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 Aburruzaga García

UCA
Universidad de Cádiz

Versión 2.0

Índice

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.

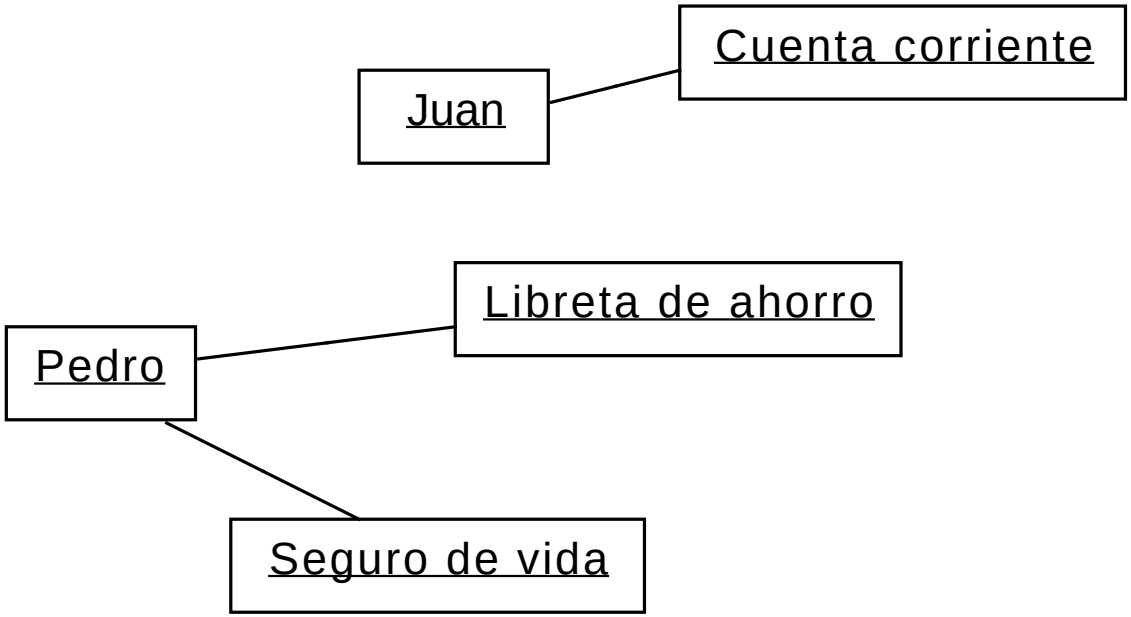
Estado Conjunto de valores de todos los atributos de un objeto en un instante. Tiene un carácter dinámico, evoluciona con el tiempo.

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

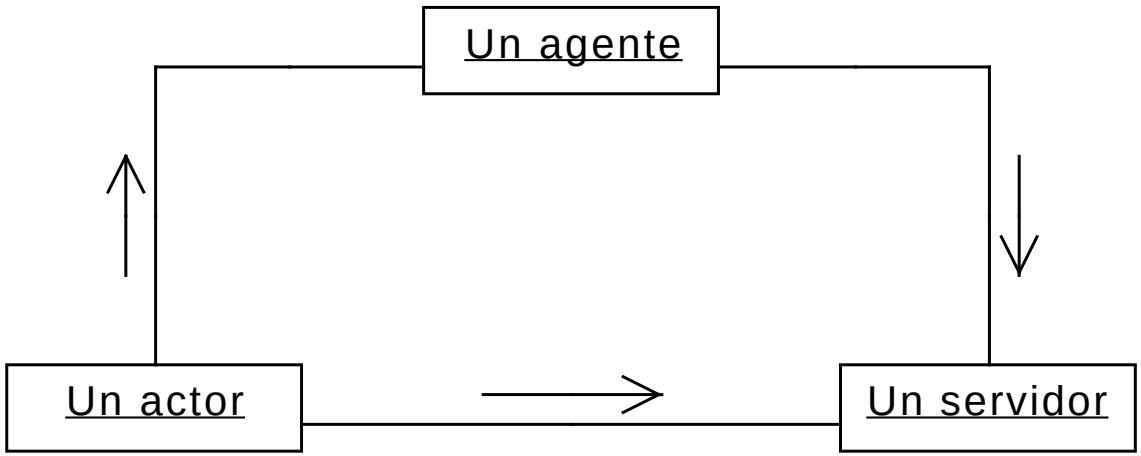


Objetos

Actores Objetos que exclusivamente emiten mensajes

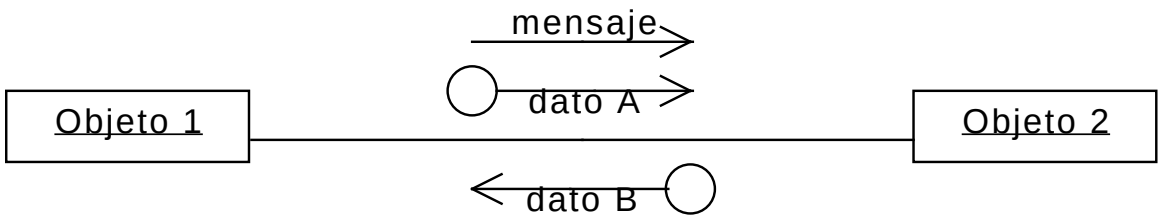
Servidores Objetos que únicamente reciben mensajes

Agentes Pueden emitir y recibir mensajes



Mensajes

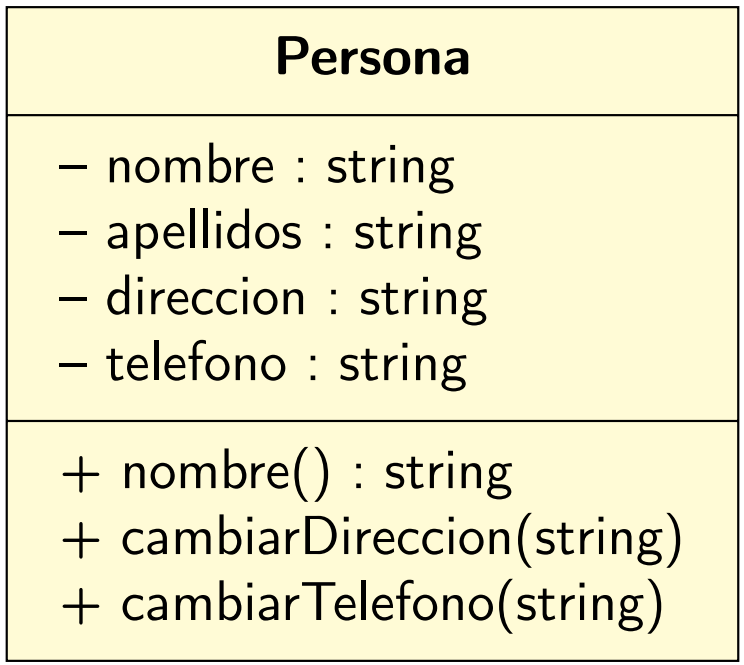
nombre[destino, operación, parámetros]



Clases

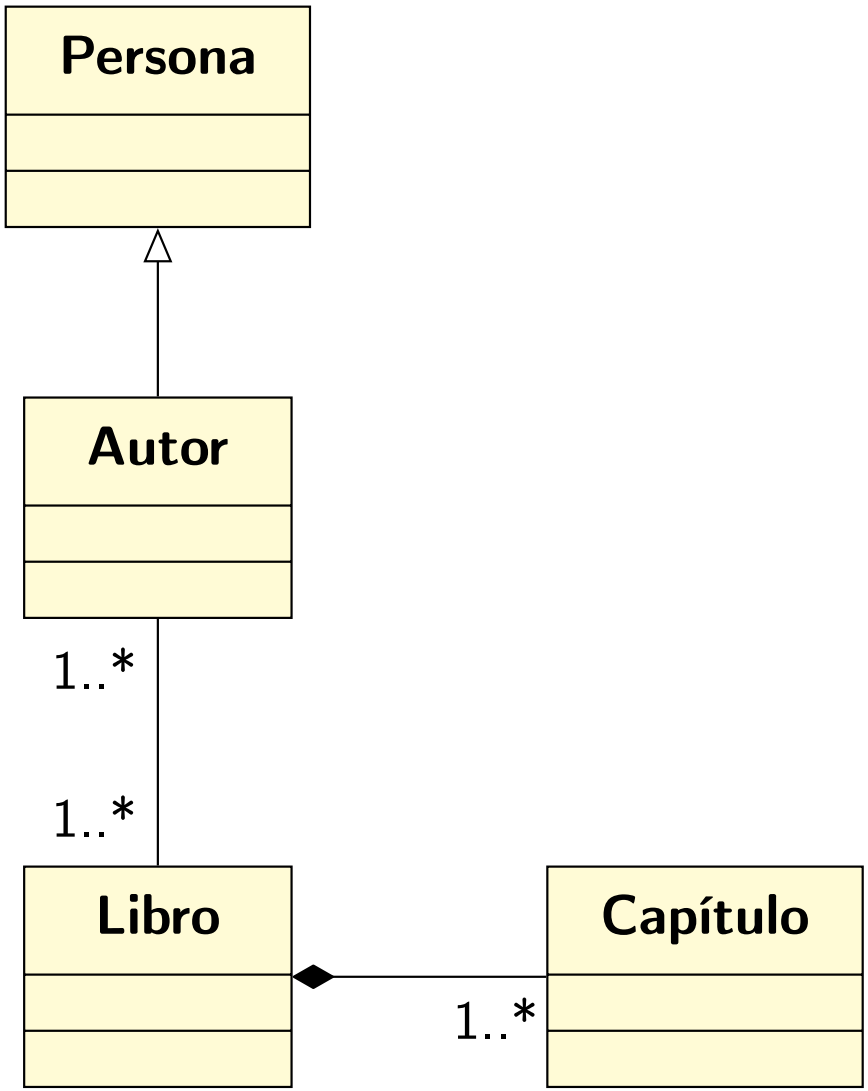
Una clase

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



Índice

Estructura de una clase

- Una clase encapsula el conjunto de miembros que la componen:
  - Atributos (datos): definen la representación de los objetos.
  - Métodos (funciones): definen las operaciones que se pueden realizar con los atributos.
  - Tipos auxiliares o relacionados con la clase.
- Con las palabras reservadas `public` y `private` se controla el acceso a cada uno de los miembros.

```
1 class C {
2 public:
3   // Miembros visibles desde el exterior (interfaz de la clase).
4   // Generalmente, métodos y tipos, pero no atributos.
5 private:
6   // Miembros ocultos al exterior (detalles de implementación de la clase).
7   // Generalmente, atributos y, a veces, métodos auxiliares
8   // y tipos internos.
9 };
```

Estructura de una clase

- Las palabras reservadas `class` y `struct` son sinónimas, aunque existen dos diferencias entre ellas: por omisión, los miembros de `class` son privados y los de `struct` públicos; la otra diferencia está relacionada con la herencia (se verá más adelante). Por tanto, los tres códigos de cada fila de la tabla son equivalentes.

```
1 class C {
2   //...
3 };
```

```
1 class C {
2 private:
3   //...
4 };
```

```
1 struct C {
2 private:
3   //...
4 };
```

```
1 struct C {
2   //...
3 };
```

```
1 struct C {
2 public:
3   //...
4 };
```

```
1 class C {
2 public:
3   //...
4 };
```

Métodos inline

- Un método definido en el cuerpo de la clase, aprovechando la declaración del propio método, es implícitamente `inline` y equivale a su definición en línea en el exterior de la clase.

```
1  class Entero {
2  public:
3    int cambiar(int x)
4    {
5      int ant = i;
6      i = x;
7      return ant;
8    }
9  private:
10   int i;
11 };
```

```
1  class Entero {
2  public:
3    int cambiar(int x);
4  private:
5    int i;
6  };

8  inline int Entero::cambiar(int x)
9  {
10   int ant = i;
11   i = x;
12   return ant;
13 }
```

El puntero this

- Un método (no estático) recibe la dirección del objeto que lo llama a través de un parámetro implícito de nombre `this`.
- En un método de la clase `X`, `this` es de tipo `X*` (puntero a `X`) y, además, no es posible cambiarlo para que apunte a otro objeto.
- En la definición de un método se puede añadir `this->` delante del nombre de cada miembro al que se acceda, pero casi nunca es necesario porque ya lo hace automáticamente el compilador.

```
1 class X {
2   int m, n;
3 public:
4   void fun(int n) {      // void fun(X* const this, int n) // ver nota
5     if (n % 2) m = n;    // m es this->m, n es el parámetro
6     this->n = n;         // this-> necesario para acceder al atributo n
7   }
8 };
```

```
1 X x;
2 x.fun(7); // X::fun(&x, 7); // nota: código ficticio que explica el significado
```

Miembros constantes

- El valor de un atributo declarado constante (`const`) no puede ser modificado después de inicializado.
- Los atributos de un objeto constante se tratan como si fueran constantes, aunque no se hayan declarado `const`, a excepción de los declarados con el calificativo `mutable`.
- Son métodos observadores los que informan del estado de un objeto sin alterarlo y se distinguen con la palabra `const` tras la lista de parámetros (tanto en la declaración como en la definición). Dicho `const` se aplica al objeto apuntado por el parámetro implícito `this`, como si fuera declarado `const X* const this` para un método de la clase `X`.
- Consecuentemente, un método `const` no puede modificar el estado del objeto al que se aplica, exceptuando los atributos declarados `mutable`.

Miembros constantes

```
1  class Reloj {
2  public:
3    // ...
4    void incrementarHoras()     // void incrementarHoras(Reloj* const this)
5    { h_ = (h_ + 1) % 24; }
6    void incrementarMinutos()   // void incrementarMinutos(Reloj* const this)
7    {
8      if ((m_ = (m_ + 1) % 60) == 0)
9        incrementarHoras();
10   }
11   int horas() const           // int horas(const Reloj* const this)
12   { return h_; }
13   int minutos() const         // int minutos(const Reloj* const this)
14   { return m_; }
15 private:
16   int h_, m_;
17 // Nota: el código de los comentarios no es válido, sino que
18 // muestra cómo interpreta el compilador el código real.
19 };
```

Atributos mutables

- Un atributo `mutable` siempre se puede modificar, aun cuando el objeto al que pertenezca sea constante.
- Un atributo mutable no afecta al estado de los objetos visible desde el exterior.

```
1  class Semaforo {
2    enum Color { ROJO, VERDE, AMBAR };
3  public:
4    // ...
5    void comprobar() const { revisado = true; } // No cambia el color c
6    void cambiar() { c = Color((c + 1) % 3); }
7  private:
8    Color c;
9    mutable bool revisado; // No afecta al estado lógico (color)
10 };
```

Sobrecarga de métodos constantes

- Un objeto `const` no puede llamar a un método no-`const` (potencialmente modificador del estado interno del objeto), porque un objeto `const` de tipo `X` no puede pasarse como el parámetro implícito de tipo `X*` (puntero a un `X` no-`const`).

```
1  class Alumno {
2  public:
3    // ...
4    string& nombre() { return nombre_; } // Lec/Esc nombre_ de Alumno no-const
5    unsigned long& dni() { return dni_; } // Lec/Esc dni_ de Alumno no-const
6    void mostrar() const {
7      cout << "Nombre: " << nombre() << endl // Error: *this es const mientras que
8           << "DNI: " << dni() << endl;      // this->nombre() y this->dni() no.
9    }
10 private:
11   string nombre_;
12   unsigned long dni_;
13   // ...
14 };
```

Sobrecarga de métodos constantes

```
1  #ifndef ALUMNO_H_
2  #define ALUMNO_H_
3  #include <iostream>
4  #include <string>
5  using namespace std;

7  class Alumno {
8  public:
9    // ...
10   string& nombre() { return nombre_; }                // Lec./Esc. de Alumno
11   const string& nombre() const { return nombre_; } // Lec. de const Alumno
12   unsigned long& dni() { return dni_; }               // Lec./Esc. de Alumno
13   unsigned long dni() const { return dni_; }        // Lec. de const Alumno
14   void mostrar() const {
15     cout << "Nombre: " << nombre() << endl // Bien: Alumno::nombre() const
16     << "DNI: " << dni() << endl;          //      Alumno::dni() const
17   }
18 private:
19   string nombre_;
20   unsigned long dni_;
21 };

23 #endif   // ALUMNO_H_
```

Sobrecarga de métodos constantes

```
1  // Programa de prueba de la clase Alumno
2
3  #include "alumno.h"
4
5  int main()
6  {
7    Alumno p1;
8    p1.nombre() = "Juan España"; // Alumno::nombre()
9    p1.dni() = 51873029;         // Alumno::dni()
10   p1.mostrar();                // Alumno::mostrar() const
11   const Alumno p2 = p1;
12   string np2(p2.nombre());     // Alumno::nombre() const
13   p2.mostrar();                // Alumno::mostrar() const
14   p2.dni() = 35925037;   // Alumno::dni() const
15                          // Error: operando izqdo. de asig. no puede
16                          // modificarse, porque es una copia temporal
17                          // de p2.dni_ que se destruirá inmediatamente.
18 }
```

Miembros estáticos

- Un *atributo de clase* representa una propiedad de la clase, no de los objetos individuales, y se declara con la palabra reservada `static`. A diferencia de un atributo ordinario, almacenado en la memoria asignada a cada objeto, un atributo estático es compartido por todos los objetos de la clase.
- Un método estático, declarado también con la palabra `static`, es una función auxiliar de una clase que no necesita operar con los objetos individuales. Por esta razón, no recibe el parámetro implícito `this` y no tiene acceso a los miembros no estáticos.
- Para referirse a un miembro estático `m` de la clase `X` puede usarse una expresión de selección de miembro (igual que si no fuera estático), `E.m` o `E->m`, donde `E` es, a su vez, una expresión de tipo `X` o `X*`, respectivamente. Pero, este estilo oculta que en realidad se trata de un miembro estático, por lo que es preferible usar el operador de resolución de ámbito: `X::m`.

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
```

Miembros estáticos

```
1  // entrada.cpp
2
3  #include <iostream>
4  #include <iomanip>    // fixed, setprecision
5  #include "entrada.h"
6  using namespace std;
7
8  // El núm. inicial de asiento será el 1 para las cuatro sesiones
9  int Entrada::proximo_asiento[4] = { 1, 1, 1, 1 };
10
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 }
```

Miembros estáticos

```
19 // Impresión de una entrada.
20 void Entrada::imprimir() const
21 {
22   cout << "Sesión " << sesion + 1
23        << "\tAsiento Nº " << 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

- La mayoría de los operadores de C++ pueden ser sobrecargados para que al menos uno de sus operandos sea de un tipo definido por el programador (clase o enumeración).
- No se puede cambiar la sintaxis (número y orden de los operandos), la precedencia ni la asociación (por la izquierda o derecha) de un operador. Tampoco es posible crear nuevos operadores, como `$`, `@`, `**`, `<>`, ...
- No se debe cambiar la semántica de los operadores o ser incoherente definiendo los relacionados (p.e., `++x`, `x += 1` y `x = x + 1` deberían ser equivalentes).
- Hay operadores que no se pueden sobrecargar, como `::` (resolución de ámbito), `.` (selección de miembro), `.*` (selección de miembro a través de puntero a miembro), `?:` (evaluación condicional), `sizeof` (tamaño de un tipo u objeto), `typeid` (información de un tipo), etc.

Sobrecarga de operadores

- Salvo excepciones, los operadores sobrecargables pueden ser:
  - Miembros no estáticos de la clase de su primer operando. Entonces, se declaran con un parámetro de menos, pues su primer operando lo reciben por el parámetro implícito `this`.
  - Funciones que no pertenecen a ninguna clase. Todos sus operandos se declaran como parámetros.
- Las reglas gramaticales de C++ obligan a que sean miembros no estáticos los siguientes operadores: = (asignación), `()` (llamada a función), `[]` (indexación) y -> (selección de miembro a través de puntero a objeto).
- Ciertos operadores deben ser no miembros porque el tipo de su primer operando no es una clase o, si lo es, no se le pueden añadir nuevos miembros, como es el caso de << (inserción en flujo) y >> (extracción de flujo), ya que no pueden ser incluidos en las clases estándar `std::ostream` y `std::istream`, respectivamente.

Sobrecarga de operadores

- Los operadores que modifican el estado de su primer operando es preferible que sean miembros (p.e. *= y ++), ya que su implementación suele ser más eficiente accediendo a la representación interna de la clase.
- En cambio, si se desean conversiones implícitas sobre todos los operandos, el operador debe ser una función externa a la clase (y amiga si hace falta acceder a la representación). Normalmente, este es el caso de los operadores binarios aritméticos (+, /, &, ^, ...), de comparación (==, <, ...) y lógicos (&&, ||) y, con menos frecuencia, también es el caso de los unarios aritméticos (+, -, ~) y lógico (!).

Sobrecarga de operadores

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

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

Sobrecarga de operadores

```
1  class Matriz {
2  public:
3    Matriz(unsigned n = 1, double d = 0.0); // Matriz cte. n × n
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);
```

Clases y funciones amigas

- Una función amiga de una clase no es miembro de ella, pero tiene acceso a su implementación al igual que las funciones miembro.
- Una función no miembro se hace amiga incluyendo su declaración en el cuerpo de la clase, igual que si fuera miembro, pero anteponiendo la palabra `friend`. La declaración de amistad puede ser indistintamente pública o privada, tiene el mismo efecto.
- Declarar una clase amiga de otra es equivalente a declarar amigos de esta todos los métodos de la primera.
- La declaración de clases y funciones amigas va en contra del **principio de ocultación de la implementación** de una clase, razón por la cual debe usarse exclusivamente en casos bien justificados.

Clases y funciones amigas

```
1  class A {
2  public:
3    friend int f(const A&); // La función f es amiga de la clase A.
4    friend class B;         // La clase B es amiga de A.
5    // ...
6  private:
7    int d;
8    // ...
9  };

11 int f(const A& a) { return a.d; } // a.d es privado, pero f es amiga de A.

13 class B {
14 public:
15   void f(A& a) { ++a.d; } // a.d es privado, pero B::f es amiga de A
16   // ...
17 };
```

Clases y funciones amigas

- Un caso común de uso de amistad es cuando interesa la sobrecarga externa de un operador y es necesario acceder a la implementación de la clase, por cuestión de eficiencia o porque la interfaz de la clase no permite el acceso a todos los miembros privados.

```
1  class Matriz {
2  public:
3    Matriz(unsigned n = 1, double d = 0.0); // Matriz cte. n × n
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 };
```

Índice

Constructores

Método	Parámetros Número	Parámetros Tipo	Descripción
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 sin modificar el original.
Constructor de movimiento	1	clase &&	Genera un clon «vaciando» el original.
Constructor de lista inicializadora	1	initializer_list<T>	Construye un contenedor con los elementos de la lista.

Constructores

- El nombre de un constructor coincide con el de su clase (`C::C` es el nombre completo para una clase llamada `C`).
- No tienen tipo de devolución, ni siquiera `void`.
- A diferencia de otros métodos, pueden tener lista de inicialización de atributos, que es procesada justo antes de ejecutar el cuerpo de la función. Si no se especifica, el valor inicial de cada atributo es el predeterminado, si está definido.
- Un constructor no puede ser llamado directamente, se llama siempre automáticamente al crear un nuevo objeto para establecer su estado inicial:
  - Cuando se define un objeto (declaración + inicialización).
  - Al crear un objeto en memoria dinámica (con new y new[]).
  - Cuando se requiere una conversión del tipo del parámetro al tipo de la clase.
  - En la transferencia por valor de parámetros y resultados de funciones (copia y movimiento).

Inicialización uniforme (C++11)

Desde 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 estilo de C++11 es el más recomendable porque:

- Es uniforme. Siempre tiene el mismo significado: crear un valor u objeto del tipo declarado determinado por el inicializador.
- Es universal: Se puede usar en cualquier contexto en el que haya que construir un nuevo objeto o inicializar una variable.
- Impide conversiones peligrosas 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 por omisión, que tiene el mismo efecto que si el programador define uno como `C::C() {}` para la clase `C`: llama al constructor predeterminado de cada atributo cuyo tipo sea una clase y deja con valores indeterminados los de tipos básicos.

```
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
2
3  class Punto {
4  public:
5    Punto(double x, double y) : x_{x}, y_{y} {} // constructor
6    // ...
7  private:
8    double x_, y_;
9  };
10
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 un objeto de la clase por referencia (a constante, normalmente, pues no se necesita modificar).

```
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 a una función.
  - Un objeto se devuelve por valor desde una función (si no existe constructor de movimiento).

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 por defecto

```
1  // Clase Persona, versión sin copia definida por el programador
2  #include <cstring>    // strlen, strcpy, strcmp

4  class Persona {
5    char* nombre_;
6    int edad_, estatura_;
7  public:
8    Persona(const char* n, int ed, int est) :
9      nombre_{new char[std::strlen(n) + 1]},
10     edad_{ed}, estatura_{est}
11   {
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 por defecto

```
1  // Prueba de la clase Persona sin copia definida por el programador
2  #include <iostream>

4  void ver_datos(Persona p) {
5    std::cout << p.nombre() << " tiene " << p.edad() << " años "
6              << "y mide " << p.estatura() / 100.0 << " m.\n";
7  }

9  int main()
10 {
11   Persona pp{"Pepe", 32, 180};
12   ver_datos(pp);
13   std::cout << "¡" << pp.nombre()
14             << (std::strcmp(pp.nombre(), "Pepe") ? "" : " NO")
15             << " ha sido destruido!\n\n";
16 }
```

Constructor de copia por defecto

Salida del programa

```
Pepe tiene 32 años y mide 1.8 m.
¡H\&@H\&@re/l ha sido destruido!

free(): double free detected...
Abortado ('core' generado)
```

Constructor de copia por defecto

- El compilador genera un constructor de copia por defecto si el programador no define ninguno de los métodos siguientes:
  - constructor de copia
  - constructor de movimiento (C++11)
  - operador de asignación de movimiento (C++11).

  Además, a partir de C++11 está desaconsejada la generación del constructor de copia por defecto, si el programador define el operador de asignación de copia o el destructor, pero depende del compilador (en el ejemplo anterior se puede ver que no ha seguido esta recomendación).
- Este constructor por defecto copia uno por uno los atributos de la clase siguiendo el orden en que hayan sido declarados. Los atributos cuyo tipo sea una clase se copian mediante sus propios constructores de copia.

Operador de asignación de copia

- El operador de asignación de copia recibe un objeto de la clase (operando derecho) para copiarlo en otro preexistente (operando izquierdo) que devuelve por referencia.

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

- Es definido implícitamente por el compilador si el programador no lo define y tampoco define constructor/asignación de movimiento. A partir de C++11, también está desaconsejada esta definición implícita si se define expresamente el constructor de copia o el destructor, pero depende del compilador.
- El operador de asignación de copia por defecto asigna uno a uno los atributos de la clase, usando el operador de asignación de cada atributo.

Copia de objetos

```
1  // Clase Persona, versión con copia definida por el programador
2  #include <cstring>    // strlen, strcpy, strcmp

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

Copia de objetos

```
20    Persona& operator =(const Persona& otra)
21    {
22      if (this != &otra) {    // Evitar autoasignación destructiva
23        char* n = nombre_;
24        nombre_ = new char[std::strlen(otra.nombre_) + 1];
25        std::strcpy(nombre_, otra.nombre_);
26        edad_ = otra.edad_;
27        estatura_ = otra.estatura_;
28        delete[] n;
29      }
30      return *this;
31    }
32    ~Persona() { delete[] nombre_; }
33    void estatura(int e) { estatura_ = e; }
34    int estatura() const { return estatura_; }
35    void edad(int e) { edad_ = e; }
36    int edad() const { return edad_; }
37    char* nombre() const { return nombre_; }
38  };
```

Copia de objetos

Salida del programa de prueba

```
Pepe tiene 32 años y mide 1.8 m.
¡Pepe NO ha sido destruido!
```

Constructor de movimiento (C++11)

- El constructor de movimiento recibe un objeto de la clase por referencia a valor-r (no constante, ya que por lo general modifica su estado).

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

- Es llamado implícitamente en los mismos casos que el constructor de copia cuando el parámetro es un valor-r.

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.

Constructor de movimiento por defecto

- El compilador genera un constructor de movimiento por defecto si el programador no define ninguno de los métodos siguientes:
  - constructor de copia
  - operador de asignación de copia
  - constructor de movimiento
  - operador de asignación de movimiento
  - destructor
- Este constructor de movimiento por defecto mueve uno por uno los atributos de la clase siguiendo el orden en que hayan sido declarados. Los atributos cuyo tipo sea una clase se mueven con sus propios constructores de movimiento, si están disponibles, o de copia, si no. Los atributos de tipos básicos se copian.

Operador de asignación de movimiento (C++11)

- El operador de asignación de movimiento recibe un objeto de la clase (operando derecho) por referencia a valor-r para moverlo a otro preexistente (operando izquierdo) que devuelve por referencia.

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

- Es definido implícitamente por el compilador si el programador no lo define y tampoco define constructor/asignación de copia, constructor de movimiento ni destructor.
- El operador de asignación de movimiento por defecto asigna uno a uno los atributos de la clase, usando la asignación de movimiento del tipo de cada atributo. Los atributos de tipos básicos se copian.

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) : eltos{new double[n]}, n_eltos{n} {}
7    Vector(const Vector& v);                    // constructor de copia
8    Vector& operator =(const Vector& v);        // asignación de copia
9    double& operator [](int i);
10   const double& operator [](int i) const;
11   int tam() const;
12   ~Vector() { delete[] eltos; }               // 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 temporales anónimos:
4                 // una por cada operador + más una asignación
```

Movimiento de objetos (C++11)

```
1  // Clase Vector con movimiento de objetos
2  class Vector {
3    double* eltos;
4    int n_eltos;
5  public:
6    Vector(int n) : eltos{new double[n]}, n_eltos{n} {}
7    Vector(const Vector& v);                  // constructor de copia
8    Vector& operator =(const Vector& v);      // asignación de copia
9    Vector(Vector&& v);                       // constructor de movimiento
10   Vector& operator =(Vector&& v);           // asignación de movimiento
11   // ...
12 };

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;  // mueve w a un temporal anónimo y se destruye w
20 }
```

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;    // Estado válido para
6    v.n_eltos = 0;        // destructor y asignación.
7  }

9  // Asignación de movimiento
10 Vector& Vector::operator =(Vector&& v) // par. no-const
11 {
12   if (this != &v) {       // Evitar autoasignación destructiva
13     delete[] eltos;
14     eltos = v.eltos;
15     n_eltos = v.n_eltos;
16     v.eltos = nullptr;    // Estado válido para
17     v.n_eltos = 0;        // destructor y asignación.
18   }
19   return *this;
20 }
```

Movimiento de objetos (C++11)

```
1  // Alternativa segura frente a autoasignación
2  Vector& Vector::operator =(Vector&& v) noexcept
3  {
4    double* t = v.eltos;
5    v.eltos = nullptr;    // En autoasig. eltos también es nulo
6    delete[] eltos;       // Inoperante en autoasignación
7    eltos = t;            // En autoasig. restaura v.eltos
8    int n = v.n_eltos;
9    v.n_eltos = 0;
10   n_eltos = n;          // En autoasig. restaura v.n_eltos
11   return *this;
12 }
```

Movimiento de objetos (C++11)

- La resolución de sobrecarga de las operaciones de copia y movimiento selecciona constructor/asignación de copia si el parámetro es valor-l y constructor/asignación de movimiento (más eficientes) si el parámetro es valor-r.
- La devolución por valor del resultado de una función se realiza mediante el constructor de movimiento, aprovechando que el objeto que se devuelve será destruido inmediatamente. Si no está definido, se utiliza el constructor de copia.
- El movimiento de un valor-l se puede forzar con la función de la biblioteca estándar `move()` (cabecera `<utility>`).

```
1  Vector f()
2  {
3    Vector x{100}, y{10000}, z{10000};
4    // ...
5    x = Vector(10000);   // asignación de movimiento
6    z = x;               // asignación de copia
7    y = std::move(x);    // conversión de x en Vector&& y asig. de mov.
8    //...
9    return z;            // devolución por valor, ctor. de movimiento
10 }
```

Constructor de conversión

- Un 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 }
```

- Es llamado
  - cuando se realiza una conversión explícita de un valor del tipo del parámetro a un objeto de la clase;
  - implícitamente, allí donde el compilador espera un objeto de la clase y encuentra un valor del tipo del parámetro.

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` en inicializaciones

```
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);
```

Constructor de conversión

```
1 class Punto {
2 public:
3   // Evitar conversiones implícitas, sólo explícitas están permitidas.
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);
```

Operador de 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& p): x_{p.x_}, y_{p.y_} {}
7    Punto(const Complejo& c): x_{c.real()}, y_{c.imag()} {}
8    Punto& operator =(const Punto& p)
9        { x_ = p.x_; y_ = p.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() { // Puede ser declarado explicit.
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); // definida en otro sitio
11 Punto p;
12 f(p); // Equivale a f(p.operator 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>    // copy

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)

- Con este constructor 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. Inic. uniforme
```

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

- El destructor no tiene parámetros (por lo que no se puede sobrecargar) y no devuelve nada, ni siquiera `void`.

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

- Se llama implícitamente al finalizar la existencia de un objeto, es decir, cuando:
  - Termina el bloque donde ha sido definido un objeto local con almacenamiento automático (variables locales no `static`).
  - Termina el programa con normalidad mediante `return` o `exit()`, si se trata de un objeto con almacenamiento estático (variables globales y `static`).
  - Un objeto creado en memoria dinámica es eliminado con los operadores `delete` o `delete[]`.
  - Termina la evaluación de una expresión en la que se crean temporales anónimos.

Destructor

- En raras ocasiones es necesario llamar al destructor explícitamente. Si se hace con un objeto ordinario, se puede producir un error de ejecución cuando se llame de nuevo implícitamente al finalizar la duración del objeto.
- A pesar de su nombre, no tiene por función destruir un objeto, sino realizar las tareas previas de limpieza que pueden ser necesarias para que sea destruido correctamente.
- Cuando la construcción de un objeto implica la adquisición de recursos (como p.e., apertura de un fichero, o reserva de memoria dinámica) que han de ser liberados después de utilizados, el programador debe definir el destructor de forma que se ocupe de esta liberación de recursos.
- Si una clase no tiene declarado su destructor, el compilador genera automáticamente uno por defecto, el cual no hace nada.

Métodos especiales

- Métodos especiales son aquellos para los que, bajo ciertas condiciones, se generan (o no) automáticamente definiciones implícitas.
- El programador puede obviar las reglas que determinan estas definiciones implícitas y decidir si incluir o suprimir las «versiones por defecto» de los métodos especiales, definiendo expresamente cada uno como `default` o `delete`.

```
1  class C {
2  public:
3    C()                          = default;
4    C(const C&)                  = delete;    // Los objetos de C
5    C& operator =(const C&)      = delete;    // no se pueden copiar,
6    C(C&&)                       = default;   // pero sí
7    C& operator =(C&&)           = default;   // mover
8    ~C()                         = default;
9    // ...
10 }
```