FACULTAD DE INFORMÁTICA

Más sobre clases y objetos

SOLUCIONES DEL TALLER

Programación orientada a objetos — Unidad 2

Autor: Luis Hernández Yáñez

FdI UCM

Cuestiones

¿Por qué? ¿Cómo? ¿Cuándo? (*resolución individual*)

- ✓ ¿Por qué no es correcto intentar inicializar los atributos en el mismo momento en el que se declaran?
  Es decir, ¿por qué no es correcto algo como lo siguiente?

```
class Punto {
  ...
private:
  double _x = 0, _y = 0;
};
```

  - Por que se intentaría llevar a cabo esa inicialización sólo una vez, al encontrar ese código durante la compilación. Y eso no sería correcto por dos razones: todavía no existiría ningún objeto cuyos atributos se pudieran inicializar (se lee la clase antes que cualquier declaración de objeto) y no se volvería a leer ese código, por lo que no se ejecutaría sobre ningún objeto creado posteriormente.

FdI UCM

Cuestiones

- ✓ ¿Por qué es más adecuado colocar el código de inicialización en un constructor que en una función miembro como **`inicializa()`**?
  - Por que podemos estar absolutamente seguros de que el código de inicialización se va a ejecutar siempre sobre cualquier objeto que se cree de esa clase. No hay forma de evitarlo.
- ✓ Explica la importancia de los métodos que la Forma canónica ortodoxa señala como esenciales.
  - Son los métodos que intervienen en los mecanismos básicos de la vida de cualquier clase de objeto: su creación (constructor), su destrucción (destructor) y su mimetización (copia con el operador de asignación). El constructor de copia se corresponde con el mecanismo de clonación (creación de una copia).

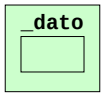
FdI UCM

En construcción

¿Qué aparecerá en la pantalla? (*resolución individual*)

Indica qué mostrará en la pantalla el programa que sigue (sin ejecutarlo):

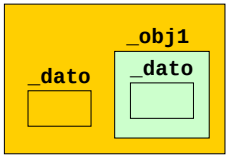
```
#include <iostream>
using namespace std;

class Una {
public:
  Una(int d = 1) : _dato(d) { }
  void mostrar() const { cout << _dato; }
private:
  int _dato;
};
```



En construcción

```
class Dos {
public:
  Dos(int d = 2) : _dato(d), _obj1(d) { }
  void mostrar() const { cout << "(" << _dato << ",";
    _obj1.mostrar(); cout << ")"; }
private:
  int _dato;
  Una _obj1;
};
```

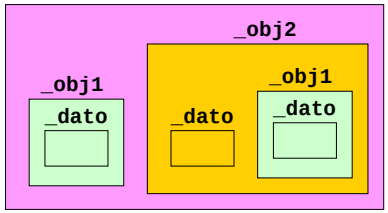
El constructor de **Una** se ejecutará con el valor que se pase (2 por defecto)



En construcción

```
class Tres {
public:
  Tres(int d = 3) : _obj1(d) { }
  void mostrar() const { cout << "("; _obj1.mostrar();
    cout << ","; _obj2.mostrar(); cout << ")"; }
private:
  Una _obj1;
  Dos _obj2;
};
```

El constructor de **Una** se ejecutará con el valor que se pase (3 por defecto)



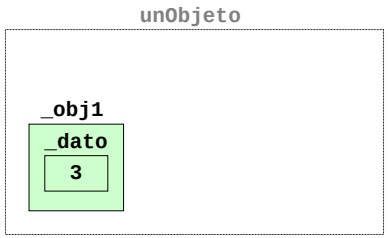
En construcción

```
int main() {
  Tres unObjeto;
```

Se han de crear primero los atributos de **unObjeto** (**_obj1** y **_obj2**)

_obj1 es de clase **Una**. El inicializador del constructor de la clase **Tres** (**_obj1(d)**) hace que se pase el argumento implícito 3 al constructor de la clase **Una** para que lo use al construir el atributo.

```
Una(int d) : _dato(d) { }
```



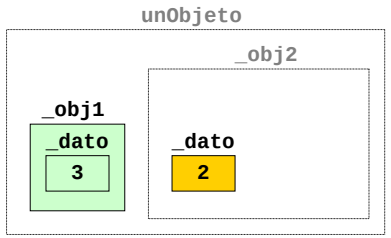
En construcción

```
Tres unObjeto;
```

Se han de crear primero los atributos de **unObjeto** (**_obj1** y **_obj2**)

_obj2 es de clase **Dos**. Desde la clase **Tres** se construye ese atributo usando su constructor predeterminado.

```
Dos(int d = 2) : _dato(d), _obj1(d) ...
```



_dato se inicializa a 2

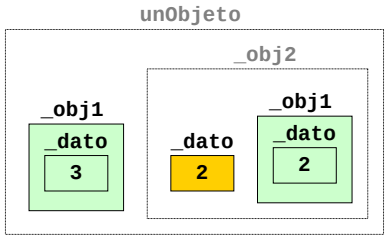
En construcción

```
Tres unObjeto;
```

Se han de crear primero los atributos de **unObjeto** (**_obj1** y **_obj2**)

_obj2 es de clase **Dos**. Desde la clase **Tres** se construye ese atributo usando su constructor predeterminado.

```
Dos(int d = 2) : _dato(d), _obj1(d) ...
```



Al constructor de **_obj1** se le pasa el valor 2

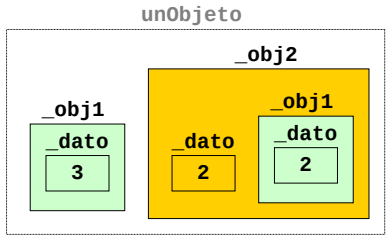
En construcción

```
Tres unObjeto;
```

Se han de crear primero los atributos de **unObjeto** (**_obj1** y **_obj2**)

_obj2 es de clase **Dos**. Desde la clase **Tres** se construye ese atributo usando su constructor predeterminado.

```
Dos(int d = 2) : _dato(d), _obj1(d) { }
```

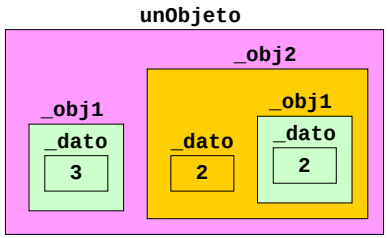


Se ejecuta el cuerpo del constructor de **Dos**, que no hace nada adicional

En construcción

```
Tres unObjeto;
```

```
Tres(int d = 3) : _obj1(d) { }
```



Se ejecuta el cuerpo del constructor de **Tres**, que no hace nada adicional

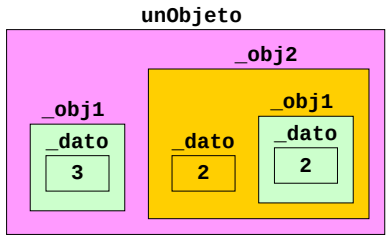
En construcción

```
unObjeto.mostrar(); cout << endl;
```

```
void mostrar() const { cout << "("; _obj1.mostrar();
  cout << ","; _obj2.mostrar(); cout << ")"; }
```

```
void mostrar() const {
  cout << "(" << _dato << ",";
  _obj1.mostrar(); cout << ")"; }
```

```
void mostrar() const {
  cout << _dato; }
```



Se muestra **(3,(2,2))**

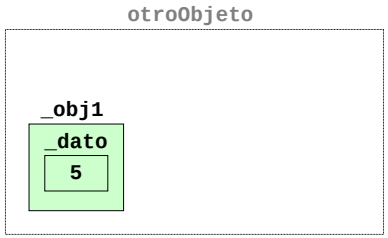
En construcción

```
Tres otroObjeto(5);
```

Se han de crear primero los atributos de **otroObjeto** (**_obj1** y **_obj2**)

_obj1 es de clase **Una**. El inicializador del constructor de la clase **Tres** (**_obj1(d)**) hace que se pase el valor 5 al constructor de la clase **Una** para que lo use al construir el atributo.

```
Una(int d) : _dato(d) { }
```



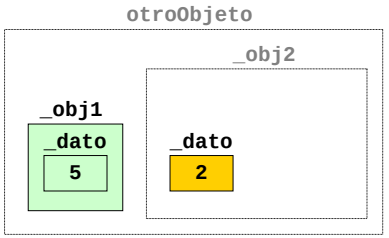
En construcción

```
Tres otroObjeto(5);
```

Se han de crear primero los atributos de **otroObjeto** (**_obj1** y **_obj2**)

_obj2 es de clase **Dos**. Desde la clase **Tres** se construye ese atributo usando su constructor predeterminado.

```
Dos(int d = 2) : _dato(d), _obj1(d) ...
```



_dato se inicializa a 2

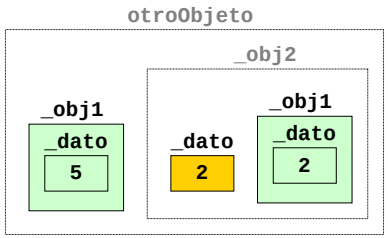
En construcción

```
Tres otroObjeto(5);
```

Se han de crear primero los atributos de **otroObjeto** (**_obj1** y **_obj2**)

_obj2 es de clase **Dos**. Desde la clase **Tres** se construye ese atributo usando su constructor predeterminado.

```
Dos(int d = 2) : _dato(d), _obj1(d) ...
```



Al constructor de **_obj1** se le pasa el valor 2

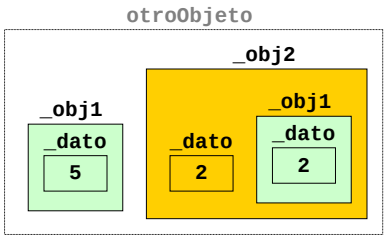
En construcción

```
Tres otroObjeto(5);
```

Se han de crear primero los atributos de **otroObjeto** (**_obj1** y **_obj2**)

_obj2 es de clase **Dos**. Desde la clase **Tres** se construye ese atributo usando su constructor predeterminado.

```
Dos(int d = 2) : _dato(d), _obj1(d) { }
```

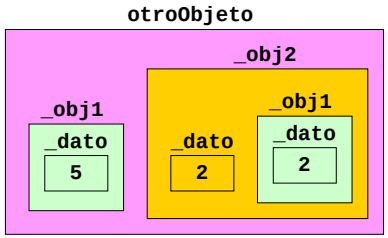


Se ejecuta el cuerpo del constructor de **Dos**, que no hace nada adicional

En construcción

```
Tres otroObjeto(5);
```

```
Tres(int d) : _obj1(d) { }
```



Se ejecuta el cuerpo del constructor de **Tres**, que no hace nada adicional

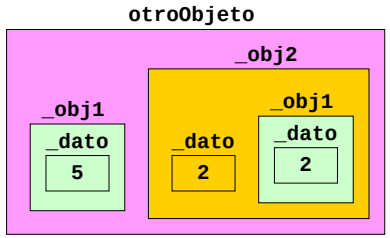
En construcción

```
otroObjeto.mostrar(); cout << endl;
```

```
void mostrar() const { cout << "("; _obj1.mostrar();
  cout << ","; _obj2.mostrar(); cout << ")"; }
```

```
void mostrar() const {
  cout << "(" << _dato << ",";
  _obj1.mostrar(); cout << ")"; }
```

```
void mostrar() const {
  cout << _dato; }
```



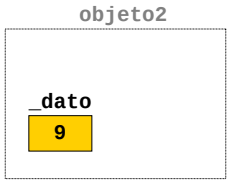
Se muestra **(5,(2,2))**

En construcción

```
Dos objeto2(9);
```

objeto2 es de clase **Dos**. Se construye el objeto usando el valor que se pasa (9) en su constructor.

```
Dos(int d) : _dato(d), _obj1(d) ...
```

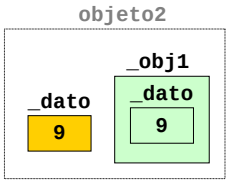


_dato se inicializa a 9

En construcción

```
Dos objeto2(9);
```

```
Dos(int d) : _dato(d), _obj1(d) ...
```

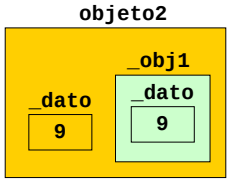


Al constructor de **_obj1** se le pasa el valor 9

En construcción

```
Dos objeto2(9);
```

```
Dos(int d) : _dato(d), _obj1(d) { }
```



Se ejecuta el cuerpo del constructor de **Dos**, que no hace nada adicional

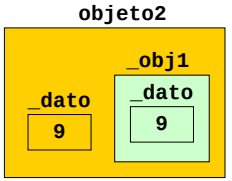
En construcción

```
objeto2.mostrar(); cout << endl;
```

```
void mostrar() const { cout << "(" << _dato << ",";
  _obj1.mostrar(); cout << ")"; }
```

```
void mostrar() const {
  cout << _dato; }
```

```
  return 0;
}
```



Se muestra **(9,9)**

Ejercicio práctico

Una clase más completa (*resolución en grupo*)

A partir de la clase **Contador2** del taller de la semana anterior, hay que crear otra clase, **Contador3**, que sea igual que la anterior y que se ajuste a la Forma canónica ortodoxa. Además, se han de declarar como constantes los métodos para los que resulte procedente. Y se ha de probar exhaustivamente la nueva clase en una función **main()**.

Ejercicio práctico

```
#include <iostream>
using namespace std;

class Contador3 {
public:
  Contador3(int = 0); // Constructor (predeterminado)
  Contador3(const Contador3&); // Constructor de copia
  Contador3& operator=(const Contador3&); // Copia (asignación)
  ~Contador3(); // Destructor
  void incrementar();
  void decrementar ();
  void mostrar() const;
private:
  int _cont;
};

Contador3::Contador3(int val){ _cont = (val > 0) ? val : 0; }

Contador3::Contador3(const Contador3& otro) {
  _cont = otro._cont;
}
```

continúa

Ejercicio práctico

```
Contador3& Contador3::operator=(const Contador3& otro) {
  _cont = otro._cont;
  return *this;
}

Contador3::~Contador3() { }

void Contador3::incrementar(){
  _cont++;
}

void Contador3::decrementar () {
  if(_cont>0) _cont--;
}

void Contador3::mostrar() const {
  cout << _cont << endl;
}
```

const tanto en el prototipo como en la implementación

continúa

Ejercicio práctico

```
int main() {
  Contador3 c1; // Constructor predeterminado
  c1.mostrar();
  Contador3 c2(5);
  c2.mostrar();
  Contador3 c3(c2); // Construcción por copia
  c3.mostrar();
  Contador3 c4;
  c4 = c3; // Copia
  c4.mostrar();

  return 0;
}
```

>>> cont3.cpp

En el mundo real

Mejorando la clase (*resolución en grupo*)

Se ha de modificar la clase **Empleado** del taller de la semana anterior, de forma que se ajuste a la Forma canónica ortodoxa. Además, se han de declarar como constantes los métodos para los que resulte procedente. Y se ha de probar exhaustivamente la clase en una función **main()**.

En el mundo real

```
// Clase Empleado - Archivo de cabecera "empleado.h"

#ifndef empleado_h // Evitar inclusiones múltiples
#define empleado_h

class Empleado {
public:
  Empleado(long int = 0, bool = false, int = 0, double = 0,
           double = 0, double = 0, double = 0);
  // Constructor (predeterminado)
  // Datos: DNI, ¿casado?, hijos, sueldo base, tipo IRPF,
  // pago por hora extra, horas extra realizadas.
  Empleado(const Empleado&); // Constructor de copia
  Empleado& operator=(const Empleado&); // Copia (asignación)
  ~Empleado(); // Destructor
```

continúa

En el mundo real

```
// Accedentes y mutadores:
 long int dni() const;
 bool casado() const;
 void casado(bool);
 int hijos() const;
 void hijos(int);
 double sueldoBase() const;
 void sueldoBase(double);
 double tipo() const;
 void tipo(double);
 double pagoHoraExtra() const;
 void pagoHoraExtra(double);
 double horasExtra() const;
 void horasExtra(double);
 // Métodos computadores:
 double complemento() const;
 double sueldoBruto() const;
 double retenciones() const;
```

Los accedentes siempre son métodos **const**

continúa

En el mundo real

```
  // Métodos visualizadores:
  void mostrar() const;
  void mostrarTodo() const;
private:
  long int _dni;
  bool _casado;
  int _hijos;
  double _sueldoBase;
  double _tipo; // porcentaje de IRPF para impuestos
  double _pagoHoraExtra;
  double _horasExtra;
};

#endif
```

Los métodos visualizadores suelen ser métodos **const**

En el mundo real

```
// Clase Empleado - Archivo de implementación "empleado.cpp"

#include <iostream>
using namespace std;
#include "empleado.h"

Empleado::Empleado(long int dni, bool casado, int hijos,
  double sueldo, double tipo, double pago, double extras)
  : _dni(dni), _casado(casado), _hijos(hijos),
    _sueldoBase(sueldo), _tipo(tipo), _pagoHoraExtra(pago),
    _horasExtra(extras) { }

Empleado::Empleado(const Empleado& otro) {
  _dni = otro._dni;
  _casado = otro._casado;
  _hijos = otro._hijos;
  _sueldoBase = otro._sueldoBase;
  _tipo = otro._tipo;
  _pagoHoraExtra = otro._pagoHoraExtra;
  _horasExtra = otro._horasExtra;
}
```

continúa

En el mundo real

```
Empleado& Empleado::operator=(const Empleado& otro) {
  _dni = otro._dni;
  _casado = otro._casado;
  _hijos = otro._hijos;
  _sueldoBase = otro._sueldoBase;
  _tipo = otro._tipo;
  _pagoHoraExtra = otro._pagoHoraExtra;
  _horasExtra = otro._horasExtra;
  return *this;
}

Empleado::~Empleado() { }

long int Empleado::dni() const { return _dni; }

bool Empleado::casado() const { return _casado; }

void Empleado::casado(bool c) { _casado = c; }
```

continúa

En el mundo real

```
int Empleado::hijos() const { return _hijos; }

void Empleado::hijos(int h) { _hijos = h; }

double Empleado::sueldoBase() const { return _sueldoBase; }

void Empleado::sueldoBase(double sb) { _sueldoBase = sb; }

double Empleado::tipo() const { return _tipo; }

void Empleado::tipo(double t) { _tipo = t; }

double Empleado::pagoHoraExtra() const { return _pagoHoraExtra; }

void Empleado::pagoHoraExtra(double phe) {
  _pagoHoraExtra = phe; }

double Empleado::horasExtra() const { return _horasExtra; }

void Empleado::horasExtra(double he) { _horasExtra = he; }
```

continúa

En el mundo real

```
double Empleado::complemento() const {
  return _pagoHoraExtra * _horasExtra;
}

double Empleado::sueldoBruto() const {
  return _sueldoBase + complemento();
}

double Empleado::retenciones() const {
// El tipo siempre es suficientemente alto, por lo que
// nunca se aplicará un tipo final negativo
  double tipoFinal = _tipo;
  if(_casado) tipoFinal -= 2;
  tipoFinal -= _hijos;
  return sueldoBruto() * tipoFinal / 100;
}
```

continúa

En el mundo real

```
void Empleado::mostrar() const {
  cout << endl;
  cout << "D.N.I.: " << _dni << endl;
  cout << "Casado: " << (_casado ? "Si" : "No") << endl;
  cout << "Hijos: " << _hijos << endl;
  cout << "Sueldo base: " << _sueldoBase << endl;
  cout << "Porcentaje IRPF: " << _tipo << endl;
  cout << "Pago por hora extra: " << _pagoHoraExtra << endl;
  cout << "Horas extra realizadas: " << _horasExtra << endl;
}

void Empleado::mostrarTodo() const {
  mostrar(); // muestra la información básica
  cout << "Complemento horas extra: " << complemento() << endl;
  double sueldo = sueldoBruto();
  cout << "Sueldo bruto: " << sueldo << endl;
  double ret = retenciones();
  cout << "Retenciones I.R.P.F.: " << ret << endl;
  sueldo -= ret;
  cout << "Sueldo neto: " << sueldo << endl;
}
```


En el mundo real

```
// El programa de prueba - emplea.cpp

#include "empleado.h"

int main()
{
  Empleado emp(333666, true, 1, 1680, 18, 45, 3.5);
  emp.mostrar();
  emp.mostrarTodo();
  Empleado otro(emp); // Construcción por copia
  otro.mostrar();
  otro.hijos(3);
  otro.sueldoBase(2145);
  otro.tipo(21);
  otro.pagoHoraExtra(63);
  otro.horasExtra(1.3);
  otro.mostrarTodo();
  otro = emp; // Copia (asignación)
  otro.mostrarTodo();

  return 0;
}
```