FACULTAD DE INFORMÁTICA

Más sobre clases y objetos

RESUMEN

Programación orientada a objetos — Unidad 2

Autor: Luis Hernández Yáñez

Inicialización de objetos sin constructores

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

void Punto::inicializa() { _x = 0; _y = 0; }
...
int main()
{
  Punto p;
  p.inicializa();
  ...
```

Función miembro específica para inicialización

NO se pueden inicializar los atributos directamente en su declaración (`double _x = 0, _y = 0;` es incorrecto)

Paso de mensaje de inicialización

Inicialización de objetos con constructores

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

Punto::Punto() { _x = 0; _y = 0; }
...

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

Función miembro constructora (constructor)

Mismo nombre que la clase y sin indicación de tipo

Se llama automáticamente al constructor

Sin paso de mensaje de inicialización

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

Uso de inicializadores

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

Punto::Punto() : _x(0), _y(0) // Inicializadores
{ /* Cuerpo vacío */ }
...
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**.

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

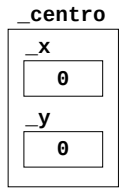
Constructores en cadena

```
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**.



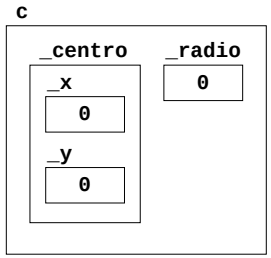
Constructores en cadena

```
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

```
class Punto {
public:
  Punto(double, double);
  void x(double);
...
private:
  double _x, _y;
};
Punto::Punto(double a, double b) : _x(a), _y(b) { }
...
int main()
{
  Punto p1 = Punto(2,3);
  Punto p2(1,2);
  ...
```

Dos formas de invocar el constructor

Método taquigráfico (el usual)

Sobrecarga y argumentos implícitos en constructores

```
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);
```

Varios constructores: varias formas de inicialización

Debemos incluir SIEMPRE el constructor predeterminado

Los tres constructores se pueden reducir a uno:

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


El constructor de copia

```
class Punto {
public:
  Punto(double = 0, double = 0);
  Punto(const Punto&); // Constructor de copia
private:
  double _x, _y;
};
Punto::Punto(double a, double b) : _x(a), _y(b) { }
Punto::Punto(const Punto& p) {
  _x = p._x; _y = p._y;  // Se copian los atributos
}
...
Punto p1(1,2); // Inicialización normal
Punto p2(p1); // Inicialización por copia
Punto p3 = p1; // Inicialización por copia
```

Se encarga de copiar los atributos del objeto argumento sobre los del objeto que se crea.

Debemos incluir SIEMPRE el constructor de copia

Referencias para parámetros pasados por variable

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.

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. (Datos u objetos de tamaño considerable: se ahorra el tiempo de copia al parámetro.)

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 en las funciones

Si el dato a devolver por la función 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". 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();
   ~Punto(); // Destructor
...
private:
   double _x, _y;
};

Punto::Punto() : _x(0), _y(0) { }
Punto::~Punto() { } // Destructor
```

Debemos incluir SIEMPRE el destructor

Los objetos se destruyen en el orden inverso al de su construcción

Se ejecuta automáticamente cuando se destruye un objeto.

Asignación (copia) de objetos

El operador de asignación ha de copiar los atributos uno a uno

```
class Punto {
public:
  Punto(double = 0, double = 0);
  Punto& operator=(const Punto&); // Asignación
private:
  double _x, _y;
};
Punto::Punto(double a, double b) : _x(a), _y(b) { }
Punto& Punto::operator=(const Punto& p) {
  _x = p._x; _y = p._y; return *this;
}
...
Punto p1(1,2);
Punto p2;
p2 = p1; // Copia (asignación)
```

Debemos incluir SIEMPRE el operador de asignación

La forma canónica ortodoxa para las clases (FCO)

- ✓ 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.

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.

Más usos de **`const`**

Métodos *constantes*

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

```
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.