

# Programación Orientada a Objetos

## Tema 2: El paradigma de la programación orientada a objetos

José Fidel Argudo Argudo    Francisco Palomo Lozano  
Inmaculada Medina Bulo    Gerardo Aburrizaga García



Versión 1.0



# Índice

- 1 Introducción
- 2 Objetos
- 3 Clases
- 4 Relaciones entre clases
- 5 Implementación de clases
- 6 Construcción y uso de objetos

# POO

- En POO un programa se organiza como un conjunto finito de objetos que contienen datos y operaciones y que se comunican entre sí mediante mensajes.
- De las interacciones, mediante el paso de mensajes, entre los objetos que componen un programa surge la funcionalidad del programa.
- Proceso de desarrollo orientado a objetos:
  - 1 Identificar los **objetos** que intervienen.
  - 2 Agrupar en **clases** los **objetos** con **características y comportamientos** comunes.
  - 3 Identificar los **datos** y **operaciones** de cada clase.
  - 4 Identificar las **relaciones** que puedan existir entre las clases.
- Principios en que se fundamenta la POO: **Abstracción, encapsulamiento, ocultación de información, generalización, polimorfismo.**

# Objetos

## Un objeto

- 1 es una entidad individual del problema que se está resolviendo, formada por la unión de un estado y un comportamiento;
- 2 es una entidad individual del programa que posee un conjunto de datos (atributos) y un conjunto de operaciones (métodos) que trabajan sobre ellos.

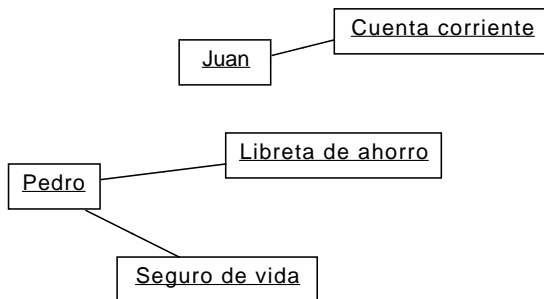
**Estado** Conjunto de valores de todos los atributos de un objeto en un instante. Tiene un carácter dinámico, evoluciona con el tiempo.

**Comportamiento** Agrupa todas las competencias del objeto y queda definido por las operaciones que posee. Actúan tras la recepción de un mensaje enviado por otro objeto.

**Identidad** Caracteriza la existencia de un objeto como ente individual. La identidad permite distinguir los objetos sin ambigüedad.

# Objetos

- Enlaces

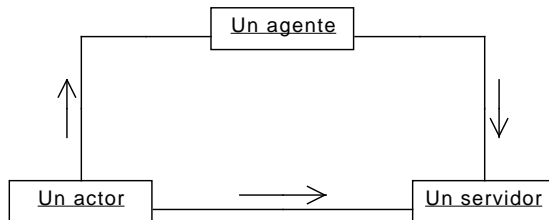


# Objetos

Actores Objetos que exclusivamente emiten mensajes

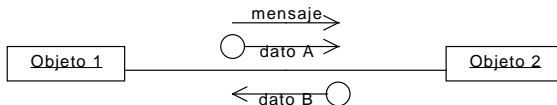
Servidores Objetos que únicamente reciben mensajes

Agentes Pueden emitir y recibir mensajes



# Mensajes

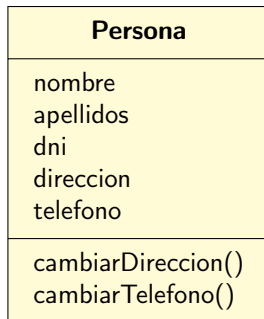
nombre[destino, operación, parámetros]



# Clases

## Una clase

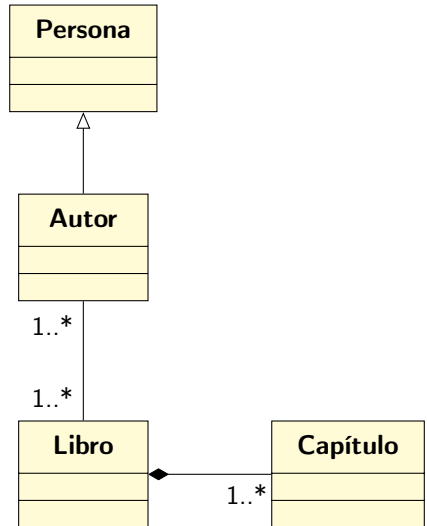
- 1 representa un concepto, idea, entidad, ... relevante del dominio del problema;
- 2 agrupa un conjunto de objetos de las mismas características;
- 3 y es la descripción mediante el lenguaje de programación de las propiedades del conjunto de objetos que la forman.





# Relaciones

- Dependencia
- Asociación (cardinalidad, multiplicidad, navegabilidad)
  - Agregación
  - Composición
- Generalización (simple y múltiple)
- Realización



# Implementación de clases

```
1 // reloj.h
2
3 #ifndef RELOJ_H_
4 #define RELOJ_H_
5
6 class Reloj {
7 public:
8     Reloj(int h = 0, int m = 0);
9     void incrementarHoras();
10    void incrementarMinutos();
11    void mostrar();
12 private:
13     int horas,
14         minutos;
15 };
16
17 #endif
```

Metodos  
mayoritariamente  
publico, podria haber  
alguno privado

Los atributos siempre  
serán privados

Por defecto, la accesibilidad de un miembro será privada (class)  
Con struct será pública

# Implementación de clases

```
1 // reloj.cpp

3 #include <iostream>
4 #include <iomanip> // para setfill() y setw()
5 #include "reloj.h"

7 // Constructor
8 Reloj::Reloj(int h, int m): horas(h), minutos(m) {}

10 // Funciones miembro modificadoras: incrementan los contadores
11 void Reloj::incrementarHoras()
12 {
13     horas = (horas + 1) % 24; // circular
14 }

16 void Reloj::incrementarMinutos()
17 {
18     minutos = (minutos + 1) % 60; // circular
19 }
```

# Implementación de clases

Const Reloj \* const this: Puntero this a reloj constante de una clase reloj constante  
funcion(const Reloj &r): La funcion utiliza el reloj pero no cambia el estado de Reloj r

NO SE PUEDE UTILIZAR UN OBJETO CONST CON UN METODO NO CONSTANTE  
CUANDO DECLAREMOS UN MEOTOD CONST, TENDREMOS QUE COMBINARLO CON UN OBJETO CONST

```
20 // Función miembro observadora: muestra las horas y los minutos
21 void Reloj::mostrar()
22 {
23     using namespace std;

25     cout << setfill('0')           // carácter de relleno : '0'
26         << setw(2) << horas       // horas: dos caracteres
27         << ':'                     // carácter separador: ':'
28         << setw(2) << minutos;    // minutos: dos caracteres
29 }
```

# Programa de prueba

```
1  // Ejemplo de uso de la clase Reloj

3  #include <iostream>
4  #include "reloj.h"

6  int main()
7  {
8      using namespace std;
9      Reloj r(23, 59);

11     cout << "Reloj:␣"; r.mostrar();
12     cout << "\nIncrementamos␣las␣horas:␣";
13     r.incrementarHoras(); r.mostrar();
14     cout << "\nIncrementamos␣los␣minutos:␣";
15     r.incrementarMinutos(); r.mostrar();
16     cout << endl;
17 }
```

# Atributos mutables

Aunque ese atributo forme parte de un objeto cte (const), si es mutable se puede modificar.

```
1 class Semaforo {
2 public:
3     // ...
4     void comprobar() const;
5     void cambiar();
6 private:
7     enum Color { ROJO, VERDE, AMBAR };
8     Color c;
9     mutable bool revisado; // ¿ha pasado la inspección?
10 };
```

# Miembros estáticos



```
1 // entrada.h

3 #ifndef ENTRADA_H_
4 #define ENTRADA_H_

6 class Entrada {
7 public:
8     Entrada(int s);
9     void imprimir() const;
10    static int n_entradas(int s);
11 private:
12    static double tarifa(int s);
13    static int proximo_asiento[];
14    int sesion,
15        asiento;
16    double precio;
17 };

19 #endif
```

Diagram illustrating static members in the `Entrada` class:

- `numero de la sesion` points to the `s` parameter in the constructor `Entrada(int s);`.
- `Numero de entradas para la sesion s` points to the static member function `static int n_entradas(int s);`.
- `Tarifa para las entradas de la sesion s` points to the static member function `static double tarifa(int s);`.
- `Asiento contiguo a la entrada *this` points to the `asiento` member variable.

# Miembros estáticos



```
1 // entrada.cpp

3 #include <iostream>
4 #include <iomanip>
5 #include "entrada.h"
6 using namespace std;

8 // El núm. inicial de asiento será el 1 para las cuatro sesiones
9 int Entrada::proximo_asiento[4] = { 1, 1, 1, 1 };

11 // Constructor de entrada (núm. de sesión entre 1 y 4)
12 Entrada::Entrada(int s)
13 {
14     sesion = s - 1;
15     asiento = proximo_asiento[sesion]++;
16     precio = tarifa(sesion);
17 }
```

Hay que declararlo en el ámbito global del programa



# Miembros estáticos

```
19 // Impresión de una entrada.
20 void Entrada::imprimir() const
21 {
22     cout << "Sesión_" << sesion + 1
23         << "\tAsiento_№_" << asiento
24         << "\t" << fixed << setprecision(2) << precio << "_EUR"
25         << endl;
26 }

28 // Entradas vendidas para una sesión (1-4)
29 int Entrada::n_entradas(int s)
30 {
31     return proximo_asiento[s - 1] - 1;
32 }

34 // Cálculo de la tarifa .
35 double Entrada::tarifa(int s)
36 {
37     return s == 1 || s == 4 ? 4.25 : 7.0;
38 }
```

# Miembros estáticos

- Los atributos static const de tipos enteros (bool, char, int) se pueden definir dentro de la clase.
- El resto de atributos de clase se deben definir externamente.

```

1  class Statics {
2      static const char scChar = 'X';
3      static const int scInt = 100;
4      static const long scLong = 50;
5      static const float scFloat;
6      static const int* pInt;
7      static const int scInts[];
8      static int sInt;
9      static char sChars[];
10 };
11 const float Statics::scFloat = 5.0;
12 const int* Statics::pInt = 0;
13 const int Statics::scInts[] = {30, 15, 17, 99, 50};
14 int Statics::sInt = 10;
15 char Statics::sChars[] = "abcde";

```

# Sobrecarga de operadores

NO TOCAR LA SINTAXIS DEL OPERADOR  
LA SEMANTICA QUE SEA SIEMPRE LA MAS OBVIA  
LAS REGLAS DE PRECEDENCIA LAS MISMAS

```

1  class Matriz {
2  public:
3      Matriz(unsigned n = 1, double d = 0.0); // Matriz cte. nxn
4      // ...
5      Matriz operator +(); // unario
6      Matriz operator -(); // unario
7      Matriz operator +(const Matriz& b); // binario
8      Matriz operator -(const Matriz& b); // binario
9      Matriz operator *(const Matriz& b); // binario
10     Matriz operator *(double k); // binario
11 };

13 Matriz r{3}, a{3}, b{3};
14 unsigned x = 3;
15 r = a + b; // equivale a r = a.operator +(b);
16 r = -a * b; // equivale a r = a.operator -().operator *(b);
17 r = a * -b; // equivale a r = a.operator *(-b.operator -());
18 r = a + x; // equivale a r = a.operator +(Matriz{x});
19 r = x + a; // Error: Matriz operator +(unsigned, Matriz) no definido

```

No puedes convertir una matriz en un numero

# Sobrecarga de operadores

```

1  class Matriz {
2  public:
3      Matriz(unsigned n = 1, double d = 0.0); // Matriz cte. nxn
4      // ...
5  };
6  Matriz operator +(const Matriz& a);
7  Matriz operator -(const Matriz& a);
8  Matriz operator +(const Matriz& a, const Matriz& b);
9  Matriz operator -(const Matriz& a, const Matriz& b);
10 Matriz operator *(const Matriz& a, const Matriz& b);
11 Matriz operator *(const Matriz& a, double k); // producto escalar
12 Matriz operator *(double k, const Matriz& a); // conmutativo

14 Matriz r{3}, a{3}, b{3};
15 unsigned x = 3;
16 r = a + b; // equivale a r = operator +(a, b);
17 r = -a * b; // equivale a r = operator *(operator -(a), b);
18 r = a * -b; // equivale a r = operator *(a, operator -(b));
19 r = a + x; // equivale a r = operator +(a, Matriz{x});
20 r = x + a; // equivale a r = operator +(Matriz{x}, a);

```

# Métodos constantes

```
1  #ifndef ALUMNO_H_
2  #define ALUMNO_H_

4  #include <iostream>
5  #include <string>
6  using namespace std;

8  class Alumno {
9  public:
10     // ...
11     string& nombre();
12     const string& nombre() const;
13     unsigned long int& dni();
14     unsigned long int dni() const;
15     void mostrar() const;
16 private:
17     string nombre_;
18     unsigned long int dni_;
19     // ...
20 };
```

# Métodos constantes

```
21 // Acceso al nombre del alumno
22 inline string& Alumno::nombre() { return nombre_; }
23 inline const string& Alumno::nombre() const { return nombre_; }

25 // Acceso al DNI del alumno
26 inline unsigned long int& Alumno::dni() { return dni_; }
27 inline unsigned long int Alumno::dni() const { return dni_; }

29 // Presentación de los datos de un alumno
30 inline void Alumno::mostrar() const
31 {
32     cout << "Nombre:_" << nombre() << endl
33         << "DNI:_" << dni() << endl
34         // ...
35     ;
36 }

38 #endif
```

# Métodos constantes

```
1  // Programa de prueba de la clase Alumno

3  #include "alumno.h"

5  int main()
6  {
7      Alumno p1;
8      p1.nombre() = "Juan_□España";
9      p1.dni() = 51873029;
10     p1.mostrar();
11     const Alumno p2 = p1;
12     p2.mostrar();
13 }
```

# Clases y funciones amigas

```
1  class A {
2  public:
3      friend class B; // la clase «B» es amiga de «A»
4      // ...
5  private:
6      int d;
7      // ...
8  };

10 class B {
11 public:
12     void f(A& a) { ++a.d; } // modifica «a.d», que es privado
13     // ...
14 };
```



# Clases y funciones amigas

```
1 class Matriz {  
2 public:  
3     Matriz(unsigned n = 1, double d = 0.0); // Matriz cte. nxn  
4     friend Matriz operator +(const Matriz& a);  
5     friend Matriz operator -(const Matriz& a);  
6     friend Matriz operator +(const Matriz& a, const Matriz& b);  
7     friend Matriz operator -(const Matriz& a, const Matriz& b);  
8     friend Matriz operator *(const Matriz& a, const Matriz& b);  
9     friend Matriz operator *(const Matriz& a, double k);  
10    friend Matriz operator *(double k, const Matriz& a);  
11 };
```

# Constructores

Método	Parámetros		Descripción
	Número	Tipo	
Constructor	> 1	cualquiera	Genera un objeto nuevo.
Constructor predeterminado	0		Objeto en estado inicial por defecto.
Constructor de conversión	1	$\neq$ clase	Convierte el parámetro en un nuevo objeto.
Constructor de copia	1	clase &	Genera un clon del parámetro.
Constructor de movimiento	1	clase &&	Genera un clon del parámetro y se destruye el original.
Constructor de lista inicializadora	1	initializer_list<T>	Construye un contenedor con los elementos de la lista.

# Constructores

Un constructor es llamado:

- Cuando se define un objeto (declaración + inicialización).
- Al crear dinámicamente un objeto (con `new` y `new[]`).
- Para realizar ciertas conversiones.
- En la transferencia por valor de parámetros y resultados de funciones.
  - Copia
  - Movimiento

# Inicialización uniforme (C++11)

En C++11 se pueden usar cuatro estilos de inicialización:

```
Tipo a1 = v;      // Sintaxis tradicional de C
Tipo a2 = {v};    // Sintaxis heredada de C. Inicialización de union,
                  // struct y vectores de bajo nivel
Tipo a3(v);       // C++98. Llamada explícita al constructor
Tipo a4 {v};      // C++11. Sintaxis de inicialización uniforme.
                  // Lista de inicialización
```

El último es el estilo más recomendable porque:

- Es universal: Se puede usar en cualquier contexto en el que haya que construir un nuevo objeto o inicializar una variable.
- Es uniforme: Siempre tiene el mismo significado, se crea un valor del tipo correspondiente llamando a un constructor.
- Impide conversiones con truncamiento (de coma flotante a entero) o estrechamiento (de mayor a menor rango).
- {} asegura la inicialización por defecto incluso de tipos básicos.

# Constructor por defecto o predeterminado

- Constructor predeterminado es aquel que puede ser llamado sin parámetros.
- Si no se define ningún constructor para una clase, el compilador genera automáticamente un constructor predeterminado.
- El constructor predeterminado generado por el compilador llama a los constructores predeterminados de los miembros de la clase.

```
1 class Punto {
2     public:
3         Punto() : x_{}, y_{} {}           // constructor predeterminado
4         Punto(double x, double y) : x_{x}, y_{y} {} // constructor
5         // ...
6     private:
7         double x_, y_;
8 };
```

# Constructor por defecto o predeterminado

```

1  // Ejemplo sin constructor predeterminado

3  class Punto {
4  public:
5      Punto(double x, double y) : x_{x}, y_{y} {} // constructor
6      // ...
7  private:
8      double x_, y_;
9  };


11 // ...
12 Punto a{3., 4.}; // Bien, se llama a a.Punto(3.0, 4.0)
13 Punto* p = new Punto{0., 0.}; // Bien, en memoria dinámica
14 Punto b; // ERROR, no hay ctor. predeterminado
15 Punto* pp = new Punto; // ERROR, no hay ctor. predeterminado
16 Punto* vp = new Punto[5]; // ERROR, no hay ctor. predeterminado
17 Punto c[5]; // ERROR, no hay ctor. predeterminado

```

# Constructor de copia

- El constructor de copia recibe una referencia constante a un objeto de la clase.

```
1 class C {  
2 public:  
3   C(const C& c) { ... }  
4   // ...  
5 }
```



- Se llama implícitamente cuando:
  - Un objeto se inicializa con otro de la misma clase.
  - Un objeto se pasa por valor.
  - Un objeto se devuelve por valor.

# Ejemplos de uso del constructor de copia

```
1 class X;           // declaración , definición en otro sitio
2 X f(X);           // declaración , definición en otro sitio

4 X a;              // definición con ctor. predeterminado
5 X b(a);           // definición con ctor. de copia
6 X c = a;          // definición con ctor. de copia
7 X d{a};           // definición con ctor. de copia
8 f(b);             // paso de b por valor y devolución por valor
9                  // de un objeto anónimo temporal con ctor de copia
10 X e = f(c);       // paso de c por valor ,
11                  // devolución por valor de un objeto anónimo temporal
12                  // y definición con ctor. de copia
13 X g(f(c));        // ídem.
14 X h{f(c)};        // ídem.
```



# Constructor de copia

```
1 // Clase Persona, versión sin copia definida
2 #include <cstring>

4 class Persona {
5     char* nombre_;
6     int edad_, estatura_;
7 public:
8     Persona(const char* n, int ed, int est)
9         : edad_{ed}, estatura_{est}
10    {
11        nombre_ = new char[std::strlen(n) + 1];
12        std::strcpy(nombre_, n);
13    }
14    ~Persona() { delete[] nombre_; }
15    void estatura(int e) { estatura_ = e; }
16    int estatura() const { return estatura_; }
17    void edad(int e) { edad_ = e; }
18    int edad() const { return edad_; }
19    char* nombre() const { return nombre_; }
20 };
```

# Constructor de copia

```
1  // Prueba de la clase Persona sin copia definida
2  #include "persona1.h"
3  #include <iostream>
4  #include <locale>
5  using namespace std;

6
7  void ver_datos(Persona p) {
8      cout << p.nombre() << " tiene " << p.edad() << " años y mide "
9          << p.estatura() / 100.0 << " m." << endl;
10 }

11
12 int main()
13 {
14     if (!setlocale(LC_NUMERIC, "")) cerr << "setlocale():error\n";
15     Persona pp{"Pepe", 32, 180};
16     ver_datos(pp);
17     cout << 'i' << pp.nombre() << " ha sido destruido!" << endl;
18 }
```

# Constructor de copia

## Salida del programa

Pepe tiene 32 años y mide 1,8 m.  
¡H&@H&@re/l ha sido destruido!



# Copia de objetos

```
1 // Clase Persona, versión con copia definida
2 #include <cstring>

4 class Persona {
5     char* nombre_;
6     int edad_, estatura_;
7 public:
8     Persona(const char* n, int ed, int est)
9         : edad_{ed}, estatura_{est}
10    {
11        nombre_ = new char[std::strlen(n) + 1];
12        std::strcpy(nombre_, n);
13    }
14    Persona(const Persona& otra)
15        : edad_{otra.edad_}, estatura_{otra.estatura_}
16    {
17        nombre_ = new char[std::strlen(otra.nombre_) + 1];
18        std::strcpy(nombre_, otra.nombre_);
19    }
```

# Copia de objetos

```
20  Persona& operator =(const Persona& otra)
21  {
22      if (this != &otra) {
23          delete[] nombre_;
24          edad_ = otra.edad_;
25          estatura_ = otra.estatura_;
26          nombre_ = new char[std::strlen(otra.nombre_) + 1];
27          std::strcpy(nombre_, otra.nombre_);
28      }
29      return *this;
30  }
31  void estatura(int e) { estatura_ = e; }
32  int estatura() const { return estatura_; }
33  void edad(int e) { edad_ = e; }
34  int edad() const { return edad_; }
35  char* nombre() const { return nombre_; }
36  ~Persona() { delete[] nombre_; }
37  };
```

# Constructor de movimiento (C++11)

- El **constructor de movimiento** recibe una **referencia a un valor-r** del tipo de la clase.

```
1 class C {  
2 public:  
3     C(C&& c) { ... }  
4     // ...  
5 }
```

- Se llama implícitamente cuando un objeto se inicializa con una **referencia a valor-r** que es otro objeto de la misma clase.

**valor-r** o derecho (en contraposición a *valor-l* o izquierdo) es el que no puede aparecer a la izquierda de una asignación. Por ejemplo, un literal o el resultado de una función devuelto por valor.

**&&** denota una referencia ligada a un valor-r.

# Movimiento de objetos (C++11)

```
1 // clase Vector sin movimiento de objetos
2 class Vector {
3     double* eltos;
4     int n_eltos;
5 public:
6     Vector(int n);
7     Vector(const Vector& v);           // constructor de copia
8     Vector& operator =(const Vector& v); // asignación
9     double& operator [](int i);
10    const double& operator [](int i) const;
11    int tam() const;
12    ~Vector();                         // destructor
13 };
```

# Movimiento de objetos (C++11)

```
14 Vector operator +(const Vector& v1, const Vector& v2)
15 {
16     Vector w{v1.tam()};
17     for (int i = 0; i < v1.tam(); ++i)
18         w[i] = v1[i] + v2[i];
19     return w; // devuelve una copia de w y se destruye w
20 }
```

```
1 Vector r{10000}, s{10000}, t{10000}, v{10000};
2 //... Se rellenan r, s y t con valores
3 v = r + s + t; // tres copias de objetos Vector
4               // temporales anónimos:
5               // una por cada operador +
6               // más una asignación
```



# Movimiento de objetos (C++11)

```
1 class Vector {
2     double* eltos;
3     int n_eltos;
4 public:
5     // ...
6     Vector(const Vector& v);           // constructor de copia
7     Vector& operator =(const Vector& v); // asignación de copia
8     Vector(Vector&& v);                // constructor de movimiento
9     Vector& operator =(Vector&& v);    // asignación de movimiento
10    // ...
11 };

13 Vector operator +(const Vector& v1, const Vector& v2)
14 {
15     Vector w{v1.tam()};
16     for (int i = 0; i < v1.tam(); ++i)
17         w[i] = v1[i] + v2[i];
18     return w; // mueve w a un temporal anónimo y se destruye w
19 }
```

# Movimiento de objetos (C++11)

```

1  // Constructor de movimiento
2  Vector::Vector(Vector&& v) // parámetro no-const
3      : eltos{v.eltos}, n_eltos{v.n_eltos}
4  {
5      v.eltos = nullptr; // Tras el movimiento se invocará
6      v.n_eltos = 0;      // al destructor de v.
7  }

9  // Asignación de movimiento
10 Vector& Vector::operator=(Vector&& v) // parámetro no-const
11 {
12     if (this != &v) {
13         delete[] eltos;
14         eltos = v.eltos;
15         n_eltos = v.n_eltos;
16         v.eltos = nullptr; // Tras el movimiento se invocará
17         v.n_eltos = 0;    // al destructor de v.
18     }
19     return *this;
20 }

```

Después de que se utilice el vector dcho v, se elimina.

# Movimiento de objetos (C++11)

- Para forzar el movimiento de un *valor-l* se utiliza la función de la biblioteca estándar move(), declarada en la cabecera `<utility>`

```
1 Vector f()
2 {
3     Vector x{10000}, y{10000}, z{10000};
4     // ...
5     z = x;           // asignación de copia
6     y = std::move(x); // conversión de x en Vector&& y asig. de mov.
7     // ...
8     return z;        // constructor de movimiento
9 }
```

# Constructor de conversión

- El **constructor de conversión** recibe un parámetro de un tipo distinto al de la clase.

```
1 class C {  
2 public:  
3     C(const D& d) { ... }  
4     // ...  
5 }
```

en C: (double)y  
en C++: double(y)

- Se llama cuando se realiza un modelado explícito o, implícitamente, cuando es necesaria una conversión de tipo.

# Ejemplo de constructor de conversión

```

1 class Punto {
2 public:
3     Punto(double x = 0.0, double y = 0.0): x_{x}, y_{y} {}
4     // ...
5 };

```

- Conversión de `double` a `Punto`

```

1 Punto w = 2.5; // conv. implícita, crea el punto (2.5, 0.0)
2 Punto x = Punto(2.5); // crea el punto (2.5, 0.0)
3 Punto y(2.5);          // crea el punto (2.5, 0.0)
4 Punto z{2.5};          // crea el punto (2.5, 0.0)

```

- Conversión de `double` a `Punto` en asignaciones

```

1 w = -0.25; // conversión implícita
2 x = Punto(-0.25);
3 y = Punto{-0.25};
4 z = static_cast<Punto>(-0.25);

```

# Ejemplo de constructor de conversión

```
1 class Punto {  
2     public:  
3         // Evitar conversiones implícitas  
4         explicit Punto(double x = 0.0, double y = 0.0): x_{x}, y_{y} {}  
5         // ...  
6 };
```

- Entonces no se podría hacer

```
1 Punto w = 2.5;           // ERROR, conversión implícita prohibida  
2 w = -0.25;              // ERROR, conversión implícita prohibida
```

- pero sí

```
3 Punto x = Punto(2.5);  
4 Punto y(2.5);  
5 y = Punto{-0.25};  
6 z = static_cast<Punto>(-0.25);
```

# Asignación con conversión

```

1  class Complejo; // definida en otro sitio
2  class Punto {
3      double x_, y_;
4  public:
5      Punto(double x = 0., double y = 0.): x_{x}, y_{y} {}
6      Punto(const Punto& o): x_{o.x_}, y_{o.y_} {}
7      Punto(const Complejo& c): x_{c.real()}, y_{c.imag()} {}
8      Punto& operator =(const Punto& o)
9          { x_ = o.x_; y_ = o.y_; return *this; }
10     Punto& operator =(const Complejo& c)
11         { x_ = c.real(); y_ = c.imag(); return *this; }
12     // ...
13 };

15 Complejo c1, c2;
16 Punto p1 = c1, // llamada a Punto(const Complejo&)
17     p2;
18 p2 = c2; // llamada a Punto& operator =(const Complejo&)

```

# Operadores de conversión

```

1  class Punto {
2  public:
3      // ...
4      operator Complejo() {
5          return {x_, y_}; // Equivale a return Complejo{x_, y_};
6          // objeto temporal creado con ctor. de Complejo que devuelve por valor
7      }
8  };

10 void f(const Complejo& z);
11 Punto p;
12 f(p); // Equivale a f(p.operator Complejo());

```

Conversion para que se pueda realizar la funcion f con un parametro complejo

## ● Problema de ambigüedad

```

1  class Complejo {
2  public:
3      Complejo(const Punto&); // ctor. de conversión
4      // ...
5  };

```



# Constructor de lista inicializadora (C++11)

- Recibe un objeto de la clase de la biblioteca estándar `initializer_list<T>`, definida en la cabecera del mismo nombre.

```
1 #include <initializer_list>

3 class C {
4 public:
5     C(std::initializer_list<T> li);
6     // ...
7 }
```

- Los objetos de la clase `initializer_list<T>` los crea automáticamente el compilador a partir de una lista de inicializadores (lista de valores separados por comas y encerrada entre `{ }`).
- Proporciona sintaxis uniforme de inicialización (`c{...}`) a los contenedores (colecciones de objetos del mismo tipo).

# Constructor de lista inicializadora (C++11)

```
1 #include <initializer_list>
2 #include <algorithm>

4 class Vector {
5 public:
6     // ...
7     Vector(int n, double e = 0.0); // n eltos. de valor e
8     Vector(std::initializer_list<double>);
9     // ...
10 private:
11     double* eltos;
12     int n_eltos;
13 };

15 Vector::Vector(std::initializer_list<double> l) :
16     eltos{new double[l.size()]}, n_eltos{l.size()}
17 {
18     std::copy(l.begin(), l.end(), eltos);
19 }
```

# Constructor de lista inicializadora (C++11)

- Ahora podemos definir objetos de tipo Vector a partir de una lista de valores.

```
1 Vector v = {1, 2, 3, 4, 5}; // Al estilo de los vectores en C
2 Vector w{1.8, 4.5, 3., 9.7, 10.}; // C++11. Inicialización uniforme
```

- Al usar sintaxis de inicialización uniforme el constructor de lista inicializadora tiene precedencia sobre otros constructores, salvo sobre el predeterminado.

```
1 Vector v1{}; // ctor. predeterminado, si existe
2 Vector v2{100}; // un elto. de valor 100
3 Vector v3{100, 3.5}; // dos eltos. de valor 100 y 3.5
4 Vector v4(100); // ctor. de conversión: 100 eltos de valor 0.0
5 Vector v5(100, 3.5); // 100 eltos de valor 3.5
```

# Destructor

- A veces la creación de un objeto implica la adquisición de recursos (como p.e., un fichero, o un bloque de memoria) que deben ser liberados después de ser utilizados.
- Las clases de estos objetos necesitan de un método que se ocupe de liberar estos recursos justo antes de que uno de sus objetos sea destruido. Este método es el destructor.
- Al destructor se llama implícitamente cuando:
  - Termina el bloque donde ha sido definido un objeto automático.
  - Termina el programa normalmente mediante `return` o `exit()`.
  - Se elimina un objeto creado en memoria dinámica con los operadores `delete` o `delete[]`.
- En raras ocasiones es necesario llamar al destructor explícitamente.