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

Vamos a trabajar muhco con la memoria del ordenador.

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.

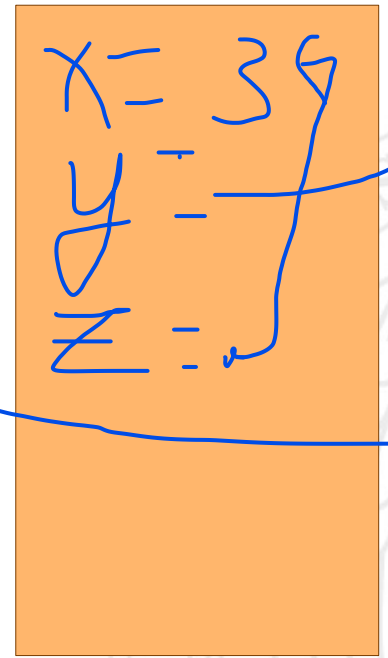
Aquí no hacemos nada

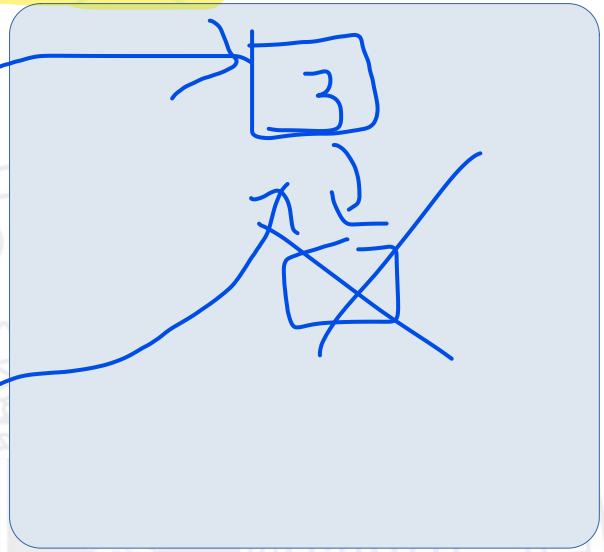
Heap o el montón, de manera manual, y se accede con punteros, y se libera mediante delete. Si no errores de run-error-

Regiones de memoria

```
int main() {
  int x = 3;
  int *y = new int;
  *y = 3;
  int *z = &x;

  delete y;
  return 0;
}
```

variable local





Liberamos todo aquello a lo que hayamos hecho un new.

Asignamos el valor 3 a lo que apunta la y

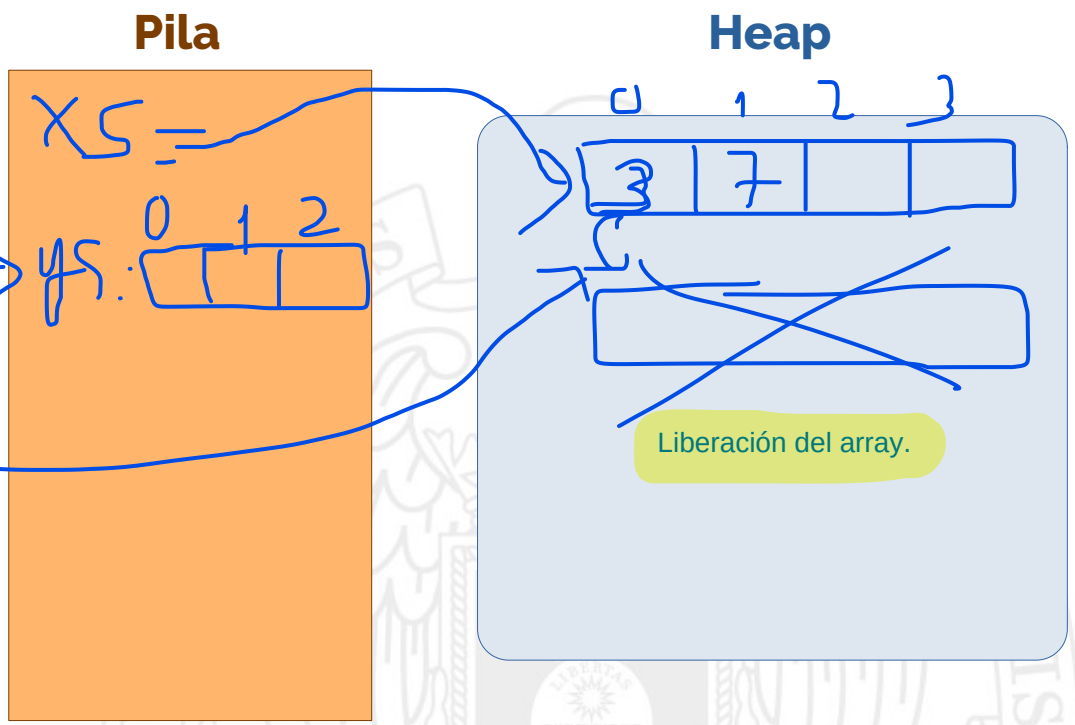
Z ES UN PUNTERO QUE APUNTA A LA DIRECCIÓN DE X

Podemos tener por tanto punteros tanto al heap como a la pila.

Regiones de memoria

```
int main() {
  int *xs = new int[4];
  xs[0] = 3;
  xs[1] = 7;

  int ys[3];

  delete[] xs;
  return 0;
}
```

array que se almacena en el heap



Lo que hayamos creado en el heap lo tenemos que eliminar, en este caso al ser un array hacemos uso de los corchetes.

Creación de objetos en el *heap*

Como se aplica esto a los objetos, y clases.

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;
};
```

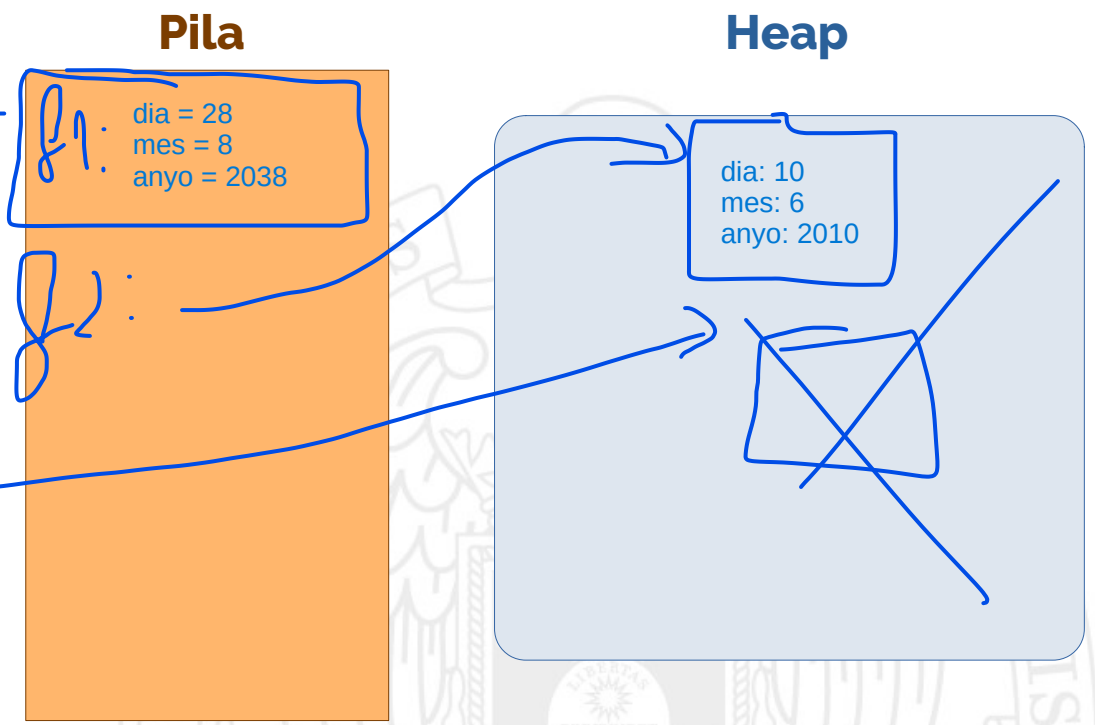
Constructores de la clase.

Métodos de modificación y acceso (const).

atributos privados

Creación de instancias en la pila y *heap*

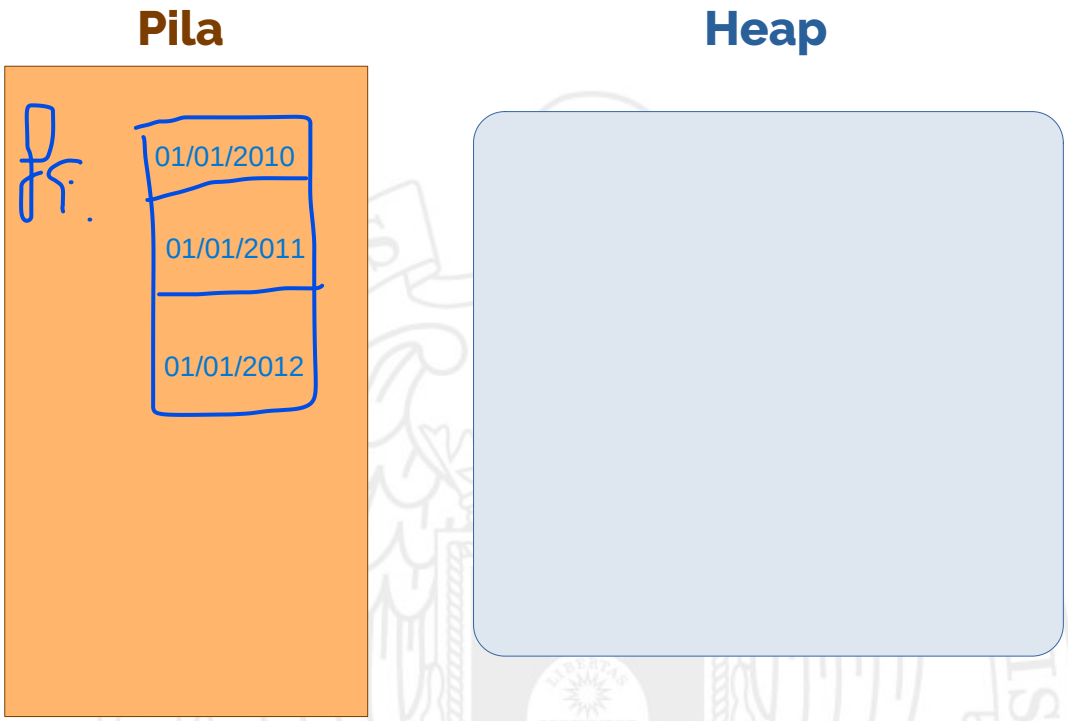
```
int main() {
  Fecha f1(28, 8, 2038);
  Fecha *f2 = new Fecha(10, 6, 2010);
      puntero a fecha, y utilizar el new para heap
  std::cout << "Fecha 1: ";
  f1.imprimir();   28/08/2038
  std::cout << std::endl;

  std::cout << "Fecha 2: ";
  f2->imprimir();   10/06/2010
  std::cout << std::endl;

  delete f2;
  return 0;
}
```



Arrays de objetos

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

  return 0;
}
```

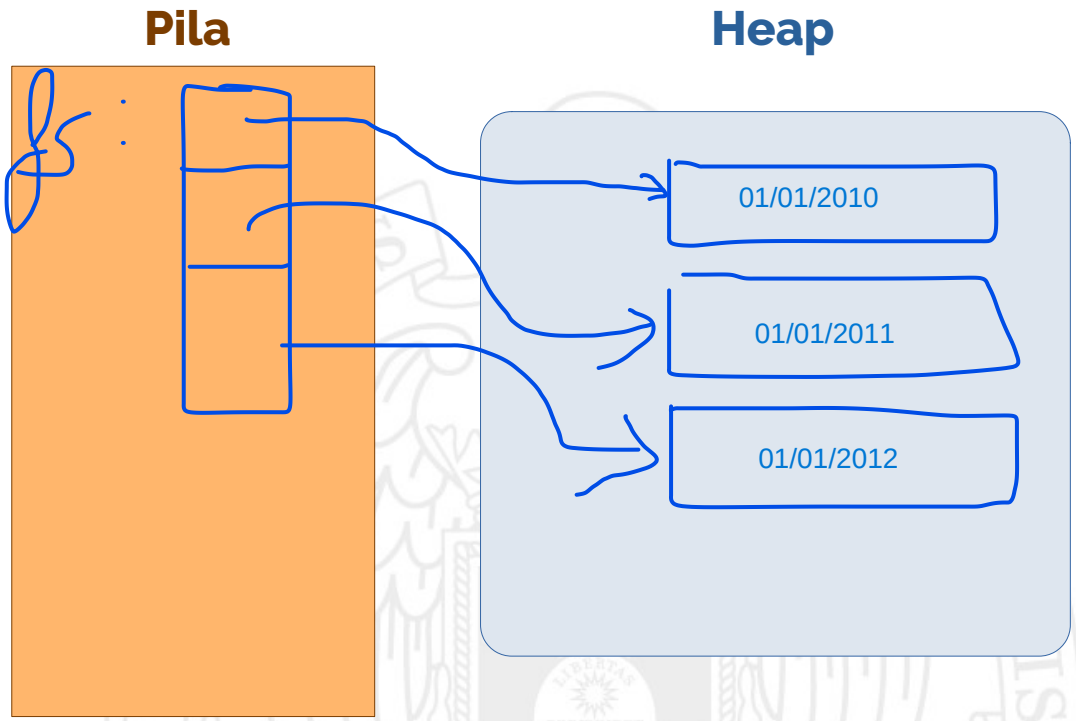
array de 3 elementos en la pila



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;
}
```

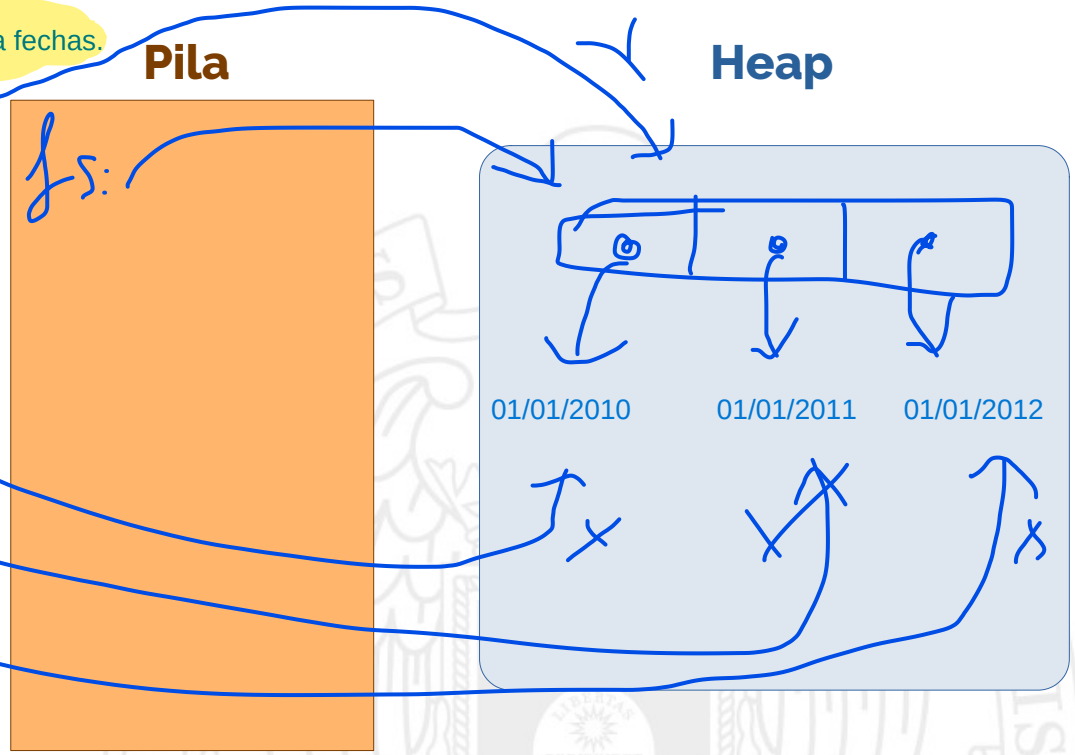
Los tenemos que liberar manualmente todos.



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;
}
```

Los componentes del array son punteros a fechas.

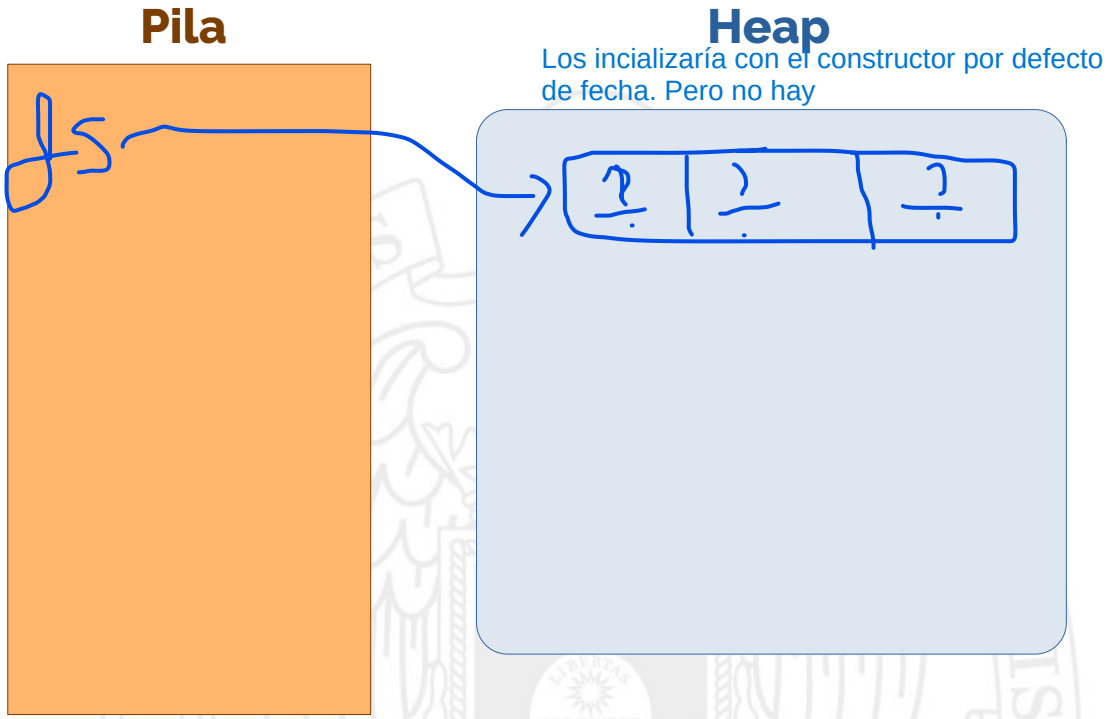


Hemos de borrar manualmente los datos y el array de punteros.

Arrays dinámicos de objetos

```
int main() {
  Fecha *fs = new Fecha[3];

  delete[] fs;
  return 0;
}
```

Daría un error



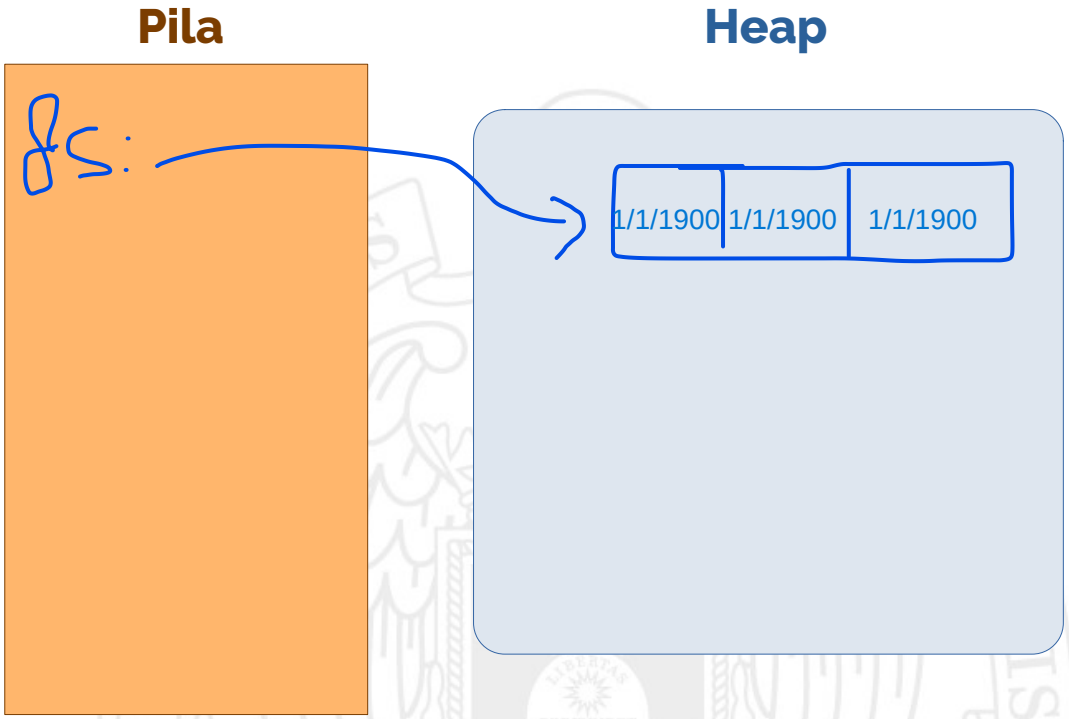
Añadiendo un constructor por defecto

```
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;
};
```

El error anterior se solucionaría creando un constructor por defecto en la clase fecha.

Arrays dinámicos de objetos

```
int main() {
  Fecha *fs = new Fecha[3];

  delete[] fs;
  return 0;
}
```



Compartición de objetos

Cuando se trabaja con punteros

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;
}
```

Apuntarían al mismo objeto

28/08/2019

28/08/2019

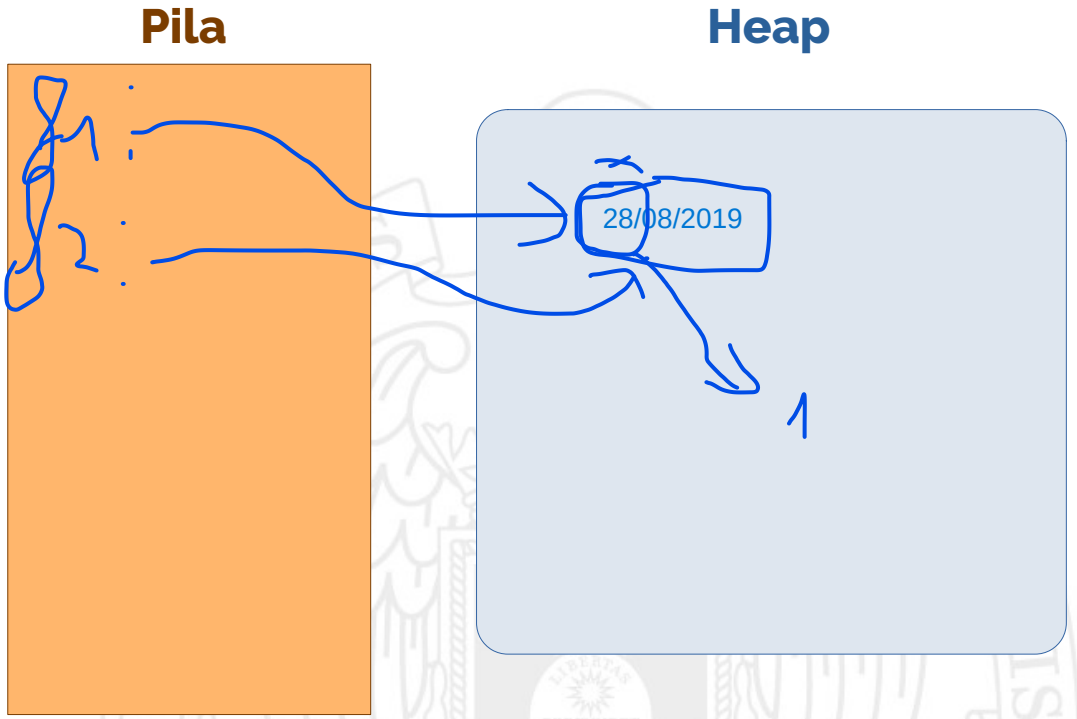
Asigno un 1 al valor de dia

01/08/2019

01/08/2019

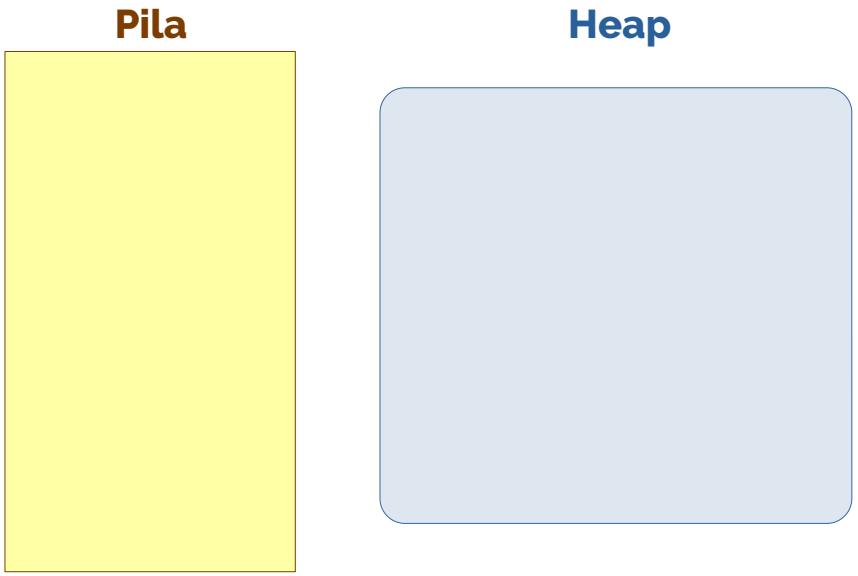
Ya que apuntan a lo mismo.

Hacer o uno u otro.

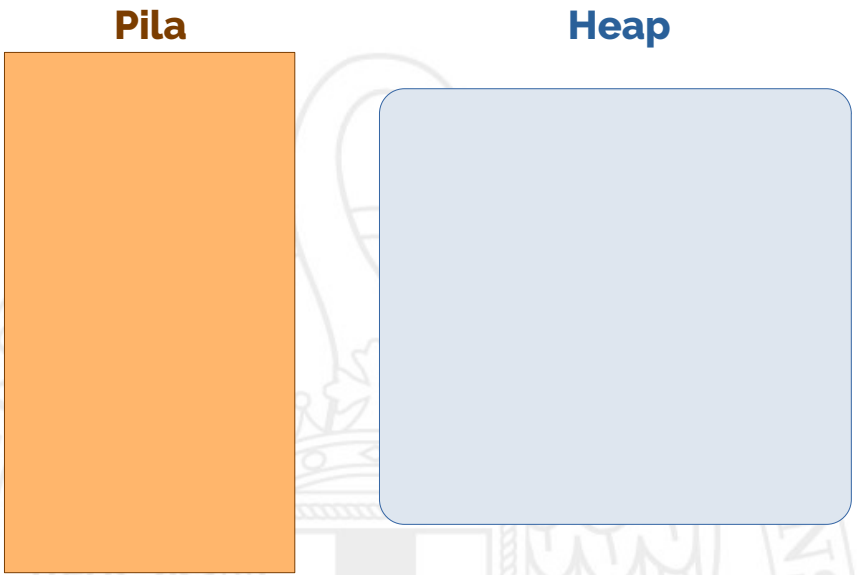


Comparación con Java

Java vs C++



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



- **Los objetos pueden almacenarse en el heap o en la pila.**

Al contrario que en Java.

Pero tendremos que eliminarlo a través del delete, y crearlo en new

Manejo de objetos en Java y C++

JAVA

```java
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.

No hay otro remedio

C++

```cpp
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++

```java
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.

```cpp
int main() {
  Fecha f(20, 3, 2010);
  f.imprimir();

  return 0;
}
```

En C++ es más sencillo crear el objeto `f` en la pila.

Vamos, que a veces es innecesario hacer uso del heap en c++ o al menos eso dice. Que es más sencillo usar 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.

Arrays que crecen a medida que vamos añadiendo elementos a ello
O para datos recursivos como listas enlazadas, árboles. Veremos ejemplos de esos dos uso del heap