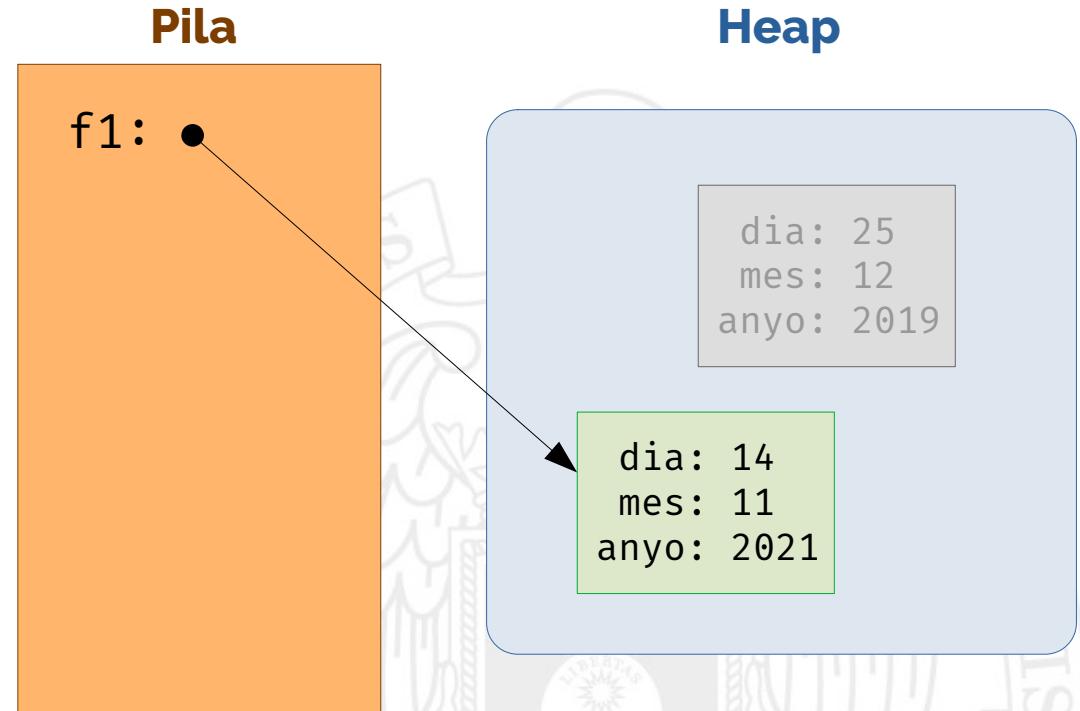
Reasignando un unique_ptr

```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);  
  
f1 = std::make_unique<Fecha>(14, 11, 2021);
```



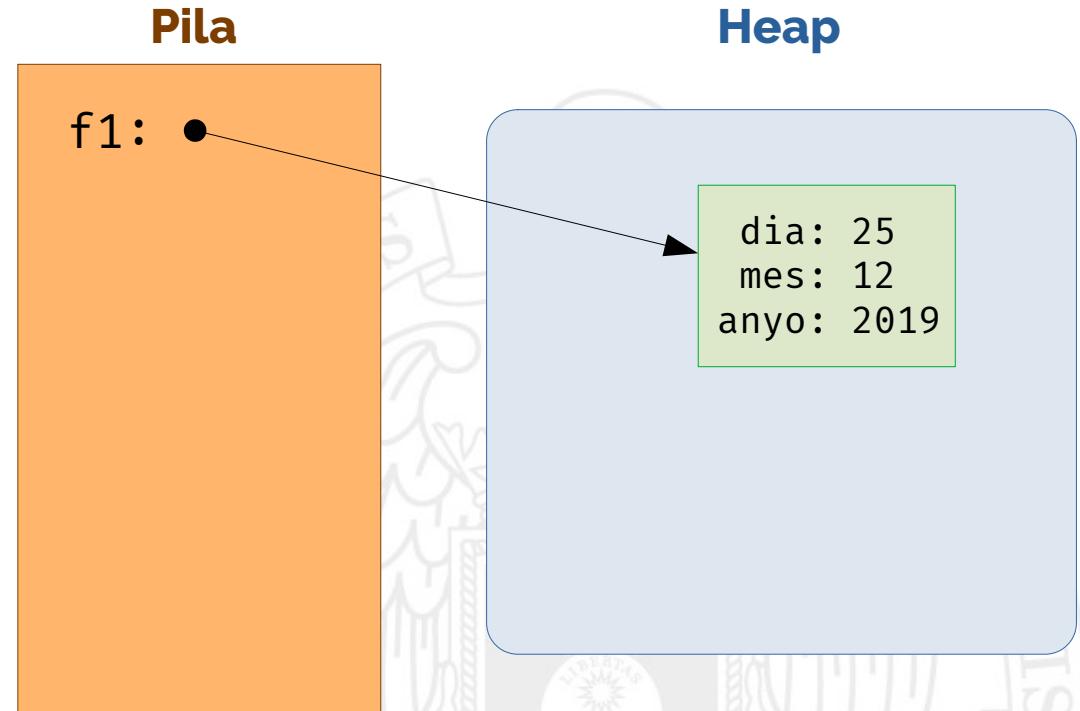
Reasignando un unique_ptr

```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);  
  
f1 = std::make_unique<Fecha>(14, 11, 2021);
```



Reasignando un unique_ptr

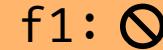
```
std::unique_ptr<Fecha> f1 =  
    std::make_unique<Fecha>(25, 12, 2019);  
  
f1 = nullptr;
```



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:

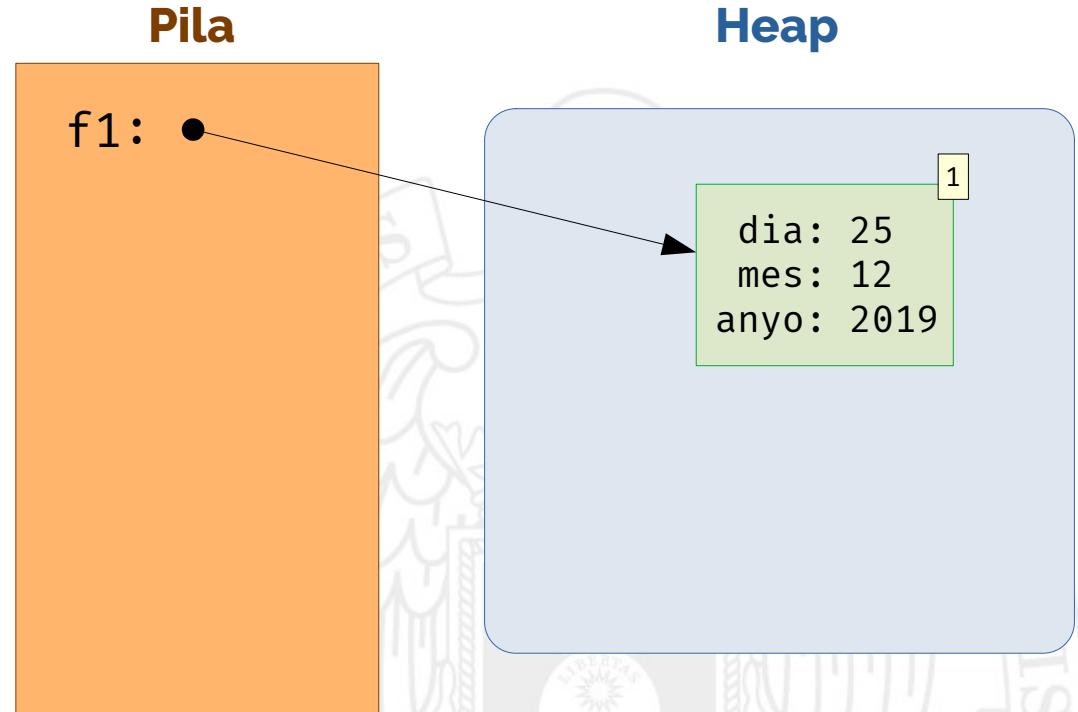
```
std :: make_shared<Fecha>(25, 12, 2019)
```

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.

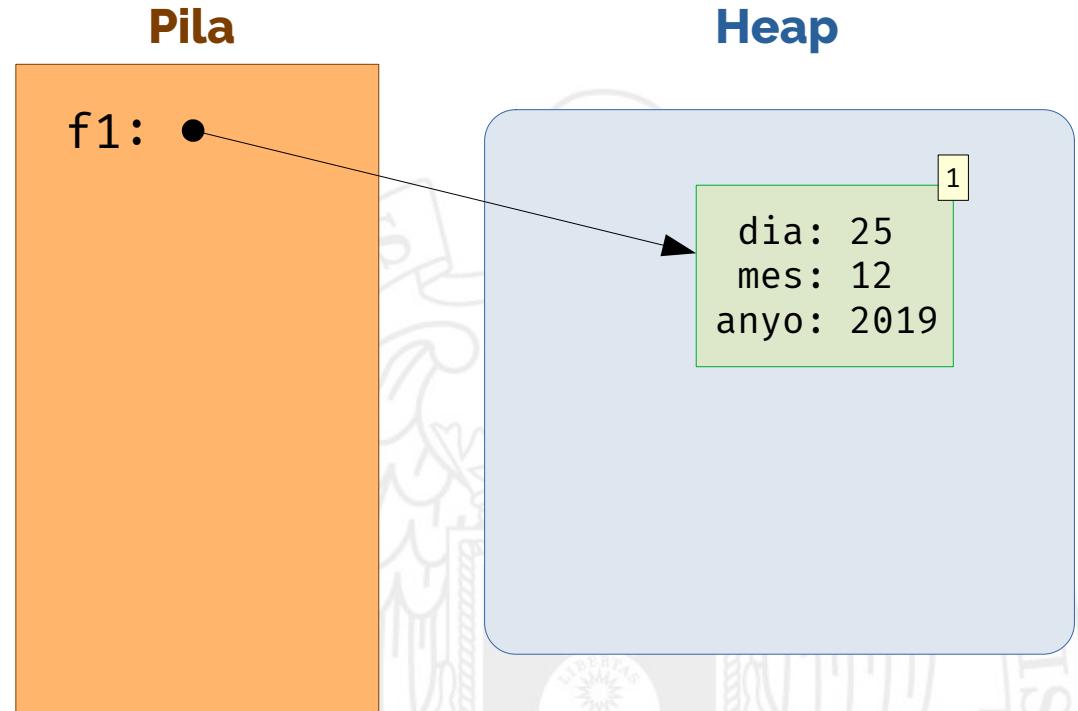
Ejemplo

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);
```



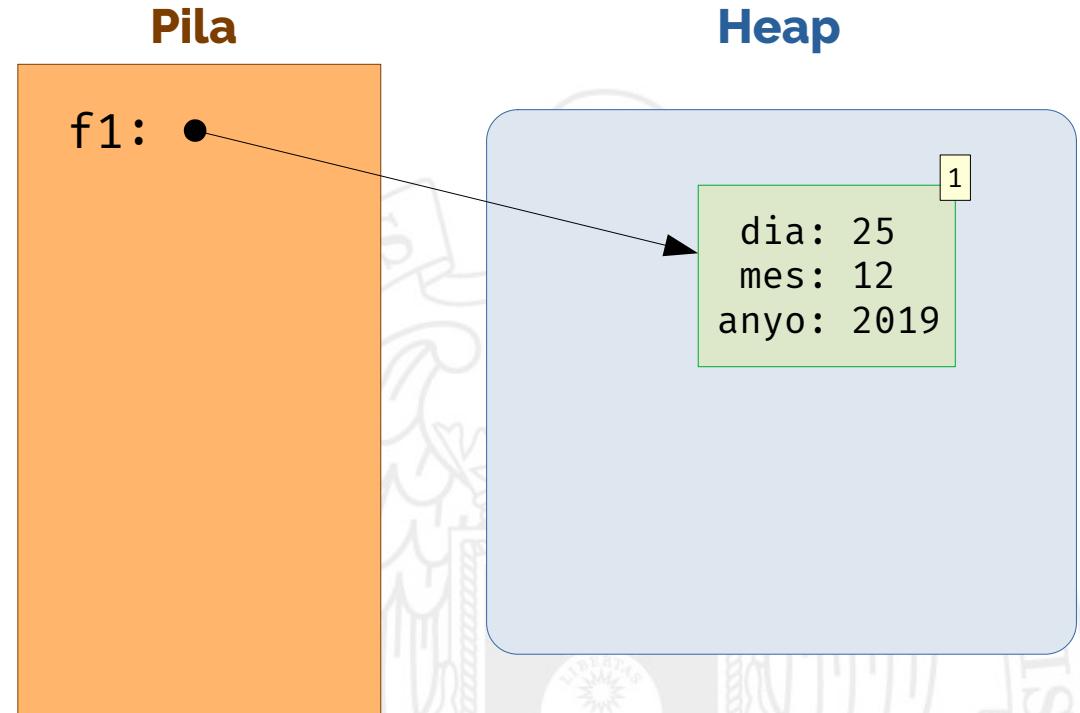
Ejemplo

```
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;  
}
```



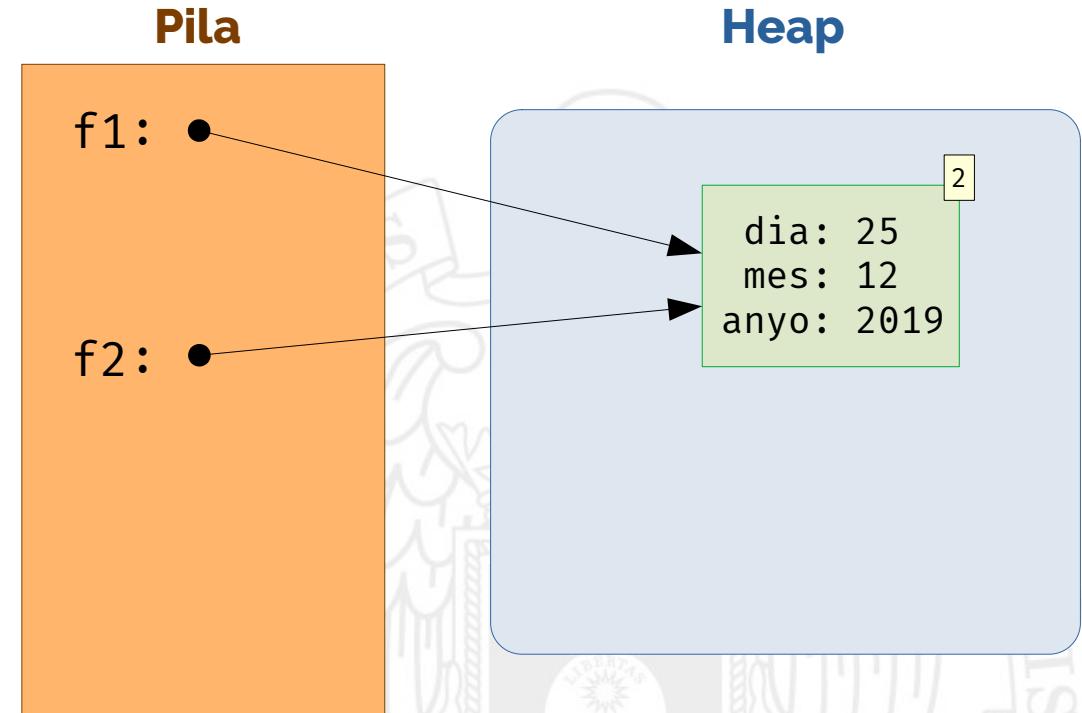
Copia de un shared_ptr

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
std::shared_ptr<Fecha> f2 = f1;
```



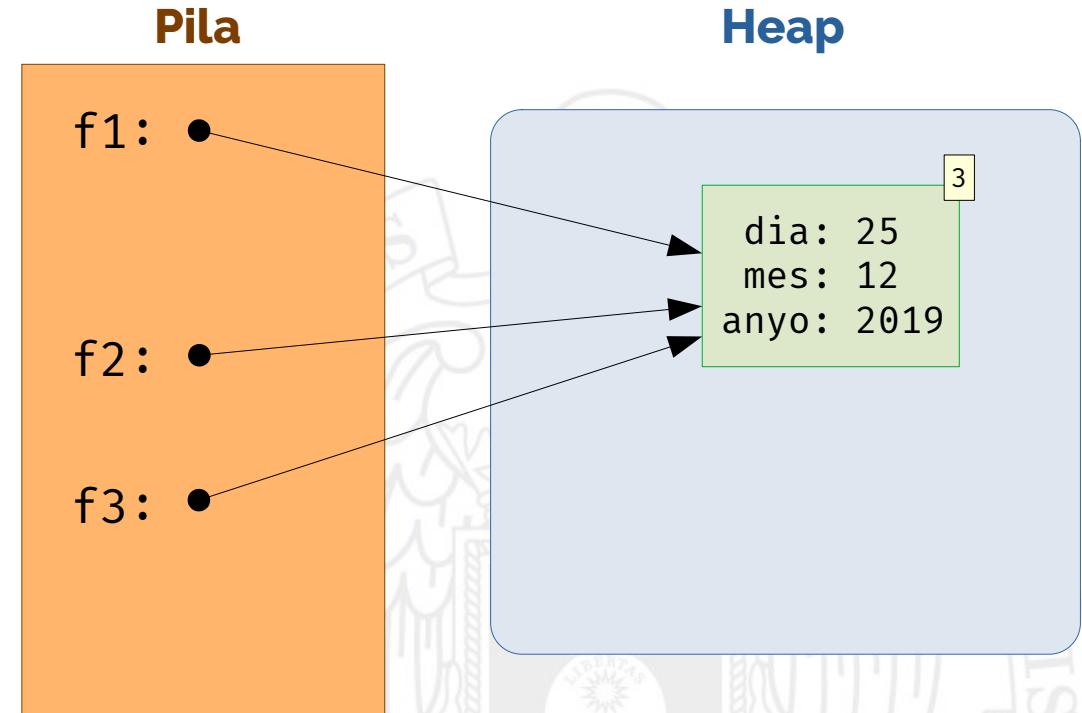
Copia de un shared_ptr

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
std::shared_ptr<Fecha> f2 = f1;
```



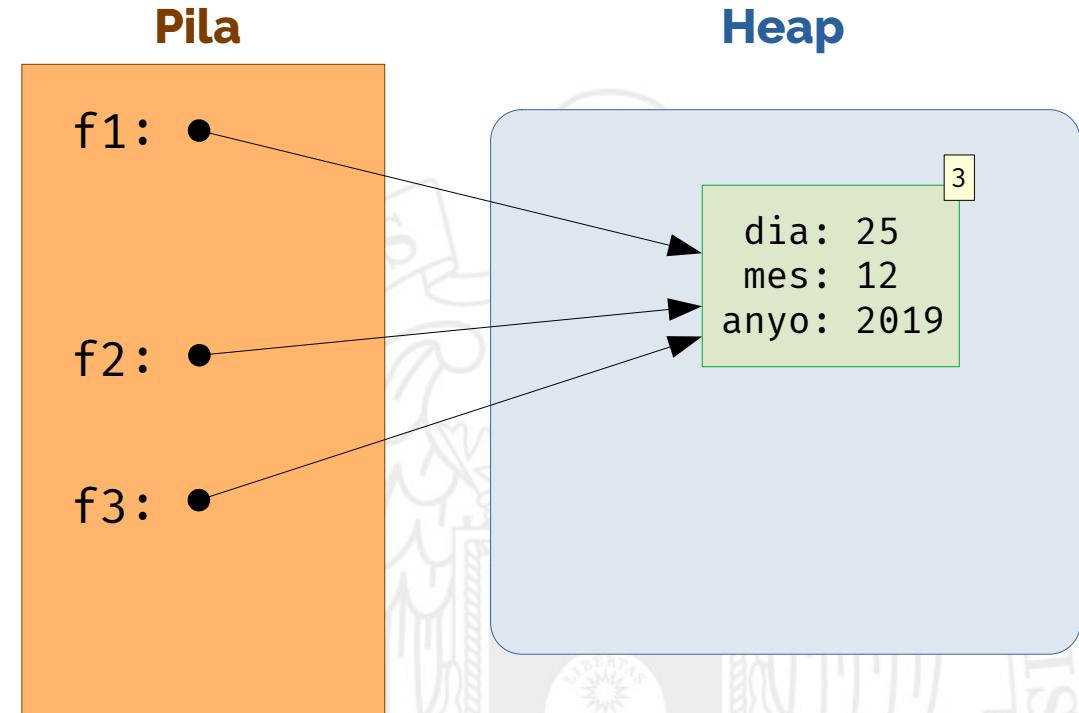
Copia de un shared_ptr

```
std::shared_ptr<Fecha> f1 =  
    std::make_shared<Fecha>(25, 12, 2019);  
  
std::shared_ptr<Fecha> f2 = f1;  
std::shared_ptr<Fecha> f3 = f2;
```



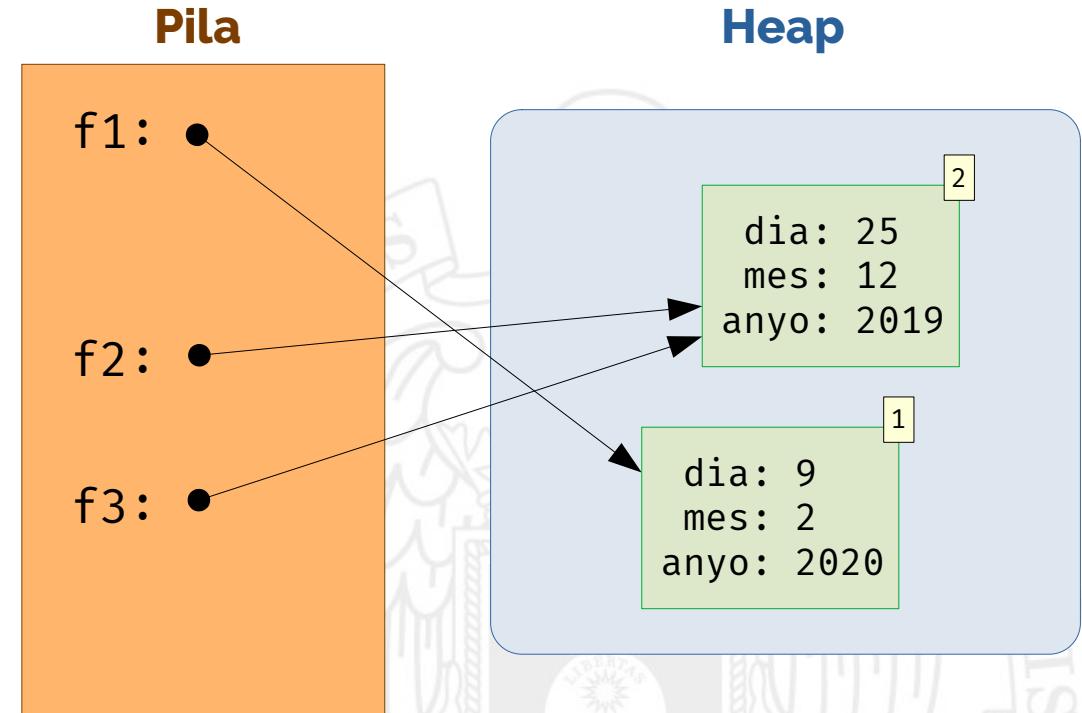
Copia de un shared_ptr

```
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);
```



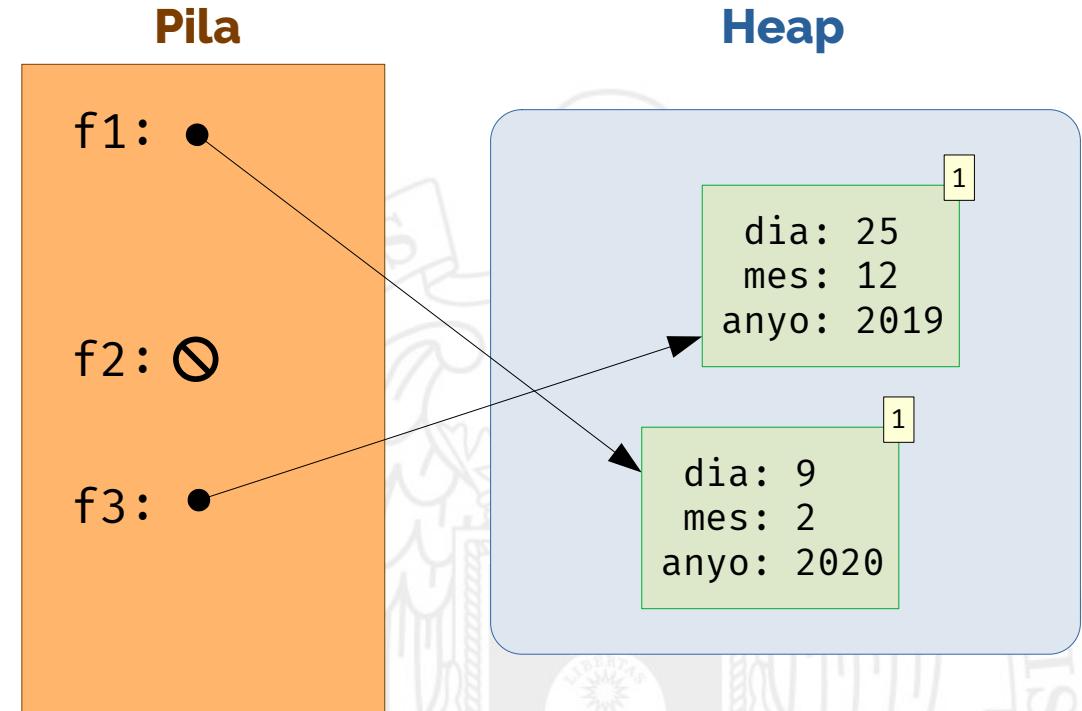
Copia de un shared_ptr

```
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);
```



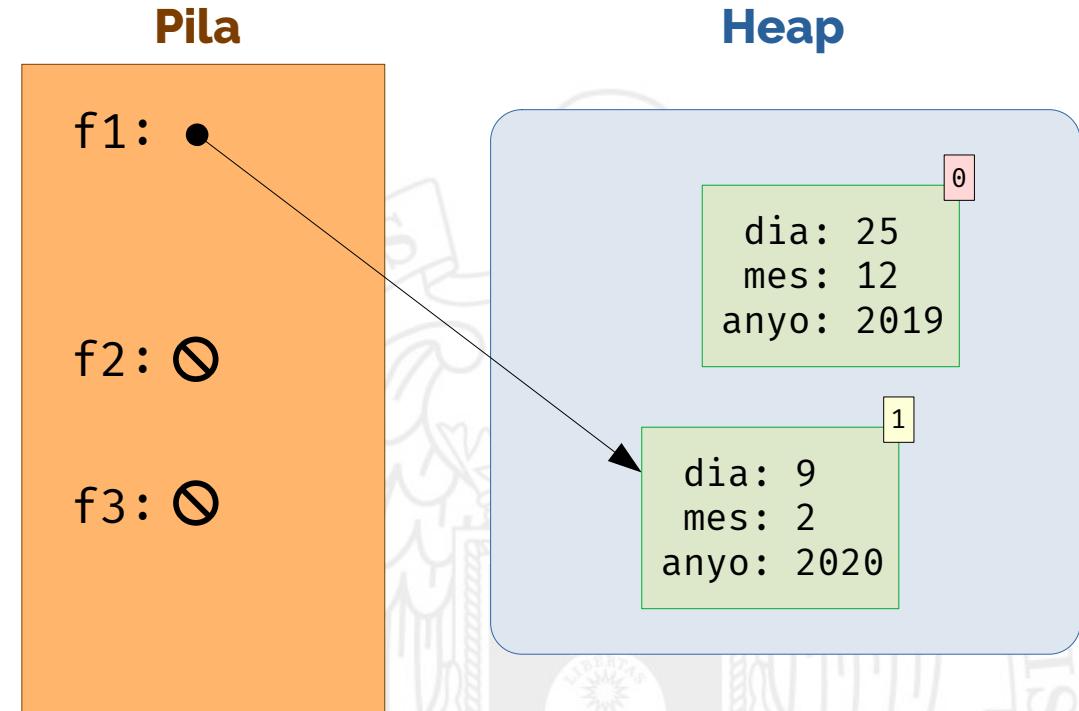
Copia de un shared_ptr

```
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;
```



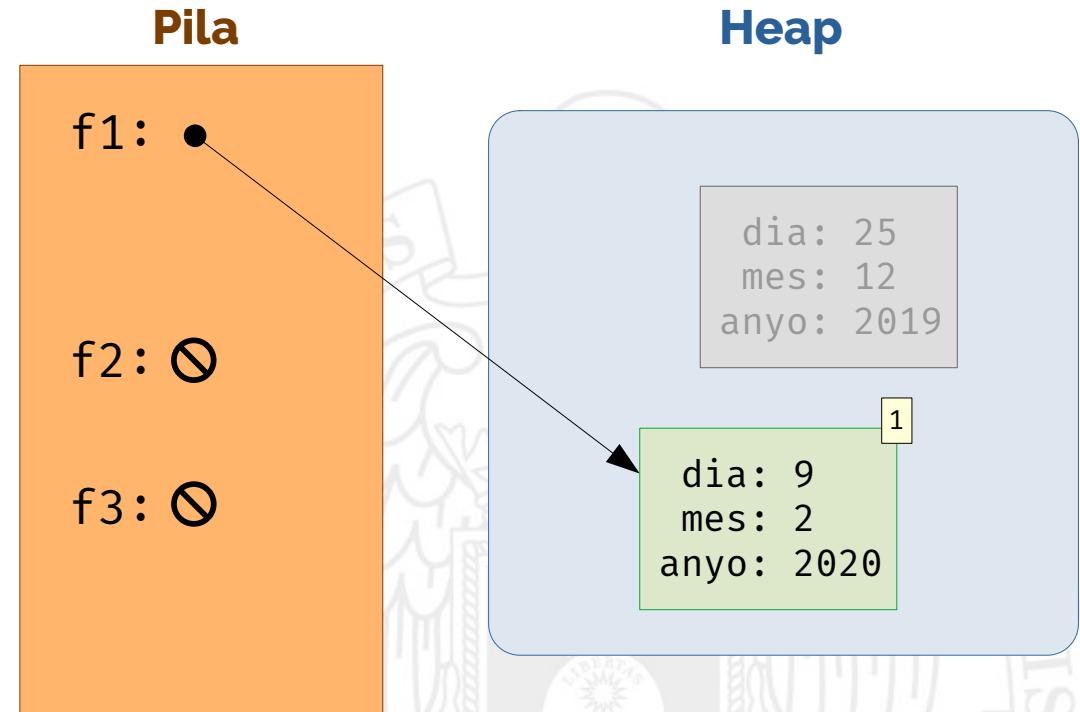
Copia de un shared_ptr

```
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;
```

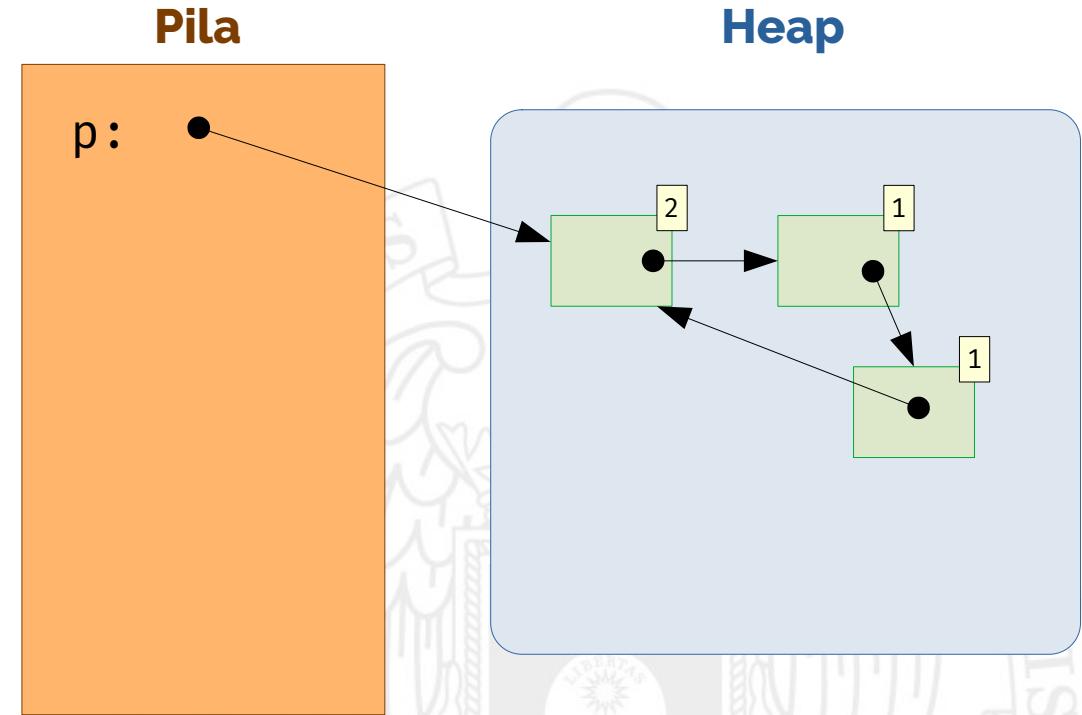


Copia de un shared_ptr

```
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!



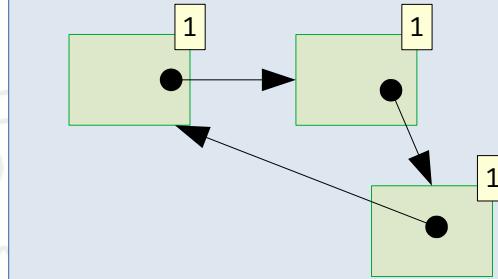
¡Cuidado con las referencias circulares!

```
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;
```

[1, 5, 9]

```
eliminar(v2, es_impar);
std::cout << v2 << std::endl;
```

[6, 10, 20]

```
std::list<int> v3 = {-2, 3, 10, -6, 20};
eliminar(v3, es_positivo);
std::cout << v3 << std::endl;
```

[-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

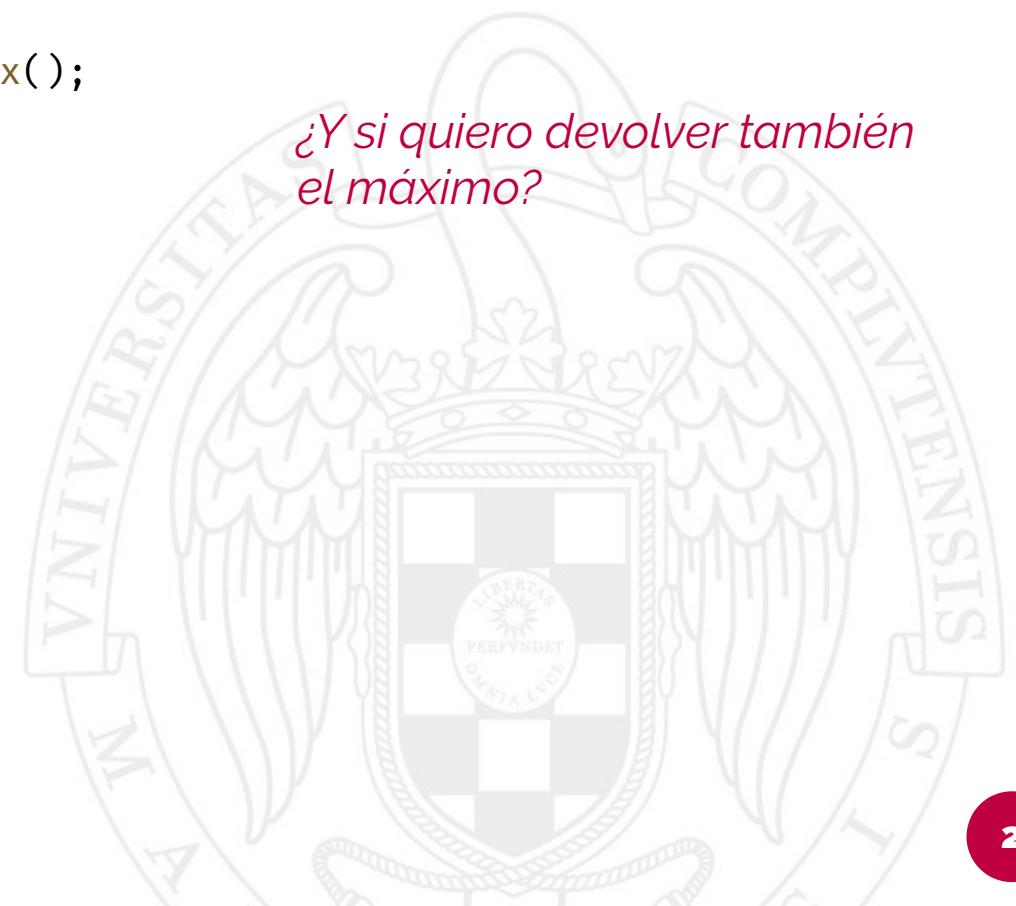
Departamento de Sistemas Informáticos y Computación
Facultad de Informática – Universidad Complutense de Madrid

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.

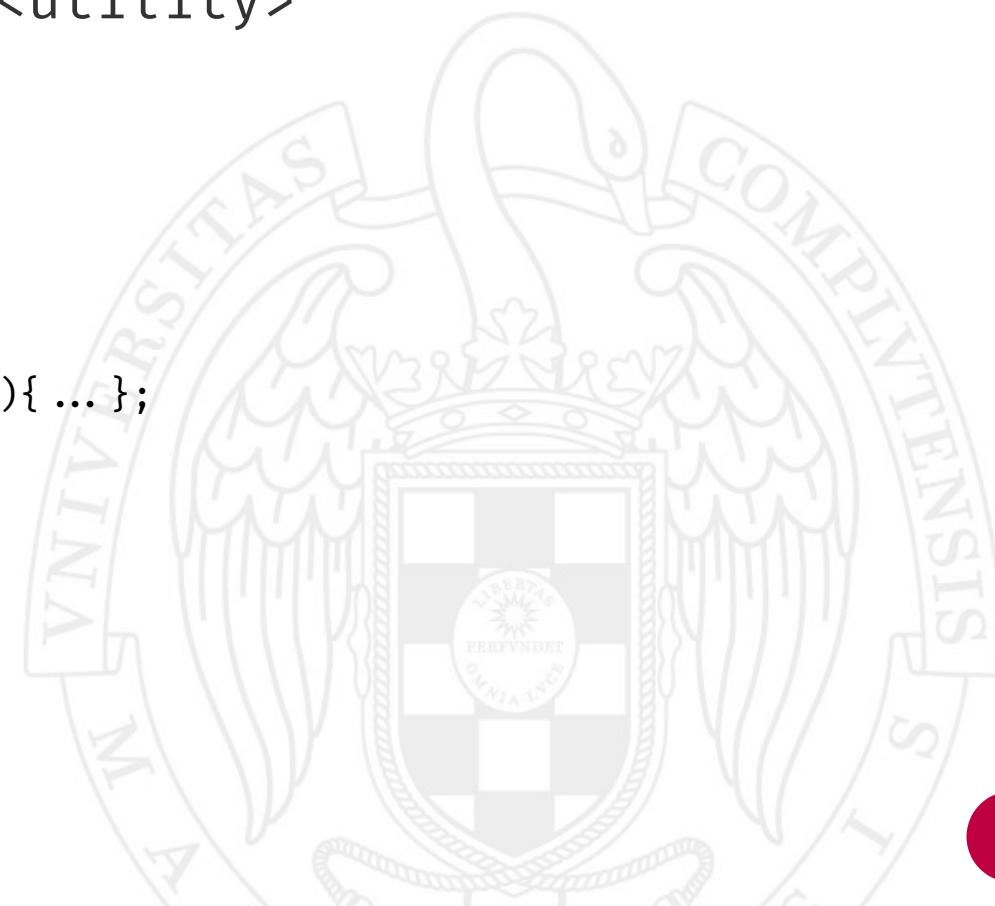
Pares – std :: pair

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
```

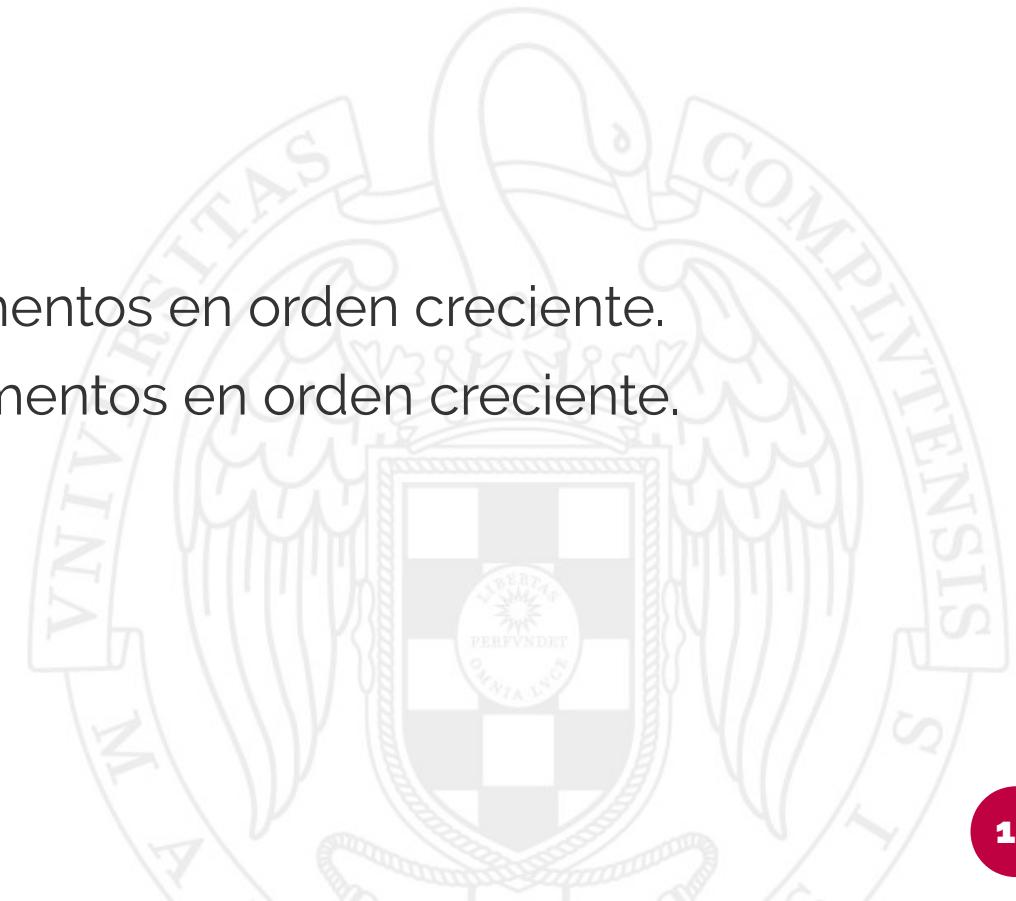
Otras funciones de orden superior

- `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
```

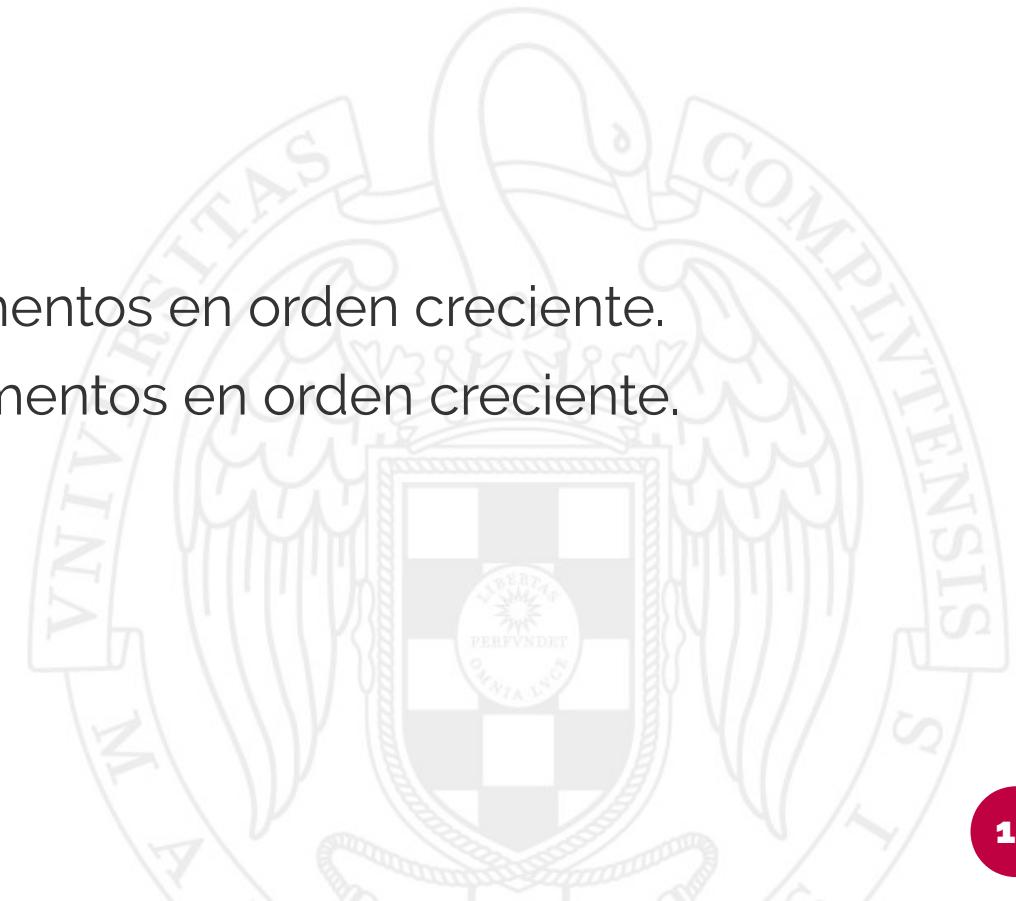
Otras funciones de orden superior

- `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**.

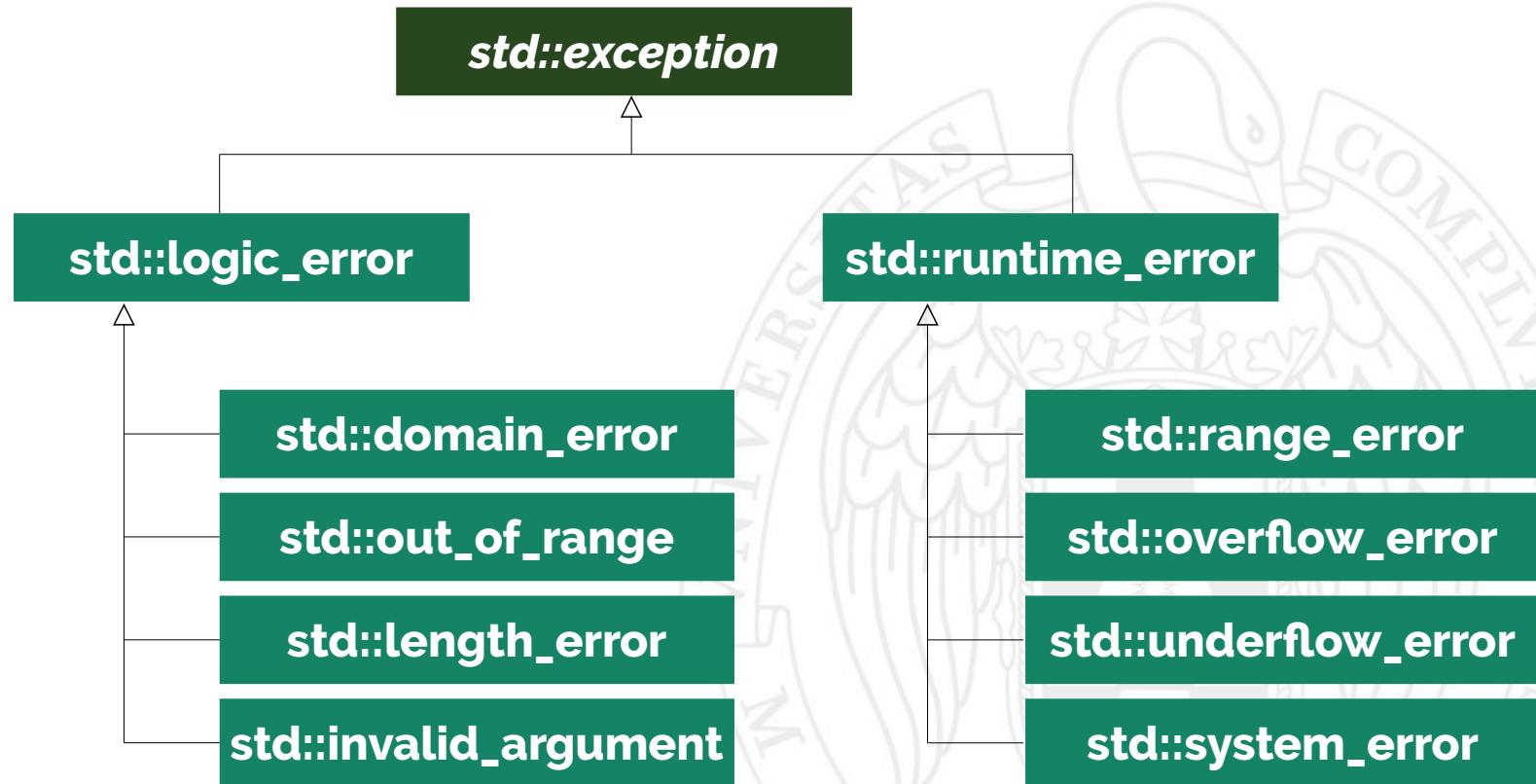
```
try {  
    dividir(1, 0);  
} catch (division_por_cero &e) {  
    std::cout << "División por cero!" << std::endl;  
}
```

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.

