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?

En C++ nosotros tenemos que liberar la memoria que reservamos. Al contrario que en Java.

- 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`.

En java no son necesarios estos tipos de punteros

- 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.

Además lleva la cuenta de todos ellos.

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:

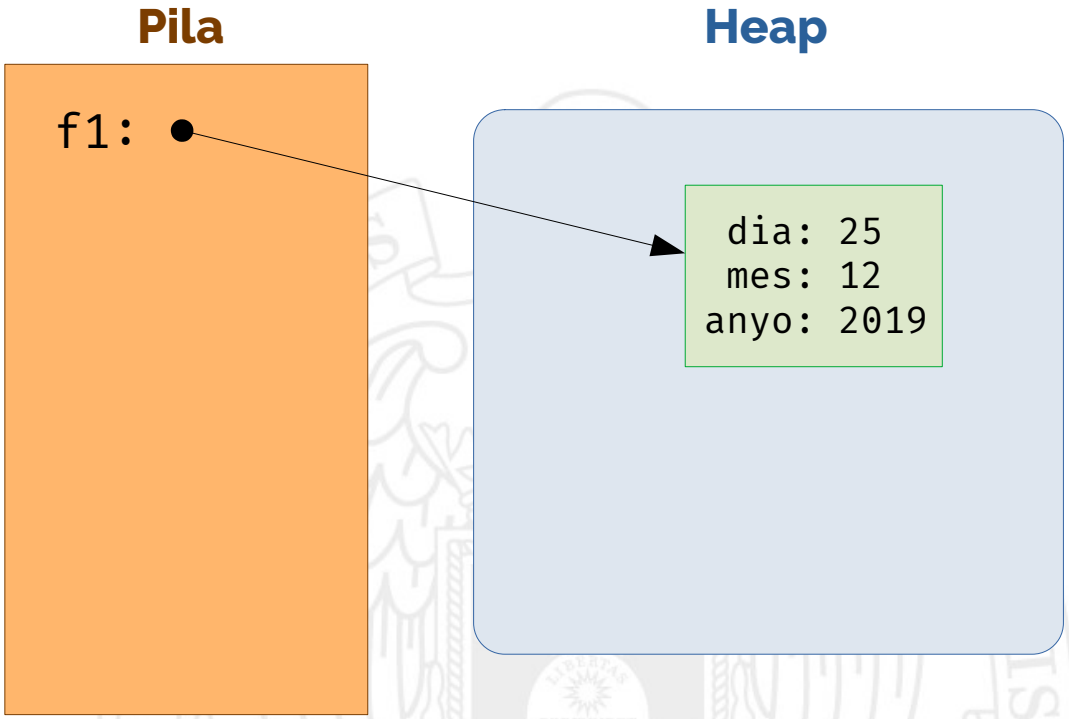
```
std::make_unique<Fecha>(25, 12, 2019)
```

Parámetros del constructor de fecha.

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

Ejemplo

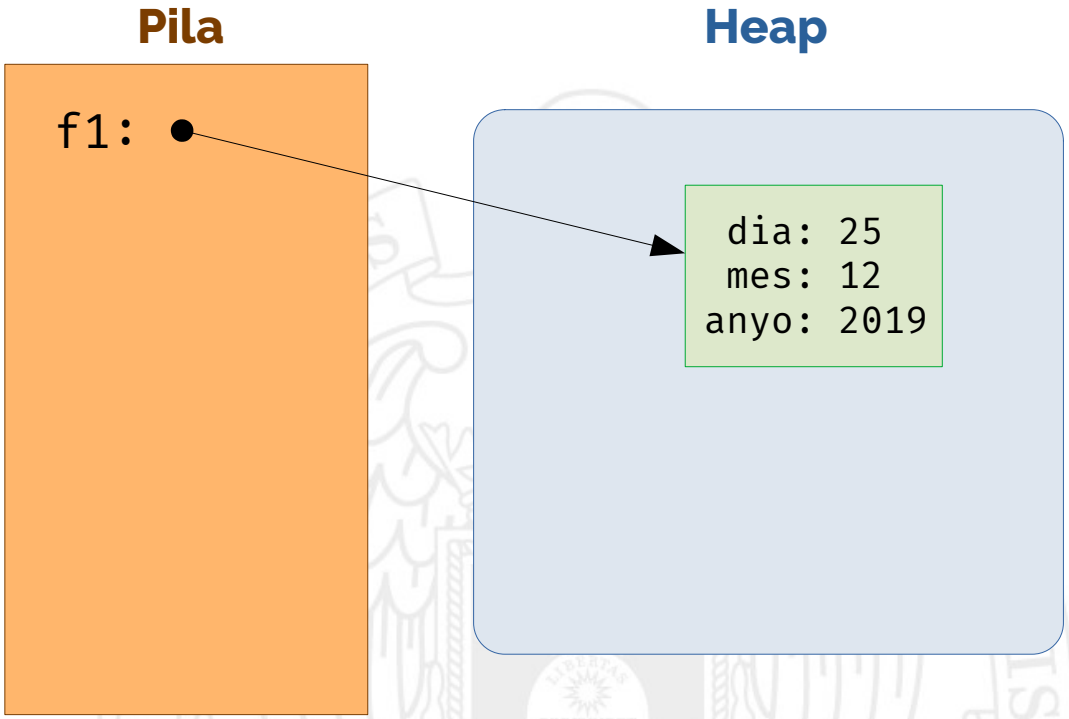
```
std::unique_ptr<Fecha> f1 =
  std::make_unique<Fecha>(25, 12, 2019);
```



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;
}
```

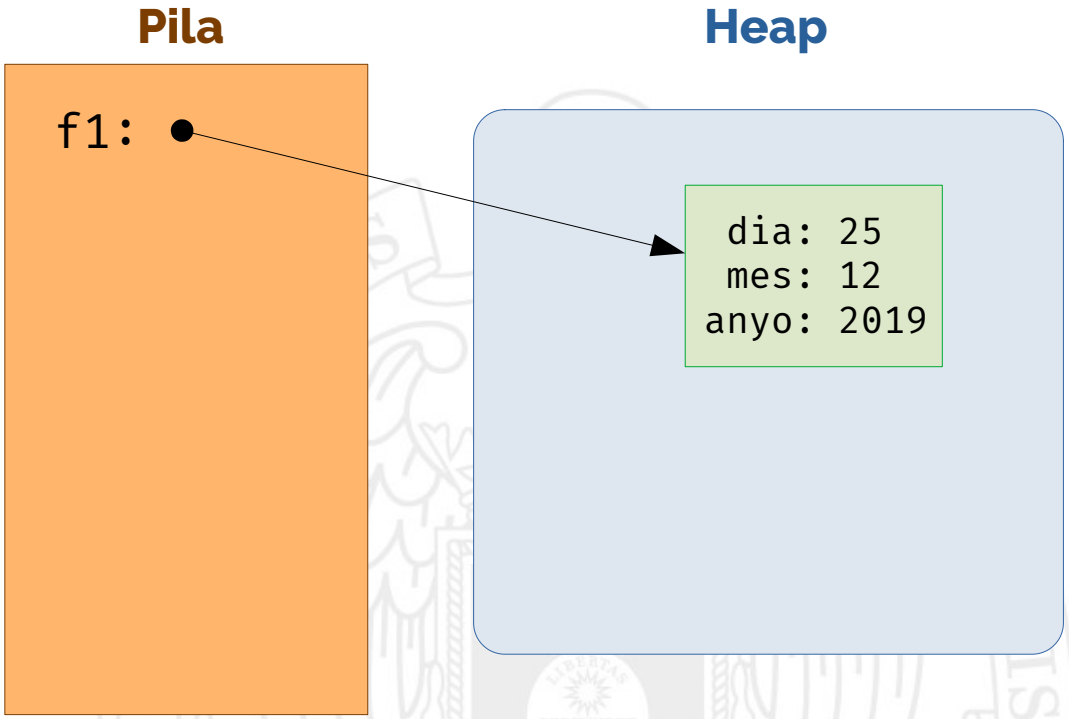
Deberíamos de sobrecargarlos para poder utilizarlos como punteros normales.



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;
```

Tiene sentido porque si no dos punteros podrían estar apuntando al mismo sitio, que es lo que queremos evitar

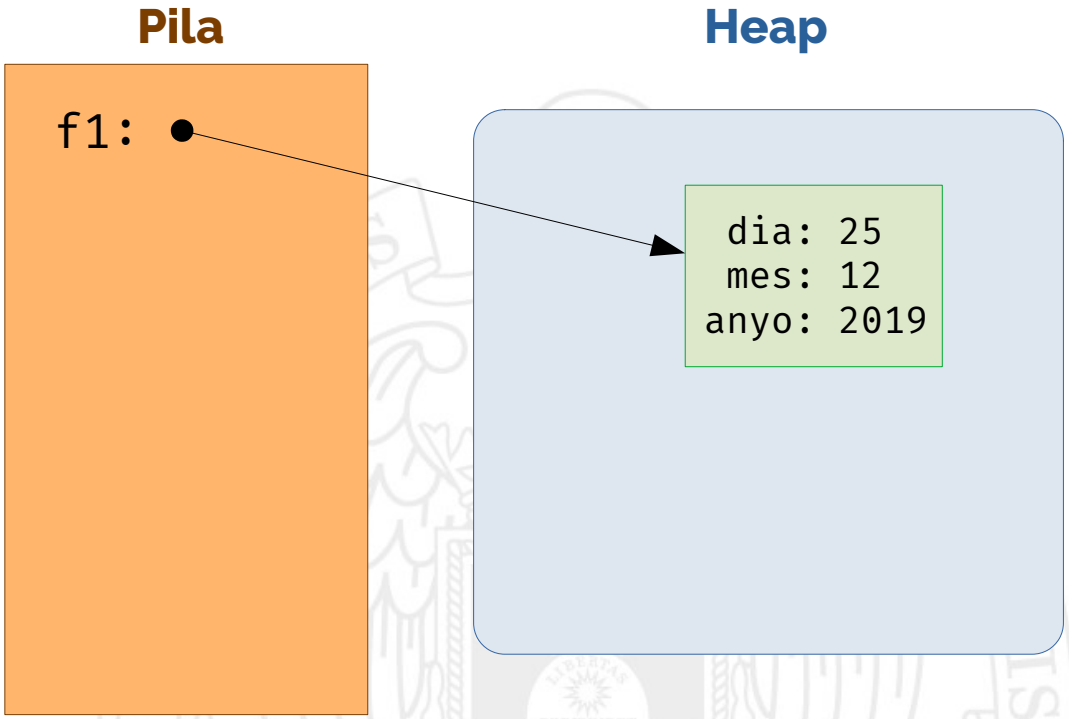


Un `unique_ptr` puede ser transferido

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

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

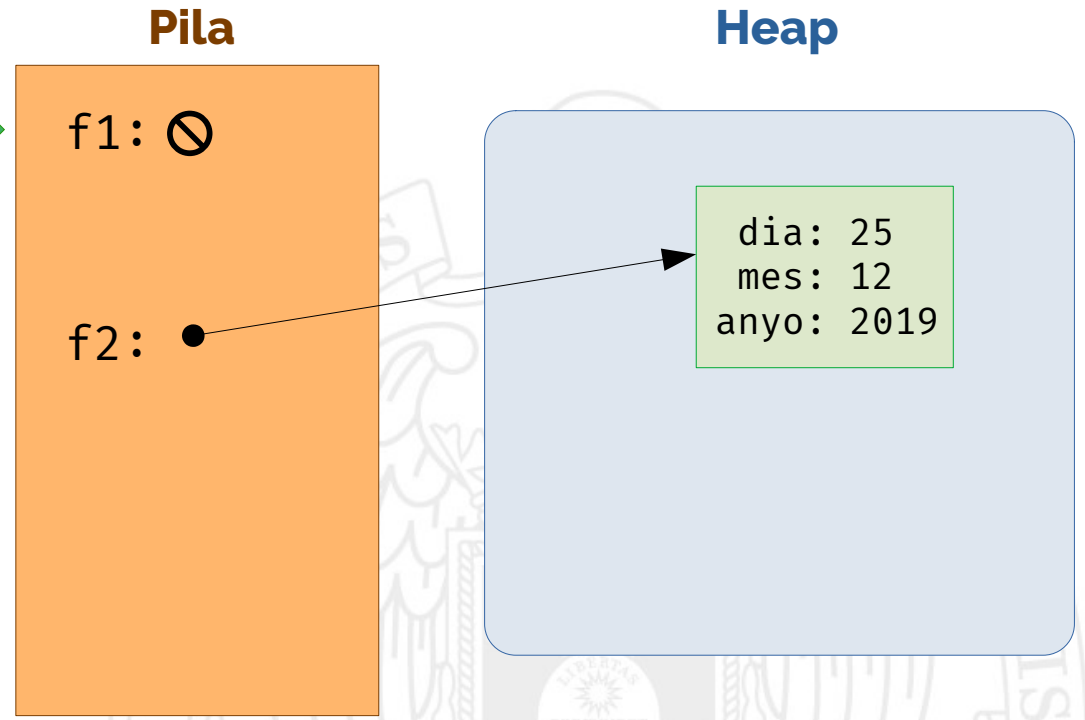
Transferir el puntero a otro unique_pointer distinto

Move es el constructor de movimiento. Es uno de los 5 tipos de constructores que aun no hemos visto.



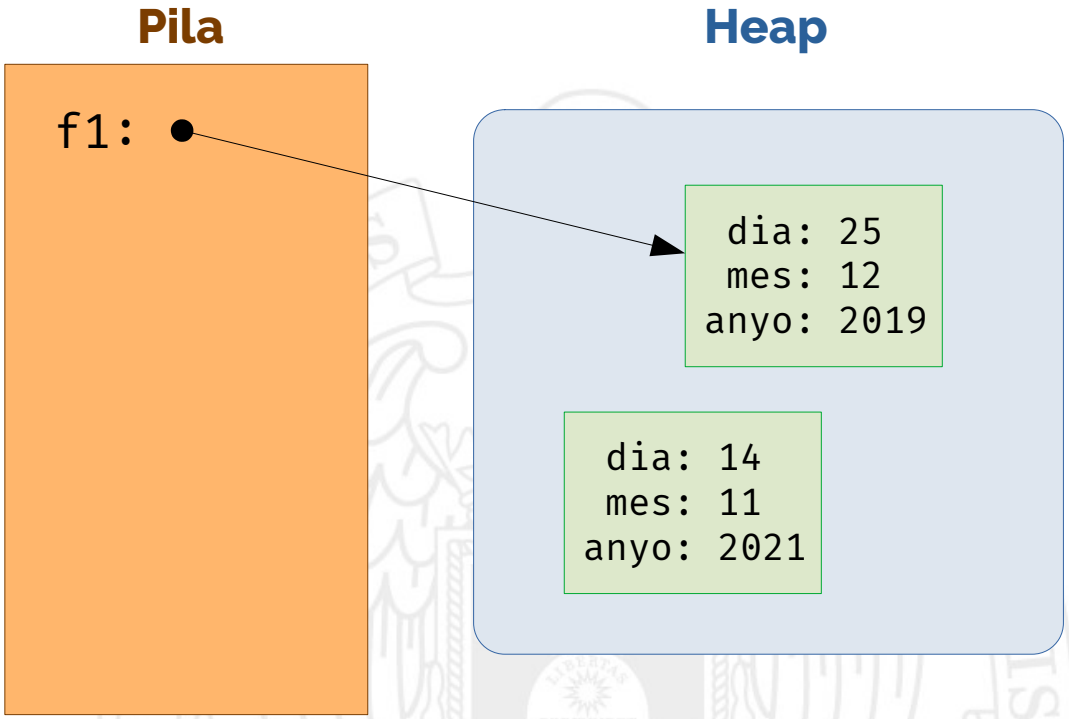
Un unique_ptr puede ser transferido

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

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



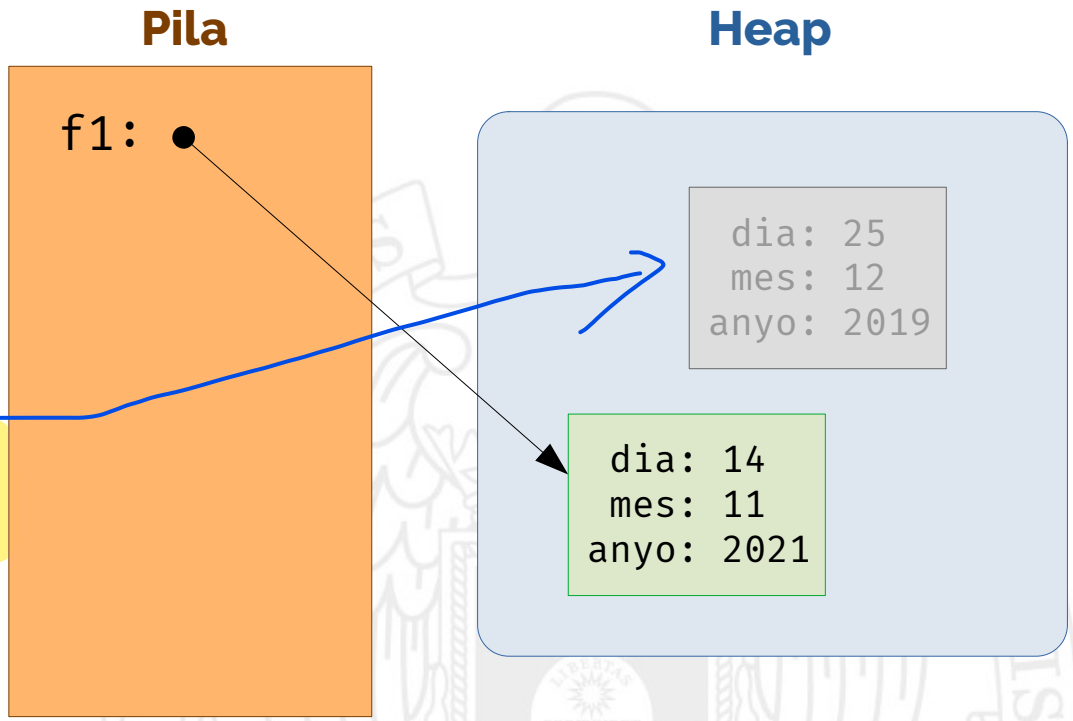
Reasignando un unique_ptr

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

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



Reasignando un unique_ptr

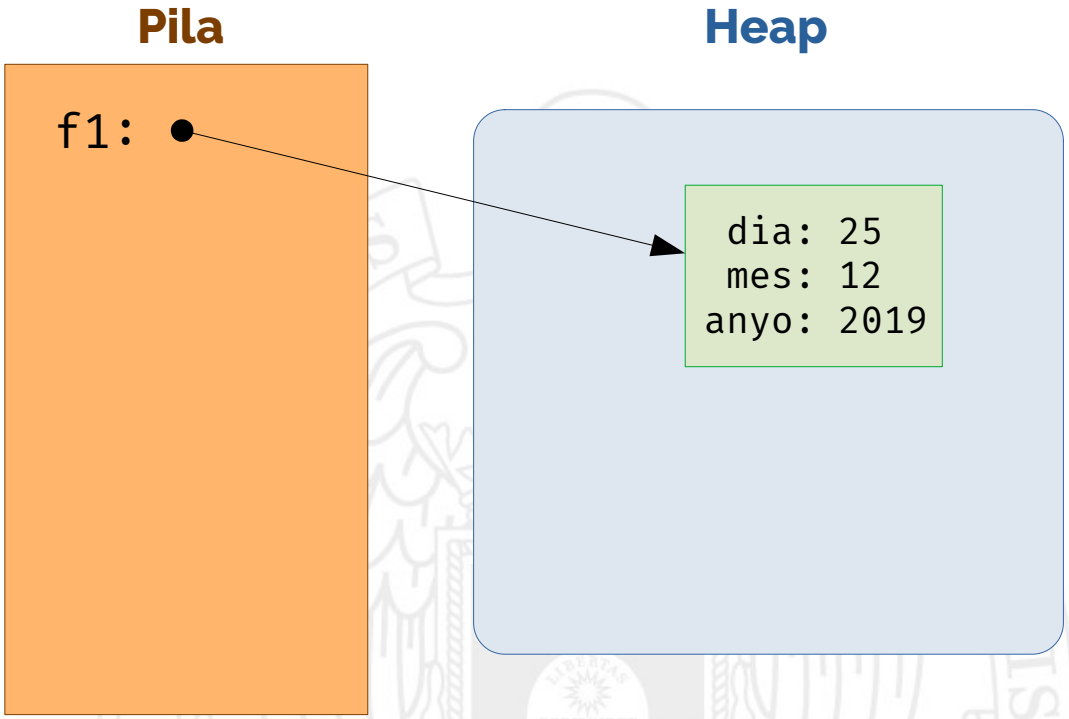
```
std::unique_ptr<Fecha> f1 =
  std::make_unique<Fecha>(25, 12, 2019);

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



Hace el delete de manera automática porque sabe que no hay ningún otro puntero apuntando a esa posición. No hace falta que lo hagamos nosotros.

Reasignando un unique_ptr

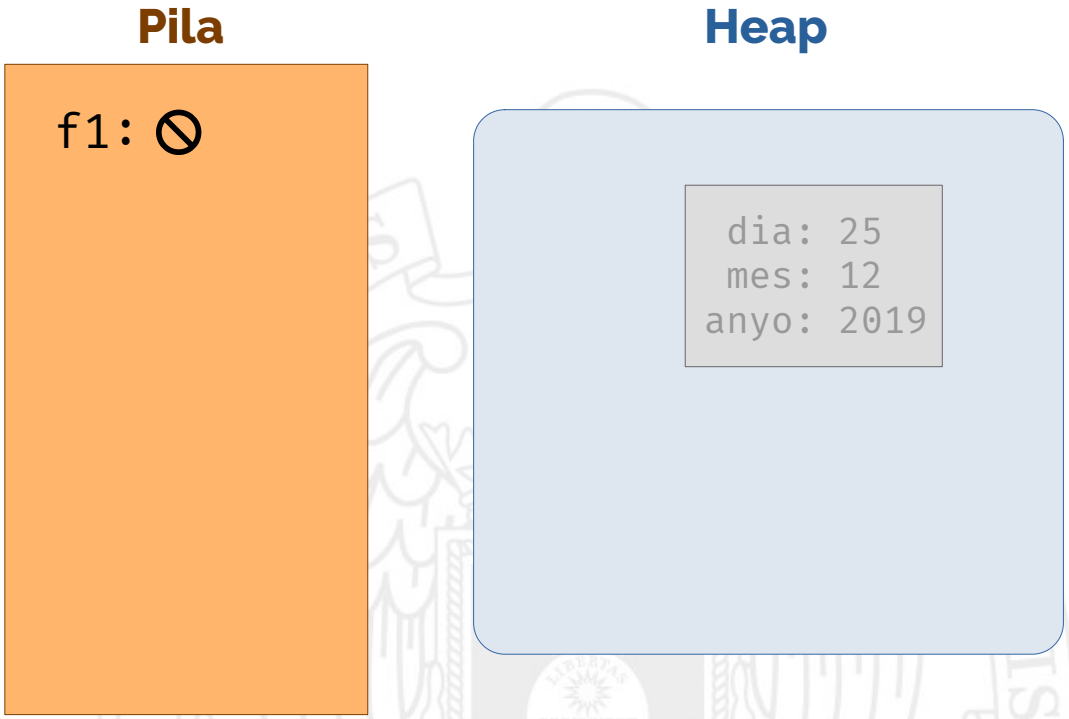
```
std::unique_ptr<Fecha> f1 =
  std::make_unique<Fecha>(25, 12, 2019);

f1 = nullptr;
```



Reasignando un unique_ptr

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

f1 = nullptr;
```

Se borraría de manera automática el objeto apuntado y este apuntaría a nullptr.



Hasta aquí los punteros únicos. Vamos a pasar a los punteros compartidos.

Punteros compartidos – `std::shared_ptr`

Crear un `shared_ptr`

Lo de std lo podemos quitar si ponemos using namespace std;

- Para crear un objeto en el heap mediante un puntero compartido:

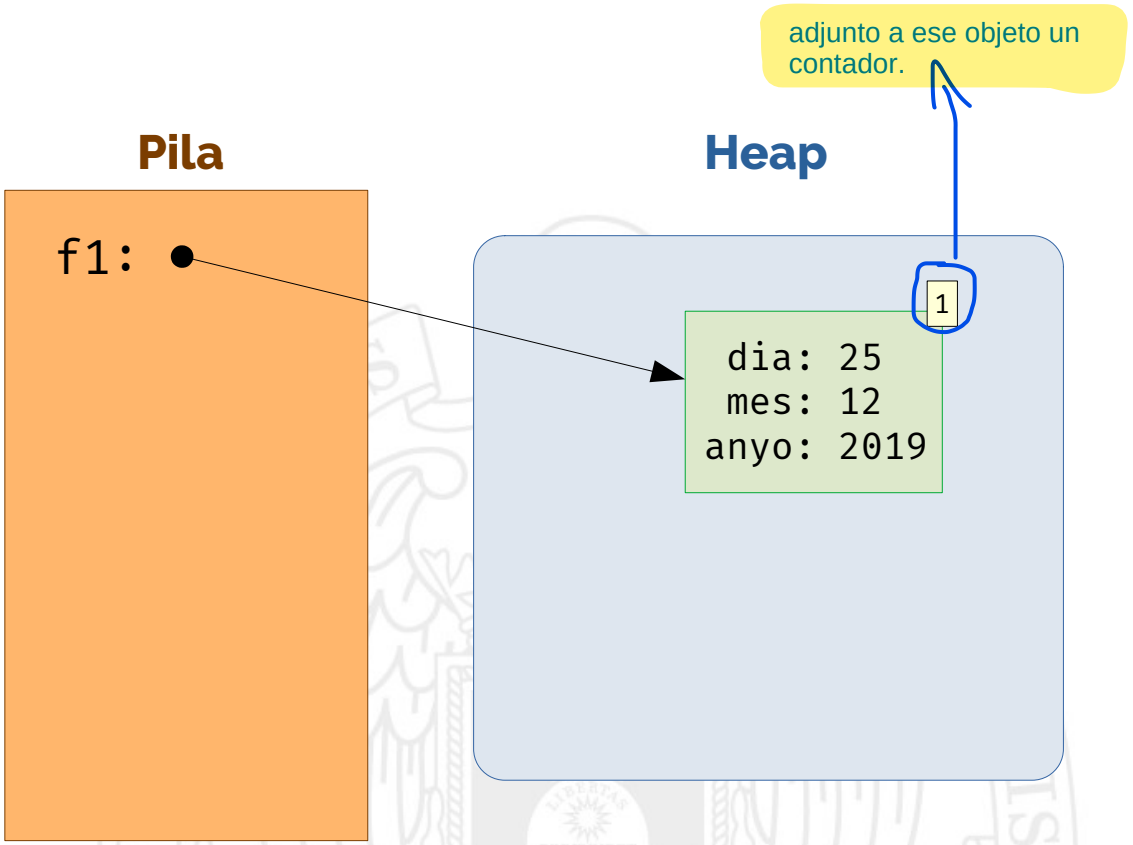
```
std::make_shared<Fecha>(25, 12, 2019)
```

  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

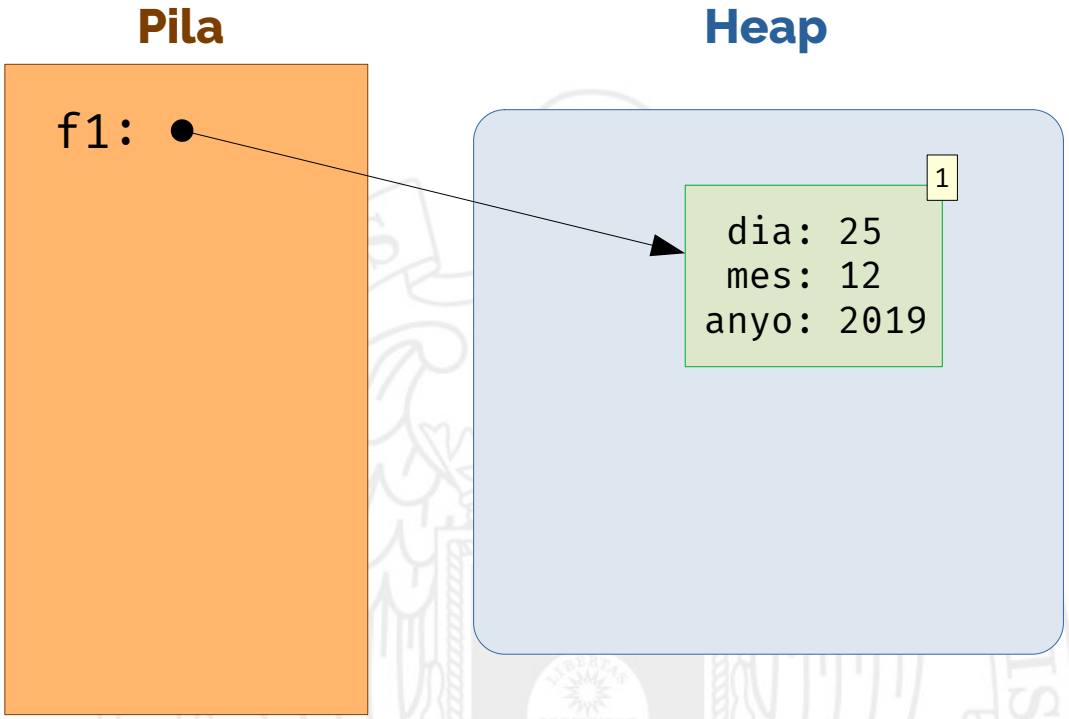
```
std::shared_ptr<Fecha> f1 =
  std::make_shared<Fecha>(25, 12, 2019);
```



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;
}
```

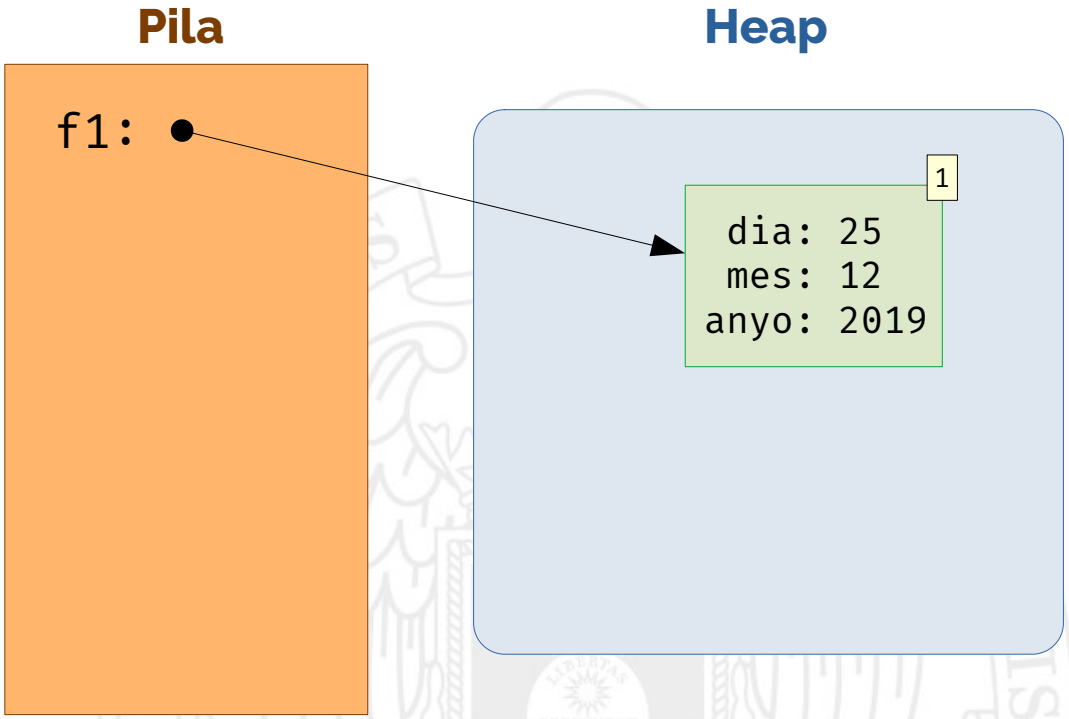
Sobrecargan estos dos operadores los shared pointers.



Copia de un shared_ptr

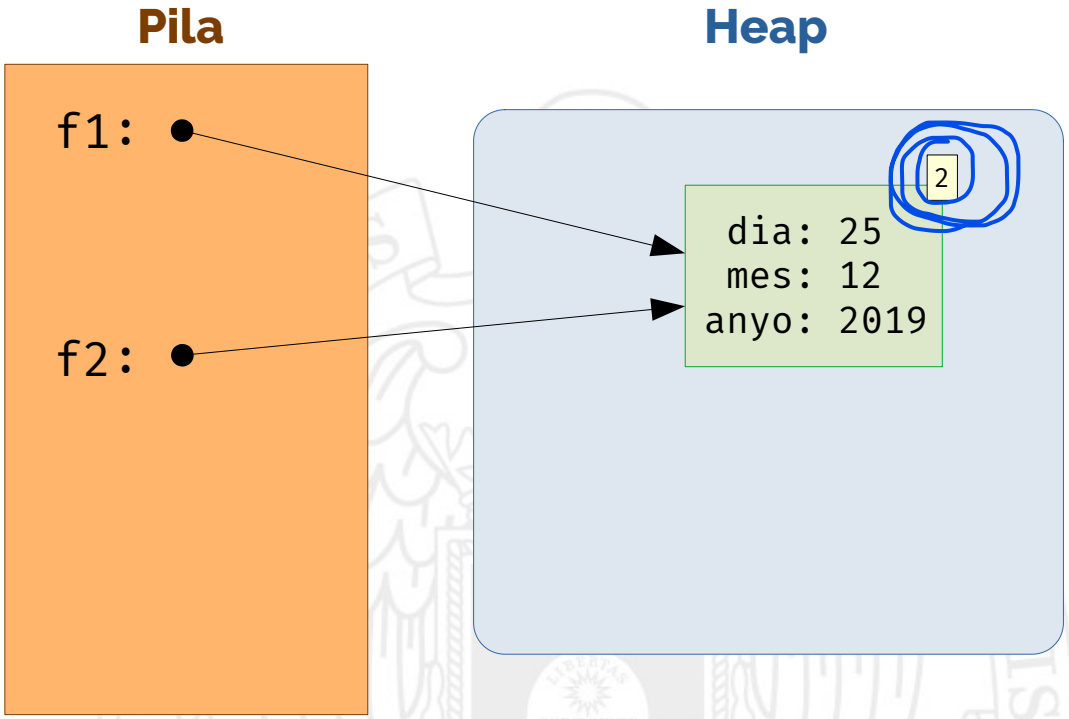
```
std::shared_ptr<Fecha> f1 =
  std::make_shared<Fecha>(25, 12, 2019);

std::shared_ptr<Fecha> f2 = f1;
```

Ahora si que podemos hacerlo, y el constructor de copia hace que aumente el contador en 1.



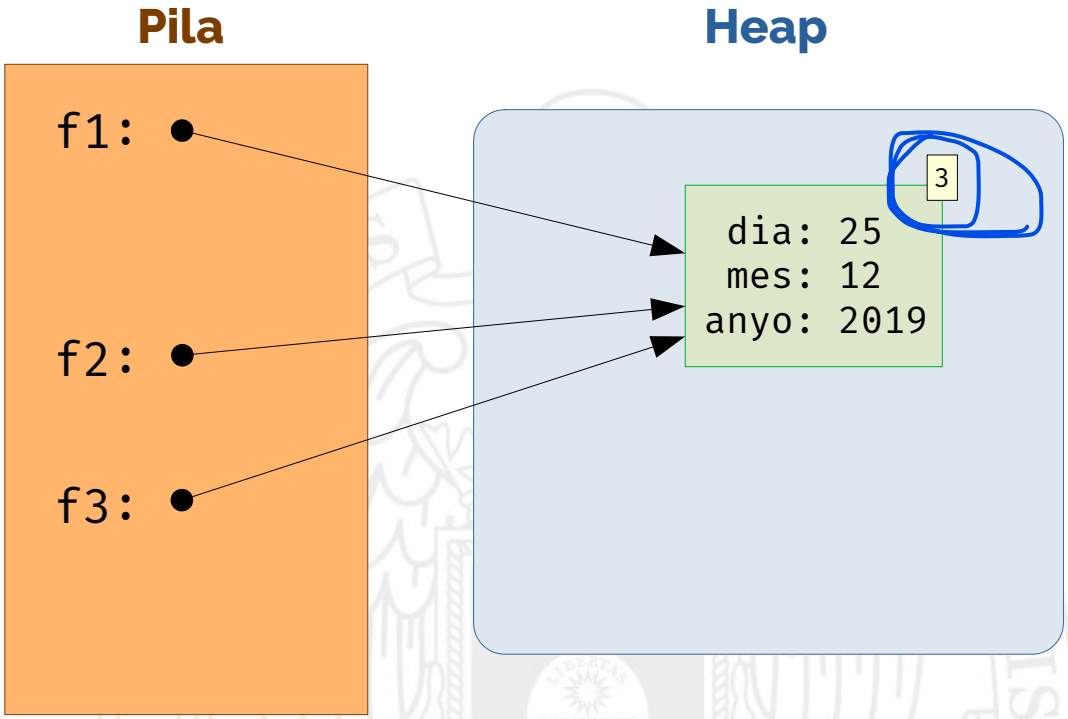
Copia de un shared_ptr

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

std::shared_ptr<Fecha> f2 = f1;
```



Copia de un shared_ptr

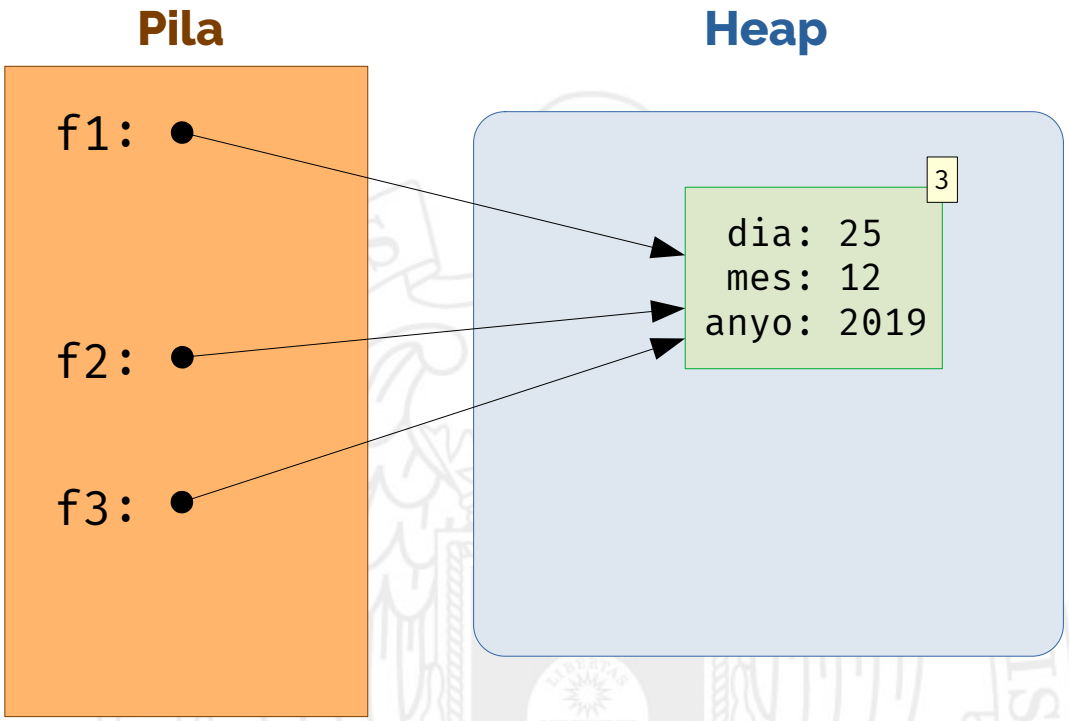
```
std::shared_ptr<Fecha> f1 =
  std::make_shared<Fecha>(25, 12, 2019);

std::shared_ptr<Fecha> f2 = f1;
std::shared_ptr<Fecha> f3 = f2;
```



Copia de un shared_ptr

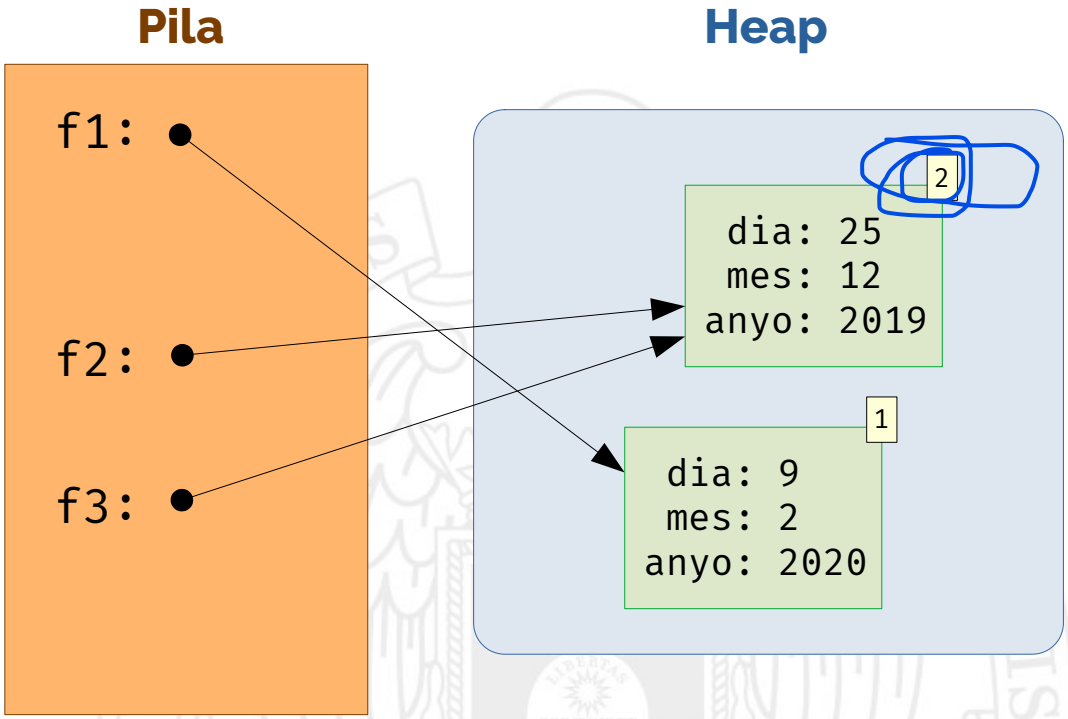
```
std::shared_ptr<Fecha> f1 =
  std::make_shared<Fecha>(25, 12, 2019);

std::shared_ptr<Fecha> f2 = f1;
std::shared_ptr<Fecha> f3 = f2;

f1 = std::make_shared<Fecha>(9, 2, 2020);
```

hacemos que f1 apunte a otro objeto fecha. No se borraría el objeto al que apunte, pero disminuiría el contador en 1. Y se crearía en el heap otro objeto fecha con el contador en 1 y f1 lo apuntaría.



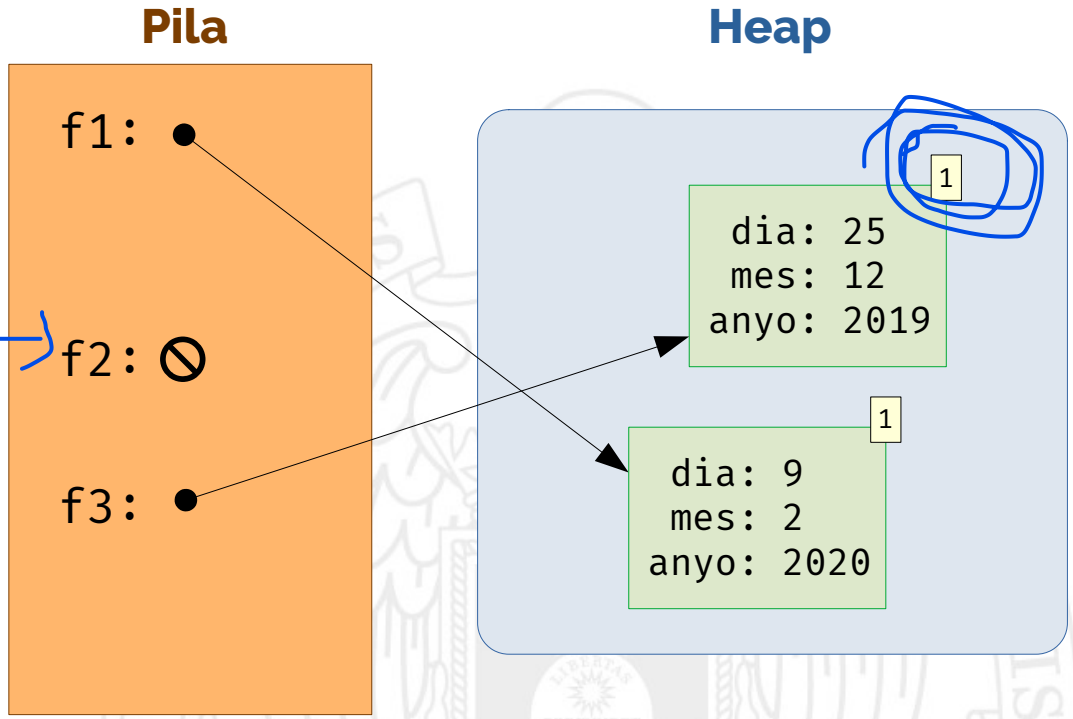
Copia de un shared_ptr

```
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);
```



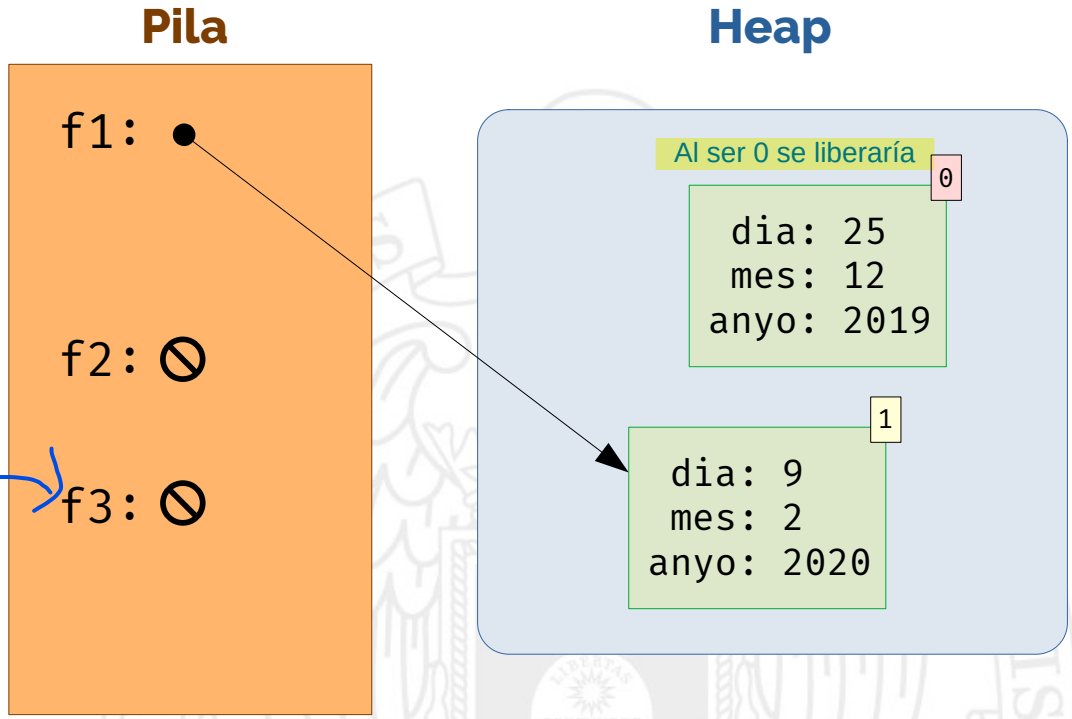
Copia de un shared_ptr

```
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;
```



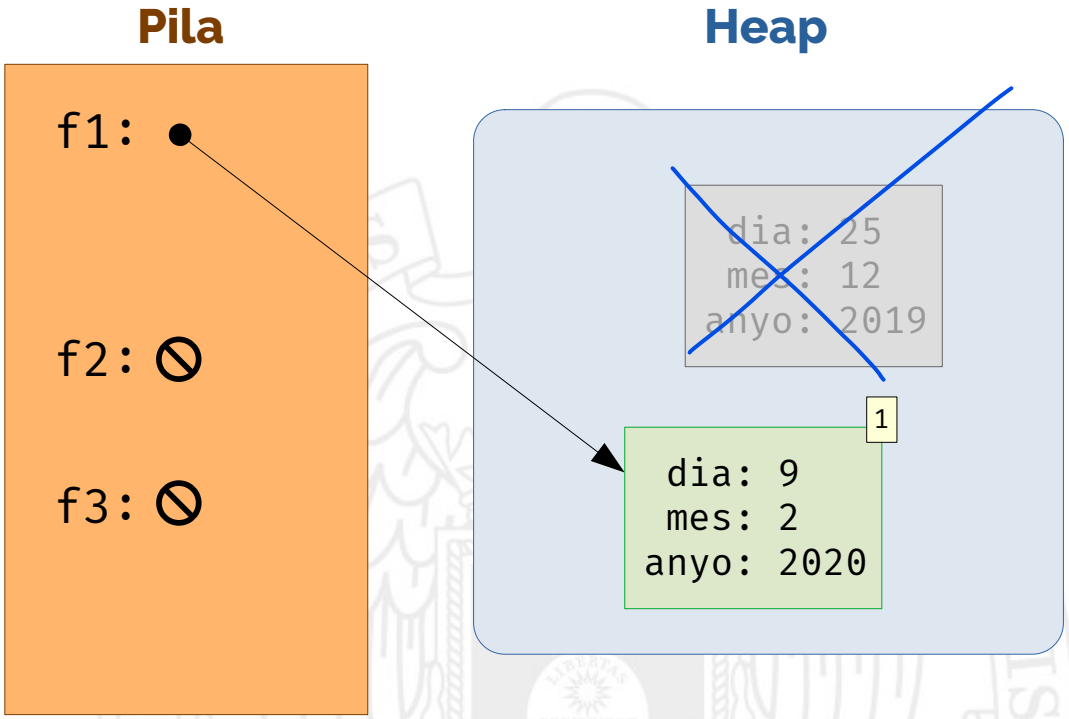
Copia de un shared_ptr

```
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;
```



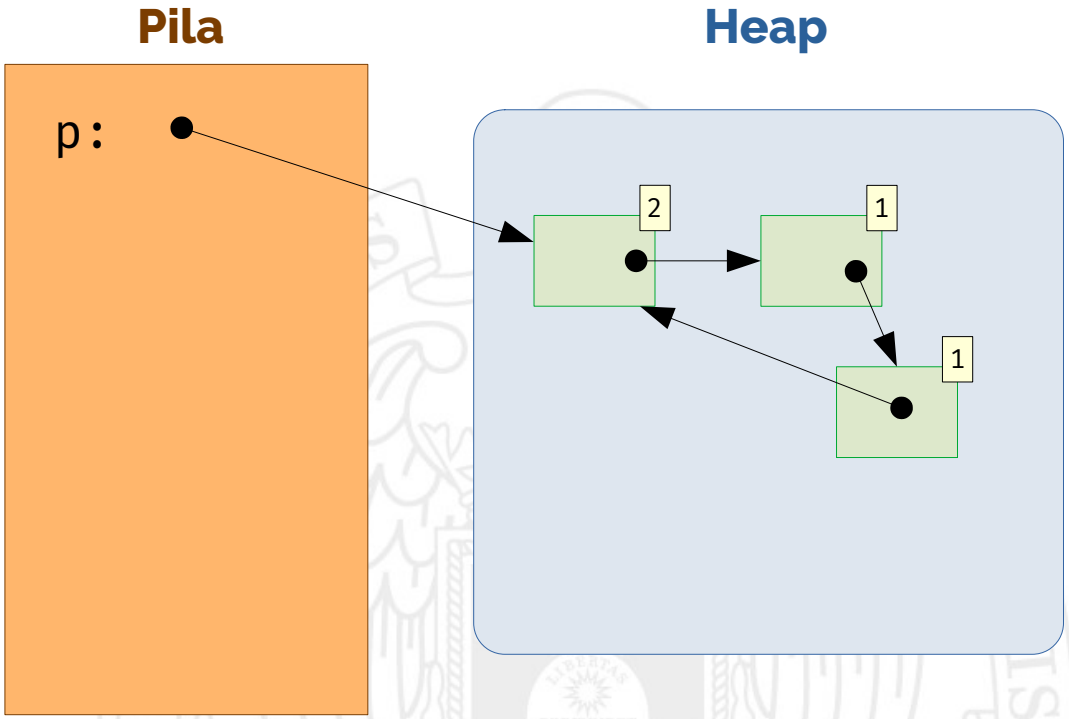
Copia de un shared_ptr

```
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;
```



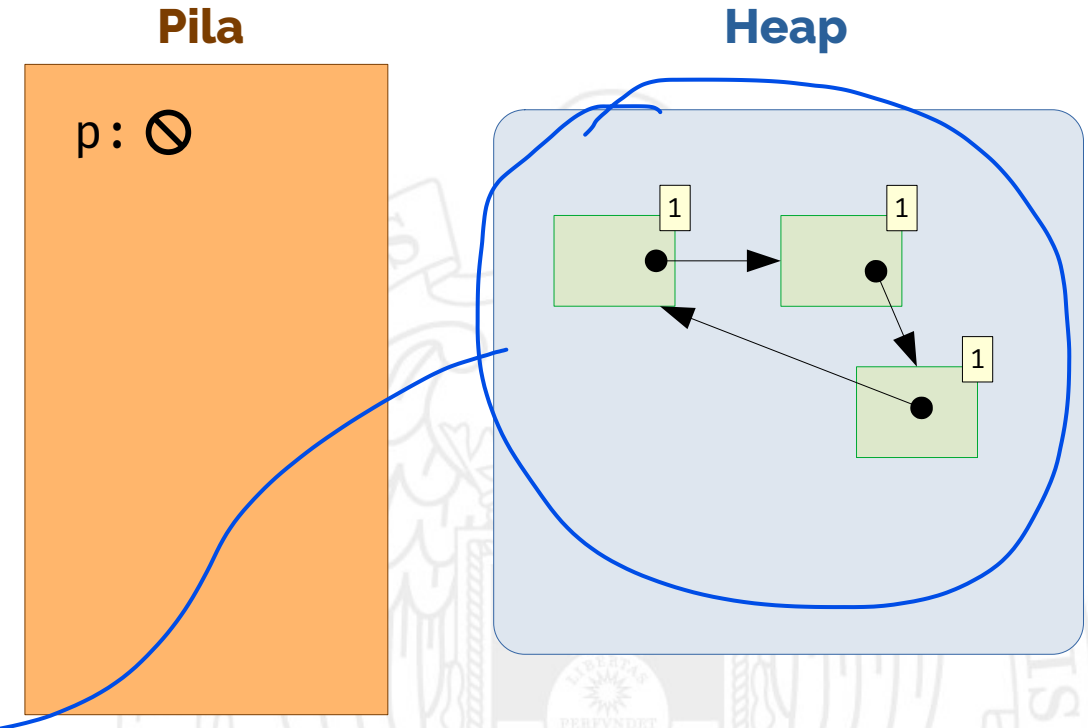
¡Cuidado con las referencias circulares!

No trata muy bien los casos en los que tenemos cadenas circulares de punteros.



¡Cuidado con las referencias circulares!

```
p = nullptr;
```



A pesar de que no le apunta nadie de la pila no se borran porque se apuntan entre ellos.