FACULTAD DE INFORMÁTICA

Introducción a la POO
Clases y objetos

SOLUCIONES DEL TALLER

Programación orientada a objetos — Unidad 1

Autor: Luis Hernández Yáñez

FdI UCM

Cuestiones

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

¿Por qué es más correcto implementar todos los métodos fuera de la estructura **class**?

- Porque así los aspectos de implementación permanecen ocultos a los clientes de la clase y la interfaz queda mucho más clara.

¿En qué casos se te ocurre que no sería adecuado incluir en una clase ciertos mutadores? ¿Y ciertos accedentes?

- En aquellos casos en los que la mutación (cambio) del atributo sea responsabilidad de la propia clase, de forma que cada estado del atributo *(su valor)* dependa de su estado anterior, de las operaciones realizadas o de otras condiciones ajenas a los clientes de la clase. Por ejemplo, en una lista el contador de elementos es mantenido por la propia clase, incrementándose con las inserciones de elementos y decrementándose con las eliminaciones; en ningún caso los clientes de la lista deben poder fijar el número de elementos por su cuenta.
- Los accedentes no tienen el peligro de los mutadores, por lo que proporcionarlos o no dependerá de la utilidad que puedan suponer.

FdI UCM

Cuestiones

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

¿En qué casos crees que los pasos de mensajes no se escribirían con la notación punto (*receptor•mensaje*)?

- Cuando las funciones miembro sean funciones operadoras. Se usará la notación infija habitual de los operadores.

¿Por qué en los métodos para los pasos de mensajes al propio objeto receptor no se coloca delante el objeto receptor y el punto?

- Porque no hay forma de saber qué objeto concreto ha recibido el mensaje que desencadenó la ejecución del método. Por ejemplo si tenemos:

```
Cuenta c1, c2;
...
c1.abonoIntereses();
c2.abonoIntereses();
```

Cada paso de mensaje **abonoIntereses()** genera un paso de mensaje **ingreso()** al objeto receptor. En el primer caso el objeto receptor es **c1**, mientras que en el segundo caso se trata de **c2**.

FdI UCM

Objetos dentro de objetos y encadenamiento de mensajes

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

Dadas las clases que se muestran a continuación, averigua qué se mostrará en la pantalla con el programa principal del final.

- Archivo de cabecera **claseA.h**:

```
class ClaseA {
public:
  void init(int = 0);
  int mas(int = 0);
  int por(int = 1);
private:
  int _a;
};
```

- Archivo de implementación **claseA.cpp**:

```
#include "claseA.h"
// Están definidos argumentos implícitos
void ClaseA::init(int val) { _a = val; }
int ClaseA::mas(int val) { return _a + val; }
int ClaseA::por(int val) { return _a * val; }
```

. . . / . . .

Objetos dentro de objetos y encadenamiento de mensajes

- Archivo de cabecera **claseB.h**:

```
#include "claseA.h"
class ClaseB {
public:
  void init(int = 0);
  int mas(int = 0);
  int por(int = 1);
  void igual(ClaseA);
private:
  ClaseA _obj;
};
```

- Archivo de implementación **claseB.cpp**:

```
#include "claseB.h"
// Están definidos argumentos implícitos
void ClaseB::init(int val) { _obj.init(val); }
int ClaseB::mas(int val) { return _obj.mas(val); }
int ClaseB::por(int val) { return _obj.por(val); }
void ClaseB::igual(ClaseA otro) { _obj = otro; }
```

. . . / . . .

Objetos dentro de objetos y encadenamiento de mensajes

- Archivo de cabecera **claseC.h**:

```
#include "claseA.h"
#include "claseB.h"
class ClaseC {
public:
  void init(int = 0, int = 0);
  int mas();
  int por();
  void obj1(ClaseA);
  void obj2(ClaseB);
private:
  ClaseA _obj1;
  ClaseB _obj2;
};
```

- Archivo de implementación **claseC.cpp**:

```
#include "claseC.h"

void ClaseC::init(int val1, int val2) // Argumentos implícitos
{ _obj1.init(val1); _obj2.init(val2); }
```

. . . / . . .

Objetos dentro de objetos y encadenamiento de mensajes

```
int ClaseC::mas() { return _obj1.mas() + _obj2.mas(); }
int ClaseC::por() { return _obj1.por() * _obj2.por(); }
void ClaseC::obj1(ClaseA obj) { _obj1 = obj; }
void ClaseC::obj2(ClaseB obj) { _obj2 = obj; }
```

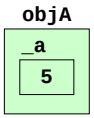
. . . / . . .

Objetos dentro de objetos y encadenamiento de mensajes

```
#include <iostream>
using namespace std;
#include "claseA.h"
#include "claseB.h"
#include "claseC.h"

int main()
{
  ClaseA objA;
  objA.init(5);
```

```
void ClaseA::init(int val) { _a = val; }
```




```
  cout << objA.por(5) << endl; // endl provoca un salto de línea
```

```
int ClaseA::por(int val) { return _a * val; }
```

Aparece en la pantalla un 25.

. . . / . . .

Objetos dentro de objetos y encadenamiento de mensajes

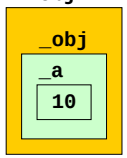
```
ClaseB objB;
objB.init(10);
```

```
void ClaseB::init(int val) { _obj.init(val); }
```

Encadenamiento de mensajes

```
void ClaseA::init(int val)
{ _a = val; }
```

objB




```
cout << objB.por(5) + objA.mas(3) << endl;
```

objA


```
int ClaseB::por(int val)
{ return _obj.por(val); }
```

```
int ClaseA::mas(int val)
{ return _a + val; }
```

Devuelve 8

Encadenamientos de mensajes

```
int ClaseA::por(int val)
{ return _a * val; }
```

Devuelve 50

Aparece en la pantalla 58 (50 + 8)

. . . / . . .

Objetos dentro de objetos y encadenamiento de mensajes

```
ClaseC objC;
objC.init(12, 7);
```

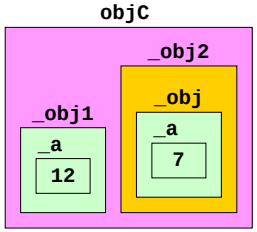
```
void ClaseC::init(int val1, int val2)
{ _obj1.init(val1); _obj2.init(val2); }
```

```
void ClaseA::init(int val)
{ _a = val; }
```

```
void ClaseB::init(int val)
{ _obj.init(val); }
```

```
void ClaseA::init(int val)
{ _a = val; }
```

objC



Encadenamientos de mensajes

. . . / . . .

Objetos dentro de objetos y encadenamiento de mensajes

```
cout << objC.mas() << endl;
```

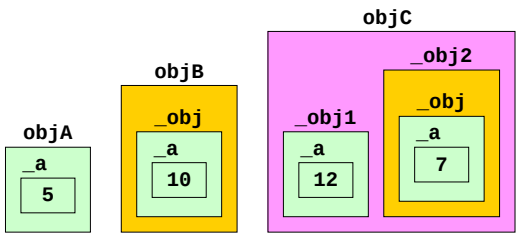
Aparece en la pantalla 19 (12 +7)

```
ClaseC::mas() --> ClaseA::mas()
              --> ClaseB::mas() --> ClaseA::mas()
```

```
cout << objC.por() << endl;
```

Aparece en la pantalla 84 (12 x 7)

```
ClaseC::por() --> ClaseA::por()
              --> ClaseB::por() --> ClaseA::por()
```



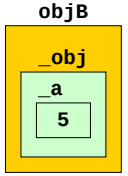
. . . / . . .

Objetos dentro de objetos y encadenamiento de mensajes

objA

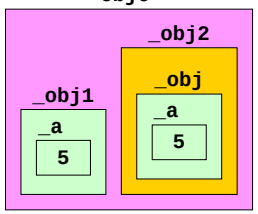

```
objB.igual(objA);
```

objB




```
objC.obj1(objA);
objC.obj2(objB);
```

objC



. . . / . . .

Objetos dentro de objetos y encadenamiento de mensajes

```
cout << objC.mas() << endl;
```

Aparece en la pantalla 10 (5 + 5)

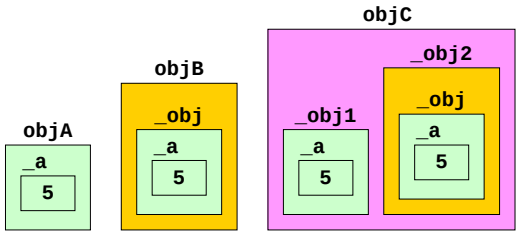
```
ClaseC::mas() --> ClaseA::mas()
              --> ClaseB::mas() --> ClaseA::mas()
```

```
cout << objC.por() << endl;
```

Aparece en la pantalla 25 (5 x 5)

```
ClaseC::por() --> ClaseA::por()
              --> ClaseB::por() --> ClaseA::por()
```

```
  return 0;
}
```



El problema de las inclusiones múltiples

El ejercicio que acabamos de ver es un claro ejemplo de las situaciones en las que hay que prevenir el problema de las inclusiones múltiples. Veamos cuál es el problema.

La **ClaseB** necesita la **ClaseA**, ya que tiene declarado un atributo de dicha clase. Por esto en la **ClaseB** se incluye la **ClaseA**.

La **ClaseC** necesita la **ClaseA** y la **ClaseB**, ya que tiene declarados atributos de dichas clases. Por esto en la **ClaseC** se incluye tanto la **ClaseA** como la **ClaseB**.

El programa principal necesita la **ClaseA**, la **ClaseB** y la **ClaseC**, ya que crea objetos de todas esas clases. Por esto en el programa principal se incluye tanto la **ClaseA** como la **ClaseB** como la **ClaseC**:

```
#include "claseA.h"
#include "claseB.h" // Vuelve a incluir la clase A
#include "claseC.h"
// Vuelve a incluir la clase A
// Vuelve a incluir la clase B
// (que incluye de nuevo la clase A)
```


El problema de las inclusiones múltiples

Como se ve, se intenta incluir cuatro veces la cabecera **claseA.h** y dos veces la cabecera **claseB.h**. El resultado, como se podría esperar, sería que se declararía 4 veces la **ClaseA** y dos veces la **ClaseB**. El compilador se quejaría por las declaraciones múltiples.

Debemos evitar que se incluya más de una vez una biblioteca (clase). Una forma sencilla de conseguirlo es mediante directivas del preprocesador.

La primera vez que se incluye la cabecera definimos un símbolo con la directiva **#define**. Para no volver a incluir la cabecera, lo primero que hacemos en ella es comprobar si el símbolo ya está definido; si lo está, entonces es que ya se ha incluido una vez la cabecera y no hay que volver a hacerlo; si el símbolo no está definido, entonces no se ha incluido todavía la cabecera y procedemos a hacerlo.

Para que este método funcione es importante que en cada cabecera se defina un símbolo único,uno que no se use en ninguna otra cabecera. Un convenio sencillo que asegura esto es utilizar como símbolo el nombre del archivo de cabecera, sustituyendo el punto por un _.

El problema de las inclusiones múltiples

Veamos como ejemplo cómo debe quedar la cabecera **claseB.h**:

```
#ifndef claseB_h
#define claseB_h

#include "claseA.h"
class ClaseB {
public:
  void init(int = 0);
  int mas(int = 0);
  int por(int = 1);
  void igual(ClaseA);
private:
  ClaseA _obj;
};

#endif
```

Los archivos de implementación no se ven afectados.

Las cabeceras de las bibliotecas del sistema (como por ejemplo **iostream**) ya vienen protegidas (si no, en el caso de que se utilizara **iostream** en las tres clases habríamos incluido ¡siete veces! **iostream**).

Ejercicio práctico

Una primera clase (*resolución en grupo*)

1. Crear una clase que se denomine **Contador1** que defina un único atributo **_cont** (un entero).
   El contador se podrá inicializar a cualquier valor no negativo.
   Otros servicios que han de proporcionar los objetos de esta clase son: incrementar el contador (en una unidad), decrementar el contador (en una unidad) y mostrar el valor actual del contador.
   Implementa todos los métodos dentro de la estructura **class**.
   El contador nunca podrá tener un valor negativo.
   Prueba la clase en una función **main()**.

Ejercicio práctico

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

class Contador1 {
public:
  void inicializa(int val = 0) { if(val < 0) val = 0; _cont = val; }
  void incrementar() { _cont++; }
  void decrementar () { if(_cont > 0) _cont--; }
  void mostrar() { cout << _cont << endl; }
private:
  int _cont;
};

int main()
{
  Contador1 c;
  c.inicializa();
  c.mostrar();
  c.incrementar();
  c.mostrar();
  c.decrementar();
  c.mostrar();
  Contador1 otro;
  otro.inicializa(5);
  otro.mostrar();
  otro.incrementar();
  otro.mostrar();
  otro.decrementar();
  otro.mostrar();

  return 0;
}
```

>>> cont1.cpp

Ejercicio práctico

Cada cosa en su sitio (*resolución en grupo*)

2. A partir de la clase anterior, crea otra clase **Contador2** que sea igual que la anterior, pero que tenga todos los métodos implementados fuera de la estructura **class**.
   Prueba la clase en una función **main()**.
   ¿Qué has tenido que cambiar en la función **main()**?
   Compara la interfaz de esta clase y la de la anterior: ¿cuál queda más clara?

Lo que hay que hacer:

1. Copiar los métodos a continuación de la estructura **class**.
2. *Limpiar* las cabeceras dentro de la estructura **class** para que sean prototipos (sustituir el cuerpo por ; y quitar los nombres de los parámetros).
3. Fuera de la estructura **class**, añadir el nombre de la clase y el operador de resolución de ámbito delante de los nombres de las funciones y reformatear el código.

Ejercicio práctico

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

class Contador2 {
public:
  void inicializa(int = 0);
  void incrementar();
  void decrementar ();
  void mostrar();
private:
  int _cont;
};

void Contador2::inicializa(int val) {
  if(val<0) val = 0; _cont = val;
}

void Contador2::incrementar(){
  _cont++;
}
```

. . . / . . .

Ejercicio práctico

```
void Contador2::decrementar () {
  if(_cont>0) _cont--;
}

void Contador2::mostrar() {
  cout << _cont << endl;
}

int main()
{
  Contador2 c;
  c.inicializa();
  c.mostrar();
  c.incrementar();
  c.mostrar();
  c.decrementar();
  c.mostrar();
  Contador2 otro;
  otro.inicializa(5);
  otro.mostrar();
```

. . . / . . .

Ejercicio práctico

```
  otro.incrementar();
  otro.mostrar();
  otro.decrementar();
  otro.mostrar();

  return 0;
}
```

La función **main()** es prácticamente la misma.
(Sólo cambia el nombre de la clase.)

La interfaz es mucho más clara ahora.

```
class Contador1 {
public:
  void inicializa(int val = 0)
  { if(val<0) val = 0; _cont = val; }
  void incrementar() { _cont++; }
  void decrementar ()
  { if(_cont>0) _cont--; }
  void mostrar()
  { cout << _cont << endl; }
...
```

```
class Contador2 {
public:
  void inicializa(int = 0);
  void incrementar();
  void decrementar ();
  void mostrar();
...
```


En el mundo real

Una clase más útil (*resolución en grupo*)

Crear una clase **Empleado** que modele la información que una empresa mantiene sobre cada empleado: número de **DNI** (*entero largo*), **sueldo base** (*real*), **pago por hora extra** (*real*), **horas extra** realizadas en el mes (*real*), **tipo** (*porcentaje*) de IRPF (*real*), **casado** o no (*verdadero/falso*) y número de **hijos** (*entero*).

porcentaje == real

```
long int _dni;
bool _casado;
int _hijos;
double _sueldoBase;
double _tipo;
double _pagoHoraExtra;
double _horasExtra;
```

. . . / . . .

En el mundo real

La clase debe contemplar **accedentes** y **mutadores** para todos los atributos *menos mutador para el correspondiente al DNI*.
Al **inicializar** cada objeto *se podrá proporcionar el DNI correspondiente*.

```
void inicializa(long int = 0);
long int dni();
bool casado();
void casado(bool);
int hijos();
void hijos(int);
double sueldoBase();
void sueldoBase(double);
double tipo();
void tipo(double);
double pagoHoraExtra();
void pagoHoraExtra(double);
double horasExtra();
void horasExtra(double);
```

Los demás servicios que deberán proporcionar los objetos de la clase serán los siguientes:

Cálculo y devolución del complemento correspondiente a las horas extra realizadas.

```
double complemento();
```

. . . / . . .

En el mundo real

Cálculo y devolución del sueldo bruto. `double sueldoBruto();`

Cálculo y devolución de las retenciones (IRPF) a partir del tipo, teniendo en cuenta que el porcentaje de retención que hay que aplicar es el tipo menos 2 puntos si el empleado está casado y menos 1 punto por cada hijo que tenga; el porcentaje se aplica sobre todo el sueldo bruto.

`double retenciones();`

Visualización de la información básica del empleado. `void mostrar();`

Visualización de toda la información del empleado. la básica más el sueldo base, el complemento por horas extra, el sueldo bruto, la retención de IRPF y el sueldo neto.

`void mostrarTodo();`

Todos los métodos se han de implementar fuera de la estructura **class**.
Prueba 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

#include <iostream>
using namespace std;

class Empleado {
public:
  void inicializa(long int = 0);
  long int dni();
  bool casado();
  void casado(bool);
  int hijos();
  void hijos(int);
  double sueldoBase();
  void sueldoBase(double);
  double tipo();
  void tipo(double);
```

. . . / . . .

En el mundo real

```
  double pagoHoraExtra();
  void pagoHoraExtra(double);
  double horasExtra();
  void horasExtra(double);
  double complemento();
  double sueldoBruto();
  double retenciones();
  void mostrar();
  void mostrarTodo();
private:
  long int _dni;
  bool _casado;
  int _hijos;
  double _sueldoBase;
  double _tipo; // porcentaje de IRPF para impuestos
  double _pagoHoraExtra;
  double _horasExtra;
};

#endif
```


En el mundo real

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

#include "empleado.h"

void Empleado::inicializa(long int dni) {
  _dni = dni;
  _casado = false;
  _hijos = 0;
  _sueldoBase = 0;
  _tipo = 0;
  _pagoHoraExtra = 0;
  _horasExtra = 0;
}

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

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

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

. . . / . . .

En el mundo real

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

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

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

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

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

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

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

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

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

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

. . . / . . .

En el mundo real

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

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

double Empleado::retenciones() {
// 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;
}
```

. . . / . . .

En el mundo real

```
void Empleado::mostrar() {
  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() {
  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 - prueba.cpp

#include "empleado.h"

int main()
{
  Empleado emp;
  emp.inicializa(333666);
  emp.casado(true);
  emp.hijos(1);
  emp.sueldoBase(200000);
  emp.tipo(18);
  emp.pagoHoraExtra(7000);
  emp.horasExtra(3.5);
  emp.mostrar();
  emp.mostrarTodo();

  return 0;
}
```