FACULTAD DE INFORMÁTICA

Más sobre clases y objetos

TEMA

Programación orientada a objetos — Unidad 2

Autor: Luis Hernández Yáñez

FdI UCM

Contenido

FdI UCM

Contenido

FdI UCM

Constructores

Un método constructor (o simplemente *constructor*) es una función miembro especial (función *constructora*) que lleva a cabo una inicialización automática de cada objeto de la clase en el momento en que se declara.

Declaración de un constructor:

- ✓ Función miembro pública con el mismo nombre que la clase.
- ✓ Sin indicación de tipo devuelto (ni siquiera **void**).

Se ejecuta automáticamente al crearse un objeto de esa clase.

```
class MiClase {
public:
  MiClase(); // constructor
  ...
private:
  ...
```

Constructores

El constructor puede implementarse fuera de la declaración de la clase, como las demás funciones miembro:

```
MiClase::MiClase() // constructor (implementación)
{
  ...
```

Pero a menudo se implementa como función insertada en la propia declaración de la clase:

```
class MiClase {
public:
  MiClase() { ... }
  ...
private:
  ...
```


Inicialización de objetos sin constructores

Función miembro específica para inicialización.

```
class Punto {
public:
  void inicializa() { _x = 0; _y = 0; }
  void x(double f) { _x = f; }
...
private:
  double _x, _y;
};

int main()
{
  Punto p;
  p.inicializa();
  ...
```

Paso de mensaje de inicialización.

Inicialización de objetos con constructores

Función miembro constructora (constructor).

```
class Punto {
public:
  Punto() { _x = 0; _y = 0; }
  void x(double f) { _x = f; }
...
private:
  double _x, _y;
};

int main()
{
  Punto p;
  ...
```

Se llama automáticamente al constructor.

Sin paso de mensaje de inicialización.

El punto **p** ve inicializados sus atributos **_x** e **_y** a 0.

Una mejor forma de inicializar atributos

Uso de inicializadores

Inicializador:
nombre-del-atributo (*valor-inicial*)

```
class Punto {
public:
  Punto() : _x(0), _y(0) // lista de inicializadores
  { /* cuerpo vacío */ }
  void x(double f) { _x = f; }
...
private:
  double _x, _y;
};

int main()
{
  Punto p;
  ...
```

Los atributos reciben los valores iniciales antes de que se ejecute el cuerpo del constructor (importante en ciertas ocasiones).

Además, es la única forma de inicializar atributos **const**.

Una mejor forma de inicializar atributos

```
class Circulo {
public:
  Circulo() : _radio(0) { } // constructor
...
private:
  Punto _centro;
  double _radio;
};

int main()
{
  Circulo c;
  ...
```

Al declarar (definir) el objeto **c**...

1. Se crean sus atributos (al crear el atributo **_centro** se ejecuta sobre él su constructor, el de su clase **Punto**).
2. Se inicializan los atributos que figuran en la lista de inicializadores (**_radio**, en este caso).
3. Se ejecuta el código del constructor (nada en este caso).

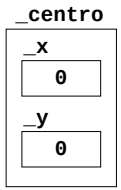
Una mejor forma de inicializar atributos

```
class Circulo {
public:
  Circulo() :
    _radio(0) { }
...
private:
  Punto _centro;
  double _radio;
};

int main()
{
  Circulo c;
```

Se crea un objeto de clase **Punto** para el atributo **_centro** del objeto **c** de clase **Circulo**.

Sobre ese objeto de la clase **Punto** se ejecuta el constructor de esa clase, inicializándose sus atributos **_x** e **_y**.



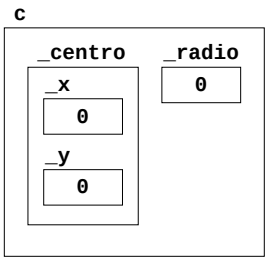
Una mejor forma de inicializar atributos

```
class Circulo {
public:
  Circulo() :
    _radio(0) { }
...
private:
  Punto _centro;
  double _radio;
};

int main()
{
  Circulo c;
```

Con el objeto de la clase **Punto** creado se crea el objeto **c** de clase **Circulo**.

Sobre este objeto de la clase **Circulo** se ejecuta el constructor de esa clase, inicializándose su atributo **_radio** y estableciéndose el **Punto** como atributo **_centro**.



Constructores parametrizados

Funciones constructoras con parámetros

```
class Punto {
public:
  Punto(double a, double b) : _x(a), _y(b) { }
  void x(double f) { _x = f; }
...
private:
  double _x, _y;
};

int main()
{
  Punto p1 = Punto(2,3);
  Punto p2(1,2);
  ...
```

Dos formas de invocar el constructor.

Método taquigráfico (el usual).

Sobrecarga de funciones constructoras

```
class Punto {
public:
  Punto() : _x(0), _y(0) { } // constructor 1
  Punto(double a, double b) // constructor 2
    : _x(a), _y(b) { }
  void x(double f) { _x = f; }
...
private:
  double _x, _y;
};

int main()
{
  Punto p1;
  Punto p2(1,2);
  ...
```

Dos funciones constructoras: dos formas de inicialización.

Argumentos implícitos en constructores

Siempre y cuando no pueda haber ambigüedad.

```
class Circulo {
public:
  Circulo() : _radio(0) { } // constructor 1
  Circulo(Punto p, double f = 0) // constructor 2
    : _centro(p), _radio(f) { }
...
};

int main()
{
  Punto p;
  ...
  Circulo c1; // invoca constructor 1
  Circulo c2(p); // invoca constructor 2 (f es 0)
  Circulo c3(p,5); // invoca constructor 2 (f es 5)
```

En este ejemplo no hay ambigüedad.

Argumentos implícitos en constructores

```
class Punto {
public:
  Punto() : _x(0), _y(0) { }
  Punto(double a = 1, double b = 1) : _x(a), _y(b) { }
...
private:
  double _x, _y;
};

int main()
{
  Punto p;
  ...
```

AMBIGÜEDAD

¿A que función constructora se ha de llamar?

El constructor predeterminado (por defecto)

```
class Punto {
public:
  Punto() : _x(0), _y(0) { } // predeterminado
                             // (sin argumentos)
  Punto(double a) : _x(a), _y(0) { } // un argumento
  Punto(double a, double b) : _x(a), _y(b) { }
  // dos argumentos
...
private:
  double _x, _y;
};
...
Punto p1, p2(1), p3(1,2);
```

Los tres constructores se pueden reducir a uno:

```
Punto(double a = 0, double b = 0) : _x(a), _y(b) { }
```

El constructor predeterminado (por defecto)

Necesidad del constructor predeterminado

En las clases debe existir un constructor predeterminado que no contemple argumentos (que se pueda invocar sin argumentos).

El constructor predeterminado se invoca automáticamente cada vez que se crean objetos sin proporcionar información de inicialización (por parte del programador o automáticamente por el compilador).

Por ejemplo, cuando se crea un objeto sin hacer uso de argumentos:

```
Punto p1;
```

se ejecuta automáticamente el constructor predeterminado.

O cuando se declara un array de objetos:

```
Punto p[10];
```

se ejecuta automáticamente el constructor predeterminado sobre todos los objetos del array (más adelante veremos los arrays).

El constructor predeterminado (por defecto)

¿Qué pasa si no implemento el constructor predeterminado?

Si en la clase no se proporciona explícitamente ningún constructor, el compilador añade automáticamente un constructor predeterminado que usará en la creación de los objetos.

Este constructor predeterminado que incluye automáticamente el compilador hará bien su trabajo siempre que no se hayan declarado en la clase atributos accedidos a través de punteros.

Si en la clase se han definido otros constructores, pero ninguno sirve como constructor predeterminado (sin argumentos), el compilador se quejará y tendremos que incluir uno.

Lo mejor es crear explícitamente los constructores adecuados en las clases, incluyendo siempre entre ellos el constructor predeterminado.

El constructor predeterminado (por defecto)

```
Punto() : _x(0), _y(0) { } // c. predeterminado
Punto(double a, double b) : _x(a), _y(b) { }
--------
Circulo() : _radio(0) { } // c. predeterminado
Circulo(Punto p, double f = 0) :
  _centro(p), _radio(f)
--------
Punto(double a = 0, double b = 0) : _x(a), _y(b) { }
// constructor predeterminado
// (se puede invocar sin argumentos)
```


El constructor de copia

Otra forma más de inicializar un objeto:
A partir de otro objeto de la misma clase (inicialización por copia).

```
Punto p1(2,5); // inicialización normal
Punto p2(p1); // inicialización por copia
Punto p3 = p1; // inicialización por copia
```

No es imprescindible declarar un constructor de copia en la clase.
A todas las clases se les añade automáticamente un constructor de copia predeterminado que permite inicializar una copia.

El constructor de copia predeterminado (por defecto) realiza una copia bit a bit de un objeto en el otro.

La copia bit a bit funciona bien mientras no se tengan en los objetos atributos que se acceden por medio de punteros: en esos casos, la copia bit a bit hará que el objeto original y su copia tengan atributos que apunten a los mismos objetos (compartirán memoria).

El constructor de copia

Invocación del constructor de copia

El constructor de copia se invoca automáticamente cada vez que se necesita crear una copia de un objeto.

Cuando se crea un objeto a partir de otro existente, el objeto que se crea se inicializa ejecutando sobre él el código del constructor de copia de la clase. Si no se ha definido constructor de copia en la clase, se copian bit a bit los atributos del objeto original (el que ya existía) sobre los atributos del objeto que se crea.

Los siguientes son ejemplos de creación de un objeto a partir de otro:

```
Punto p1;
...
Punto p2(p1); // se crea p2 a partir de p1
Punto p3 = p1; // se crea p3 a partir de p1
```


El constructor de copia

El constructor de copia también se invoca cuando se pasa un objeto como argumento para un parámetro por valor de una función.

```
void f(Punto p) { ... }
Punto p1;
...
f(p1); // se crea una copia en el parámetro p
```

Al comenzar la ejecución de la función se crea el objeto parámetro (**p**) como una copia del objeto parámetro (**p1**).

Y también se invoca el constructor de copia cuando un método devuelve, por valor, un objeto como resultado.

```
Punto f() { Punto tmp; ... return tmp; }
...
f().x(12);
```

Al terminar la ejecución de la función se crea una copia del objeto que se devuelve (**tmp**) y luego se pasa a ese objeto copia el mensaje **x()**.

Constructores de copia personalizados

```
class Punto {
public:
  Punto(double a = 0, double b = 0) : _x(a), _y(b) { }
  Punto(const Punto& p) // constructor de copia
  { _x = p._x; _y = p._y; }
...
private:
  double _x, _y;
};
...
Punto p1(1,2);
Punto p2(p1); // inicialización por copia
Punto p3 = p1; // inicialización por copia
```

Cuando los objetos tienen atributos que se acceden por medio de punteros, los constructores de copia resultan imprescindibles, ya que se han de crear *manualmente* copias de los atributos. Lo veremos.

Constructores de copia personalizados

```
...
Punto p2(p1); // inicialización por copia 1
Punto p3 = p1; // inicialización por copia 2
```

En el primer caso el objeto que se crea es **p2**. En el segundo caso el objeto que se crea es **p3**. En ambos casos el objeto origen que se usa como punto de partida es **p1**.

Como se trata de un constructor, en el código del constructor de copia los atributos sin cualificar se refieren a los del objeto que se crea.
El parámetro es el objeto a partir del que se crea la copia (**p1**).

```
Punto(const Punto& p) { _x = p._x; _y = p._y; }
```

Primer caso: `{ p2._x = p1._x; p2._y = p1._y; }`

Segundo caso: `{ p3._x = p1._x; p3._y = p1._y; }`

Referencias

En los constructores de copia, el objeto parámetro a partir del que se realizará la copia está declarado como una referencia:

```
Punto(const Punto& p)
```

Una referencia (&) es un puntero constante que se *desreferencia* de forma automática.

```
int x = 0;
int& a = x;
```

x es una variable normal de tipo **int** que se inicializa a cero.

a es una referencia: un puntero a un dato de tipo **int**.

La referencia es constante, de forma que **a** siempre va a apuntar al mismo dato **int** y no puede pasar a apuntar a otro dato **int**.

➡ El dato al que va a apuntar **a** (**x** en este caso) se debe vincular con la referencia en el momento en que se crea ésta.

Referencias

```
int& a = x;
```

Como puntero que es, la referencia se utiliza para manipular aquello a lo que apunta (**x** en el caso de **a**).

Pero, a diferencia de los punteros normales, no se puede hacer que apunte a otro dato (es decir, modificar el puntero en sí, la dirección que contiene); recordemos que es un puntero constante.

Así, **a** siempre va a apuntar a **x**.

El puntero y aquello a lo que apunta se confunden.

Es por esto que, una vez que la referencia está creada, automáticamente se *desreferencia* para que sea utilizada más cómodamente: sin necesidad de colocar el * delante para acceder a lo apuntado.

Referencias

Cuando se utiliza la referencia **a** se utiliza aquello a lo que apunta (**x** en este caso). Si se accede a la referencia **a** se accede realmente a **x**. Y si se modifica **a** realmente se está modificando **x**.

```
int x = 0;
int& a = x;
cout << x << " : " << a << endl;
a++;
cout << x << " : " << a << endl;
```

El puntero y aquello a lo que apunta se confunden.

El uso anterior de las referencias puede no tener mucho sentido, ya que se puede utilizar directamente **x** sin necesidad de ninguna referencia suya. El uso de las referencias cobra sentido con los parámetros de las funciones y sus valores devueltos.

Referencias como parámetros de funciones

Las referencias simplifican el uso de los parámetros por referencia en las funciones:

```
void f(int& d)
{ d++; }
```

Declarando el parámetro **d** como una referencia, cualquier modificación que se realice sobre el parámetro se reflejará sobre el argumento.

Aunque la referencia se maneja en el cuerpo de la función como si no fuera un puntero, internamente el dato se accede mediante un puntero, lo que hace posible que sea modificado.

El vínculo entre la referencia y aquello a lo que va a puntar se establece en el momento en que se invoca la función, momento en el que se crea la referencia y se conoce el argumento con el que se debe vincular.

Referencias como parámetros de funciones

Las referencias se utilizan principalmente para el paso de parámetros por referencia: cuando se necesita que el argumento, al finalizar la ejecución de la función, refleje los cambios que se le hayan realizado durante la ejecución de la función.

Sin embargo, hay otros casos en los que se utilizan parámetros por referencia aún cuando no se va a realizar ningún cambio sobre el argumento.

Estos casos son aquellos en los que el dato (u objeto) que se pasa tiene un tamaño considerable y se quiere ahorrar el tiempo que supone la copia del argumento en el parámetro por valor.

Si la función no va a modificar el argumento y éste se quiere pasar como referencia, el parámetro deberá estar declarado como una referencia constante.

Referencias constantes

Un parámetro por referencia que sea una referencia constante se declara colocando delante del tipo el modificador **const**:

```
void f(const int& d)
```

Si el código de la función intenta modificar de alguna forma el dato, se producirá un error de compilación.

Cuando se declara un parámetro referencia constante que es un objeto, como se hace en el constructor de copia:

```
Punto(const Punto& p)
```

al objeto sólo se le pueden pasar mensajes que correspondan a métodos constantes (que los veremos enseguida) y no se puede modificar de ninguna forma ninguno de sus atributos.

En el constructor de copia esto significa que el objeto **p** de clase **Punto** que se usa para crear la copia no se modificará. ¡Normal!

Devolución de referencias

Las referencias también se pueden utilizar en las funciones para devolver resultados con la instrucción **return**.

El mecanismo normal de devolución de datos en las funciones es el del paso de parámetros por valor: devolver una copia.

```
Punto f()
{
  Punto tmp;
  ...
  return tmp;
}
```

Se devuelve una copia del objeto **tmp** (y el objeto local **tmp** se destruye); la copia se asignará a algún otro objeto en el lugar en que se realizó la invocación o se usará pasándole algún mensaje.

```
Punto p;
p = f();                f().x(12);
```


Devolución de referencias

Si el dato a devolver se declara como una referencia:

```
Punto& f()
{ ... }
```

no se devuelve ninguna copia, sino el propio objeto que figure en la instrucción **return**.

Lo que se devuelve realmente es la dirección de ese objeto, aunque en forma de referencia.

El uso principal de la devolución de referencias se encontrará (como veremos al final de esta unidad) en los métodos que deben devolver como resultado el propio objeto receptor del mensaje.

ATENCIÓN: Hay que tener muy presente en estos casos que lo que se debe devolver como referencia (con la instrucción **return**) es un objeto que no se destruya al terminar la ejecución del método, ya que si no, la dirección de la referencia no apuntará a nada existente.

Destructores

Los destructores son funciones miembro que realizan tareas de "limpieza". Más adelante veremos su utilidad.

Al igual que los constructores no van precedidas de un tipo de valor devuelto. Su nombre es el de la clase precedido por tilde (~). No toman argumentos.

```
class Punto {
public:
   Punto() : _x(0), _y(0) { }
   ~Punto() { } // destructor
...
private:
   double _x, _y;
};
```

Se ejecuta automáticamente cuando se destruye un objeto.

Construcción y destrucción de objetos

Los objetos se construyen cuando el programa así lo indica explícitamente: cuando se declara el identificador correspondiente[1]. Así, sabemos perfectamente cuándo se ejecutan los constructores.

Pero, ¿cuándo se ejecutan los destructores? Cuando termina la ejecución del bloque de código (normalmente una función) en el que se han creado los objetos.

```
int main() {
  Punto p; // Primero se crea un Punto
  Circulo c; // A continuación se crea un Circulo
  ...
};
```

Al terminar la ejecución de la función **main()** se destruyen los dos objetos, **p** y **c**. ¿En qué orden? En el orden contrario al de definición. Se destruye antes **c** que **p**.

[1] Los objetos dinámicos, como veremos, se crean y se destruyen de forma diferente.

Asignación (copia) de objetos

Para todas las clases está definido de forma automática el operador de asignación:

```
Punto p1(2,5);
Punto p2;
p2 = p1; // copia de p1 en p2
```

Se copian, bit a bit, todos los atributos. Si algún atributo es un puntero, la copia (asignación) normal no resultará adecuada, ya que los dos objetos compartirán memoria.

En esos casos se puede sobrecargar el operador de asignación:

```
class Punto {
...
  Punto& operator=(const Punto&);
```

Su código se ha de encargar de realizar adecuadamente la copia.

Asignación (copia) de objetos

```
...
p2 = p1; // copia de p1 en p2
```

El objeto que figura a la izquierda del = es el objeto que recibe el mensaje. El mensaje es el operador de asignación. Y el objeto a la derecha del = es el objeto origen de la copia.

```
Punto& operator=(const Punto& p)
{ _x = p._x; _y = p._y; return *this; }
```

El parámetro (**p**) representa el objeto origen de la copia (**p1** en el ejemplo de arriba). Los atributos sin cualificar son los del objeto sobre el que se copia (el receptor del mensaje; **p2** en el ejemplo).

La función termina devolviendo el propio objeto receptor (veremos esto con detenimiento más adelante).

En la siguiente unidad veremos más sobre métodos operadores y la sobrecarga de operadores en las clases.

La forma canónica ortodoxa para las clases

La Forma canónica ortodoxa (FCO) establece una serie de métodos importantes que se deben declarar en las clases:

- ✓ Constructor predeterminado (sin argumentos)
  Inicialización de objetos cuando no se proporciona ningún valor
- ✓ Constructor de copia
  Se usa, por ejemplo, para el paso de parámetros por valor
- ✓ Operador de asignación (=)
  Copias de objetos
- ✓ Destructor
  Operaciones de terminación

Aunque algunos de esos métodos no hagan nada adicional, su existencia dejará constancia de que el programador ha tomado en consideración la implementación adecuada de esos elementos.

Nuestras clases se ajustarán siempre a la FCO.

Atributos y memoria

Por lo que sabemos hasta ahora, cada objeto tiene su propio conjunto de atributos. Sin embargo, puede haber atributos compartidos.

Atributos compartidos por todos los objetos de la clase

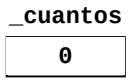
Un atributo que se declara como **`static`** sólo se crea (inicializa) una vez, siendo un *dato* único que se encuentra fuera de todos los objetos de la clase, pero que puede ser accedido por todos ellos.

```
class MiClase {
public:
   MiClase() { _cuantos++; }
...
private:
   static int _cuantos; // número de objetos creados
   double _f;
};
int MiClase::_cuantos = 0;
```

Inicialización fuera de la clase
Para que se inicialice una sola vez y no cada vez que se cree un objeto de la clase.

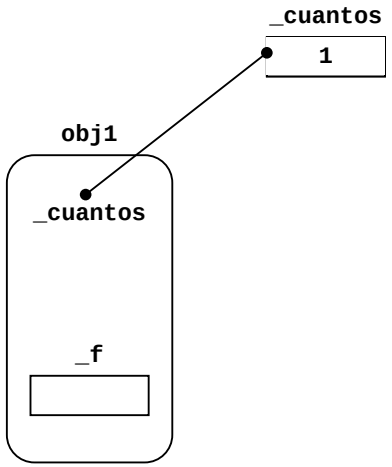
Atributos compartidos

```
MiClase obj1, obj2, obj3;
```



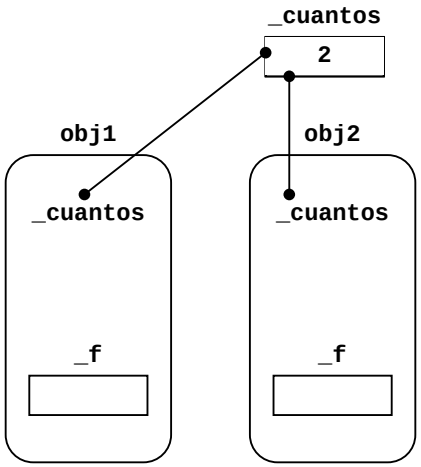
Atributos compartidos

```
MiClase obj1, obj2, obj3;
```



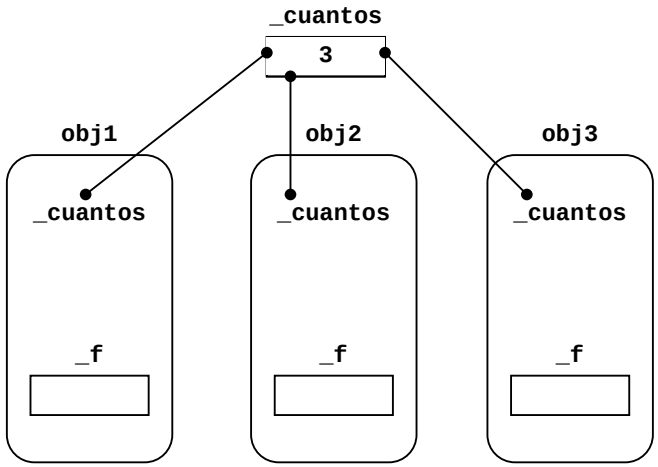
Atributos compartidos

```
MiClase obj1, obj2, obj3;
```



Atributos compartidos

```
MiClase obj1, obj2, obj3;
```



Métodos *constantes*

Un método *constante* (función miembro **const**) garantiza que no va a modificar en ningún momento algún atributo.

Para que la función miembro sea **const**, basta poner esa palabra reservada tras la lista de parámetros y antes del cuerpo:

```
class MiClase {
...
  void mostrar() const;
```

Si, por error, la función intenta modificar algún atributo, el compilador detectará el error y lo notificará.

Objetos *constantes*

Si un objeto se declara como constante:

```
const Punto p(1,2); // p es un objeto constante
```

no se podrá modificar de ninguna forma.

Es decir, sólo se le podrán pasar mensajes que se correspondan con métodos constantes.

Objetos argumentos constantes

No se podrá modificar en un método un argumento que esté declarado como constante.

```
class MiClase {
...

   void metodo(const Punto& p)
   { ...
```

Parámetro por referencia **const**
(El argumento de un parámetro por valor nunca se modifica, ya que se usa una copia.)

Motivo: eficiencia (no se pasa una copia, sino una referencia).