FACULTAD DE INFORMÁTICA

Objetos y memoria dinámica

TEMA

Programación orientada a objetos — Unidad 8

Autor: Luis Hernández Yáñez

FdI UCM

Contenido

Introducción a la memoria dinámica ... 3
Variables estáticas frente a variables dinámicas ... 4
Punteros ... 5
- El operador & (la dirección de) ... 6
- El operador * (en la dirección) ... 10

Punteros a estructuras u objetos ... 13
Punteros y el modificador **const** ... 14
Punteros y variables dinámicas ... 15
El operador **new** ... 16
El operador **delete** ... 17
Punteros y direcciones válidas - la constante **NULL** ... 18

FdI UCM

Contenido

Objetos dinámicos ... 19
- Creación y destrucción de objetos dinámicos ... 20

Atributos punteros ... 21
Atributos dinámicos ... 22
- Creación de objetos con atributos dinámicos ... 23
- Destrucción de objetos con atributos dinámicos ... 25
- Constructores de copia para atributos dinámicos ... 37
- Operadores de asignación para atributos dinámicos ... 42

Manipulación de objetos con atributos dinámicos ... 48
Atributos relaciones ... 72
- Relaciones en UML ... 75

FdI UCM

Introducción a la memoria dinámica

Mecanismo de gestión de memoria en tiempo de ejecución.

Ajuste de las necesidades de memoria a cada ejecución concreta.

Cuando se necesita memoria para una variable se solicita ésta al Sistema de gestión dinámica de memoria, quien reserva la cantidad adecuada para el tipo de variable y devuelve la dirección de la primera celda de memoria de la zona reservada.

Cuando ya no se necesita más la variable, se libera la memoria que utilizaba indicando al Sistema de gestión dinámica de memoria que puede contar de nuevo con la memoria que se había reservado anteriormente. Para "devolver" esa memoria se proporciona la dirección de la primera celda de la zona reservada.

Dos tipos de datos en los programas:

- ✓ Datos estáticos: se les asigna la memoria que necesitan antes de empezar la ejecución.
- ✓ Datos dinámicos: se les asigna memoria durante la ejecución.

Variables estáticas frente a variables dinámicas

Datos estáticos (variables estáticas) (no confundir con **static**)

- ✓ Variables declaradas como de un tipo concreto: `int i;`
- ✓ Su(s) dato(s) se accede(n) directamente con la propia variable `cout << i;`
- ✓ Existen durante todo el tiempo de ejecución de su ámbito: se les asigna una zona de la memoria principal al comenzar la ejecución y se libera esa memoria al terminar la ejecución.

Datos dinámicos (variables dinámicas)

- ✓ Variables accedidas a través de su dirección de memoria (dirección de la primera celda de memoria utilizada).
- ✓ Se necesita tener guardada esa dirección de memoria inicial en algún otro lugar: en un **puntero**.
- ✓ Los punteros guardan direcciones de memoria y sirven para acceder a las variables dinámicas.

Los datos estáticos también se pueden acceder a través de punteros.

Punteros

LOS PUNTEROS SON DIRECCIONES DE MEMORIA

Una variable puntero (o simplemente puntero) sirve para acceder a través de ella a otra variable.

La variable a la que apunta un puntero, al igual que cualquier otra variable, debe ser de un tipo concreto.

El tipo de variable a la que apunta un puntero se establece al declarar la variable puntero:

tipo ***nombre**;*

El puntero denominado ***nombre*** apuntará a una variable del ***tipo*** indicado (el tipo base del puntero).

```
int *p; // p inicialmente contiene una dirección
        // que no es válida ("no apunta a nada")
```

El puntero **p** apuntará a una variable entera.

Las variables puntero no se inicializan, por lo que tras declararlas contienen direcciones que no son válidas.

Punteros

Un puntero *puede* apuntar a cualquier variable de su tipo base.

Un puntero *no tiene por qué* apuntar necesariamente a una variable (puede no apuntar a nada).

El operador & (*la dirección de*)

El operador monario & devuelve la dirección de memoria inicial de la variable a la que se aplica el operador.

```
int i;
```

```
int *p;
```

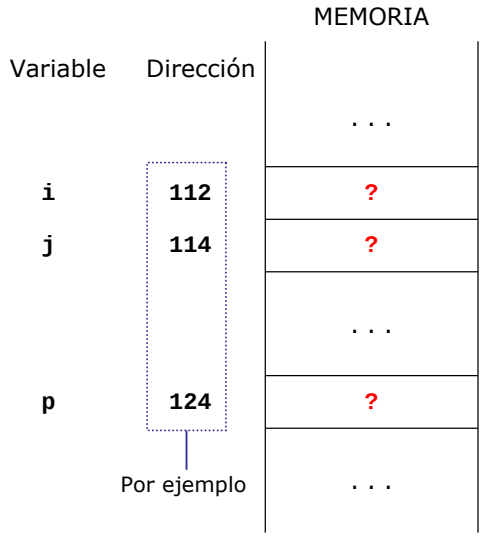
A un puntero se le puede asignar la dirección de una variable del mismo tipo que el tipo base del puntero.

```
p = &i; // la dirección de i
```

Ahora, el puntero **p** ya contiene una dirección de memoria válida.

Punteros

```
int i, j;
...
int *p;
```



Punteros

```
int i, j;
...
int *p;
...
i = 765;
```

MEMORIA

Variable	Dirección	
		...
i	112	765
j	114	?
		...
p	124	?
		...

Punteros

```
int i, j;
...
int *p;
...
i = 765;
p = &i;
```

MEMORIA

Variable	Dirección	
		...
i	112	765
j	114	?
		...
p	124	112
		...

Punteros

El operador * (*en la dirección*)

El operador monario * accede a lo que hay en la dirección de memoria a la que se aplica el operador (un puntero).

Una vez que un puntero contiene una dirección de memoria válida, se puede acceder al dato al que apunta con este operador.

```
p = &i;
cout << *p;
```

***p**: lo que hay en la dirección **p**.

Como el puntero **p** contiene la dirección de memoria de la variable **i**, ***p** accede al contenido de esa variable **i**.

Punteros

```
int i, j;
...
int *p;
...
i = 765;
p = &i;
```

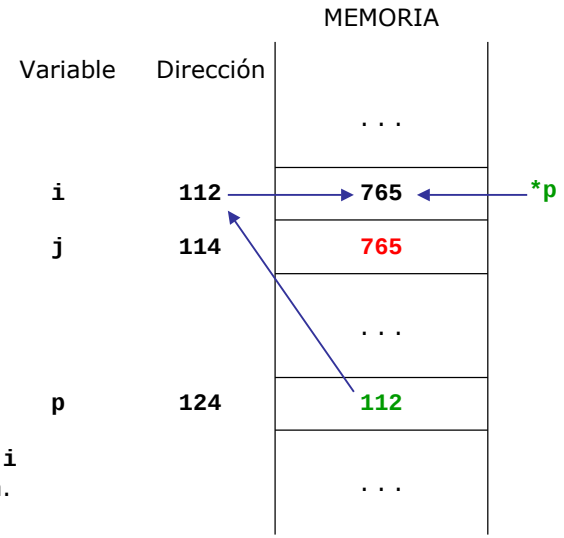
MEMORIA

Variable	Dirección	
		...
i	112	765
j	114	?
		...
p	124	112
		...

La dirección de memoria 112

Punteros

```
int i, j;
...
int *p;
...
i = 765;
p = &i;
j = *p;
```



Esto se llama direccionamiento indirecto (o indirección). Se accede al dato **i** de forma indirecta.

Punteros a estructuras u objetos

Cuando un puntero apunta a una estructura o a un objeto, para acceder a los miembros de la estructura u objeto se ha de utilizar el operador flecha (**->**), en lugar del operador punto:

```
struct Hora {
  int horas;
  int minutos;
  int segundos;
} unaHora;

Hora *p;

p = &unaHora;
unaHora.horas = 12;
cout << p->horas << endl;
```


Punteros y el modificador **const**

Cuando se declaran punteros con el modificador de acceso **const**, su efecto depende de dónde se coloque en la declaración:

```
const tipo *puntero;
```
Puntero a una constante

```
tipo *const puntero;
```
Puntero constante

```
int i[2] = { 12, 21 };
const int *p1 = i;
int *const p2 = i;

(*p1)++; // ERROR: puntero a una constante
p1++; // Dato siguiente del tipo base
(*p2)++;
p2++;    // ERROR: puntero constante
```


Punteros y variables dinámicas

Hasta ahora hemos trabajado con punteros que contienen direcciones de datos estáticos (variables en memoria principal).

Sin embargo, los punteros también son la base sobre la que se apoya
el sistema de gestión dinámica de memoria, el que nos permite crear variables en tiempo de ejecución (mantenidas en el montón o *heap*).

- ✓ Cuando queremos crear una variable dinámica de un tipo determinado, pedimos memoria del montón con el operador **new**.

  El operador **new** reserva la memoria necesaria para ese tipo de variable y devuelve la dirección de la primera celda de memoria asignada a la variable; esa dirección se guarda en un puntero.

- ✓ Cuando ya no necesitemos la variable, devolvemos la memoria que utiliza al montón mediante el operador **delete**.

  Al operador se le pasa un puntero con la dirección de la primera celda de memoria (del montón) utilizada por la variable.

El operador **new**

new *tipo* reserva memoria del montón para una variable de ese *tipo* y devuelve la primera dirección de memoria utilizada, que debe ser asignada a un puntero.

```
int *p; // todavía sin una dirección válida
p = new int; // ya tiene una dirección válida
*p = 12;
```

La variable dinámica se accede exclusivamente a través de punteros; no hay ningún identificador asociado con ella que permita accederla directamente.

```
int i; // i es una variable estática
int *p1, *p2;
p1 = &i; // puntero que da acceso a la variable
         // estática i (accesible con i o con *p1)
p2 = new int; // puntero que da acceso a una
              // variable dinámica
              // (accesible sólo con *p2)
```


El operador **delete**

delete *puntero* devuelve al montón la memoria utilizada por la variable dinámica apuntada por *puntero*.

```
int *p;
p = new int;
*p = 12;
...
delete p; // ya no se necesita el entero
          // apuntado por p
```

El puntero deja de contener una dirección válida y no debe ser utilizado hasta que no contenga nuevamente otra dirección válida.

Punteros y direcciones válidas - la constante **NULL**

Un puntero sólo debe ser utilizado, para acceder al dato al que apunte, si se está seguro de que contiene una dirección válida.

Un puntero NO contiene una dirección válida tras ser definido.

Un puntero obtiene una dirección válida:

- ✓ al asignarle otro puntero (con el mismo tipo base) que ya contenga una dirección válida.
- ✓ al asignarle la dirección de otra variable con el operador **&**.
- ✓ al asignarle la dirección devuelta por el operador **new**.
- ✓ al asignarle el valor **NULL** (que indica que se trata de un puntero nulo, un puntero que no apunta a nada).

```
int i;
int *q; // q no tiene aún una dirección válida
int *p = &i;
q = NULL;
q = p;
q = new int;
```

Formas en las que un puntero toma una dirección válida

Objetos dinámicos

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

#include "persona.h"

int main()
{
  Persona *p;
  // p es un puntero a objeto Persona
```

1. Crear

```
  p = new Persona(); // Creado el objeto Persona dinámico
```

2. Usar

```
  p->leer();    // Uso del objeto pasándole mensajes
  p->mostrar(); // a través del puntero (operador flecha)
```

3. Destruir

```
  delete p; // Liberada la memoria del objeto dinámico
  return 0;
}
```

Uso de objetos dinámicos

Antes de usar el objeto dinámico, pasándole mensajes a través del puntero que lo apunta, se ha de crear el objeto con **new**.

Cuando ya no se necesita el objeto dinámico, se destruye con **delete**.

Objetos dinámicos

Creación y destrucción de objetos dinámicos

Los objetos dinámicos se crean como las variables dinámicas: con **new**. Pero a continuación de **new** colocamos la función constructora que queremos que se ejecute al crear el objeto.

Siempre que se crea un objeto dinámico se ejecuta automáticamente un constructor sobre el objeto dinámico. ¿Cuál? El que indiquemos:

```
p = new Persona();
p = new Persona("223344F", 30, "Juan", "Pérez Gómez");
```

Los objetos dinámicos se destruyen como las variables dinámicas: con **delete**.

```
delete p;
```

Siempre que se destruye un objeto dinámico se ejecuta antes el destructor sobre el objeto dinámico.

Atributos punteros

Cuando en una clase se declara un atributo en forma de puntero, podemos tener dos casos en cuanto a la utilidad del atributo:

- ✓ El atributo va a apuntar a un objeto dinámico, un objeto creado durante la ejecución en memoria dinámica. Decimos que se trata de un *atributo dinámico*. El objeto al que apunta el atributo se considera parte del objeto que contiene el atributo. Aunque no se encuentra realmente dentro, se considera como si fuera el propio atributo.
- ✓ El objeto al que apunta el atributo tiene una existencia independiente del objeto que contiene el atributo puntero. En este caso decimos que se trata de un *atributo relación* (o simplemente *relación*). El atributo puntero establece una relación entre los dos objetos: el objeto que contiene el atributo puntero y el objeto al que apunta el atributo.

Atributos dinámicos

Simplemente otra forma de guardar en memoria los atributos.

```
class Circulo {
  ...
private:
  Punto _centro;
};
```

```
class Circulo {
  ...
private:
  Punto *_centro;
};
```

Se consideran ambos casos equivalentes, aunque con una implementación distinta.

Cada objeto de la clase **Circulo** dispone de su propio centro (objeto de la clase **Punto**), por lo que se ha de crear (**new**) el atributo dinámico al crear los objetos de esa clase **Circulo** (constructores) y se ha de liberar el atributo dinámico (**delete**) al destruir los objetos de esa clase **Circulo**.

Además, el operador de asignación ha de tener en cuenta que ya existe el atributo dinámico en el objeto de destino.

Atributos dinámicos

Creación de objetos con atributos dinámicos

Como los objetos han de disponer en todo momento de sus atributos dinámicos (les son propios, los contienen), cuando se crea un objeto con atributos dinámicos se crean al mismo tiempo esos atributos dinámicos. ¿Dónde? En el constructor, que es lo que se ejecuta automáticamente cuando se crean objetos.

Una clase **Punto** con atributos dinámicos:

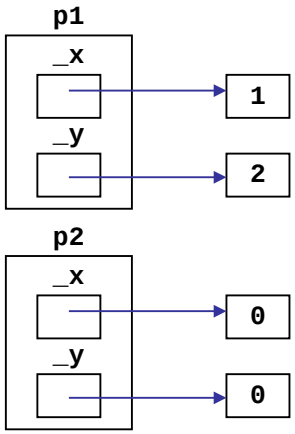
```
class Punto {
public:
  Punto(int x = 0, int y = 0) // Constructor
    { _x = new int; *_x = x; _y = new int; *_y = y; }
  ...
private:
  int *_x;
  int *_y;
};
```

El constructor crea los atributos dinámicos

Atributos dinámicos

Para cada objeto de la clase **Punto** que se crea se crean sus atributos dinámicos. Cada objeto dispone de sus propios atributos dinámicos.

```
int main()
{
  Punto p1(1,2);
  Punto p2();
  ...
  return 0;
}
```



Atributos dinámicos

Destrucción de objetos con atributos dinámicos

De la misma forma, cuando se destruye un objeto con atributos dinámicos, se destruyen al mismo tiempo esos atributos dinámicos. ¿Dónde? En el destructor, que es lo que se ejecuta automáticamente cuando se destruyen objetos.

La clase **Punto** con atributos dinámicos:

```
class Punto {
public:
  Punto(int x = 0, int y = 0) // Constructor
    { _x = new int; *_x = x; _y = new int; *_y = y; }
  ~Punto() { delete _x; delete _y; } // Destructor
  ...
private:
  int *_x;
  int *_y;
};
```

El destructor destruye los atributos dinámicos

Atributos dinámicos

Cuando los constructores y los destructores de las clases se encargan de crear y destruir, respectivamente, los atributos dinámicos,
la creación y destrucción de objetos que contienen atributos dinámicos, que a su vez son objetos de otras clases que contienen atributos dinámicos, se realiza de forma ordenada y correcta.

Por ejemplo, dada la clase **Punto** con atributos dinámicos:

```
class Punto {
public:
  Punto(int x = 0, int y = 0) // Constructor
    { _x = new int; *_x = x; _y = new int; *_y = y; }
  ~Punto() { delete _x; delete _y; } // Destructor
private:
  int *_x;
  int *_y;
};
```


Atributos dinámicos

Sea también la clase **Circulo** con atributos dinámicos, uno de ellos de la clase **Punto**:

```
class Circulo {
public:
  Circulo(int r = 0) {
    _radio = new int;
    *_radio = r;
    _centro = new Punto(); }
  ~Circulo() { delete _centro; delete _radio; }
private:
  Punto *_centro;
  int *_radio;
};
```

Invoca el constructor de **Punto** sobre **_centro**

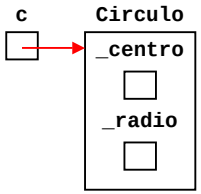
Invoca el destructor de **Punto** sobre **_centro**

Atributos dinámicos

```
Circulo *c = new Circulo(5);
```

Creación

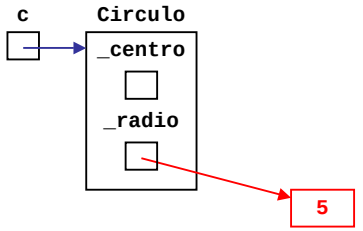
Se crea en memoria dinámica el objeto dinámico de clase **Circulo**, se coloca en **c** su dirección y se ejecuta sobre el objeto el constructor de su clase.



Atributos dinámicos

```
Circulo *c = new Circulo(5);
```

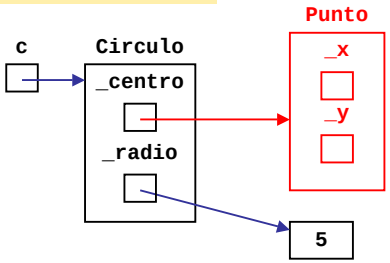
```
Circulo(int r = 5) {
  _radio = new int; *_radio = r;
  _centro = new Punto(); }
```



Atributos dinámicos

```
Circulo *c = new Circulo(5);
```

```
Circulo(int r = 5) {
  _radio = new int; *_radio = r;
  _centro = new Punto(); }
```

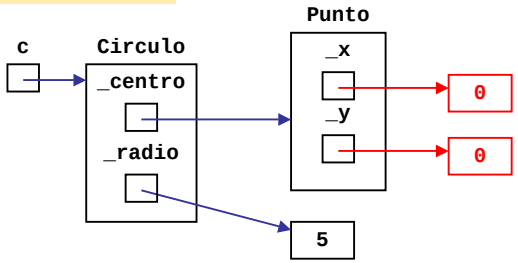


Se ejecuta sobre el objeto dinámico de clase **Punto** el constructor de su clase.

Atributos dinámicos

```
Circulo *c = new Circulo(5);
```

```
Punto(int x = 0, int y = 0) {
  _x = new int; *_x = x;
  _y = new int; *_y = y; }
```

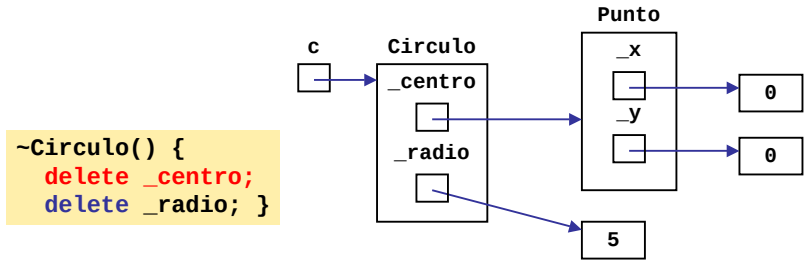


Atributos dinámicos

```
Circulo *c = new Circulo(5);
...
delete c;
```

Destrucción

Se ejecuta sobre el objeto apuntado por **c** el destructor de su clase.

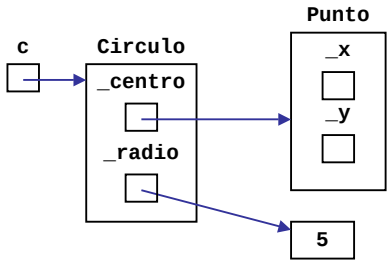



```
~Circulo() {
  delete _centro;
  delete _radio; }
```

Se ejecuta sobre el objeto apuntado por **_centro** el destructor de su clase (**Punto**).

Atributos dinámicos

```
Circulo *c = new Circulo(5);
...
delete c;
```

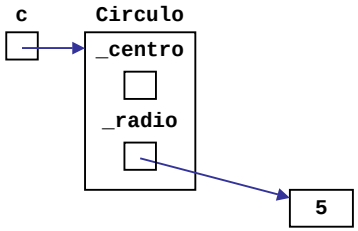



```
~Punto() {
  delete _x;
  delete _y; }
```

Termina la ejecución del destructor y se libera la memoria del objeto dinámico.

Atributos dinámicos

```
Circulo *c = new Circulo(5);
...
delete c;
```

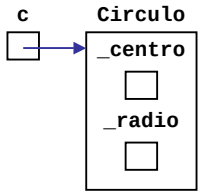


Prosigue la ejecución del destructor de la clase **Circulo**.

Atributos dinámicos

```
Circulo *c = new Circulo(5);
...
delete c;
```

Se ejecuta sobre el objeto apuntado por **c** el destructor de su clase.




```
~Circulo() {
  delete _centro;
  delete _radio; }
```

Termina la ejecución del destructor y se libera la memoria del objeto dinámico.

Atributos dinámicos

```
Circulo *c = new Circulo(5);
...
delete c;
```

Atributos dinámicos

Constructores de copia para atributos dinámicos

Como sabemos, cuando se crean objetos por copia se ejecuta el constructor de copia de la clase.

Si en la clase están definidos atributos dinámicos, el constructor de copia que añade el compilador automáticamente no va a funcionar, ya que copia bit a bit los atributos.

Veamos por qué.

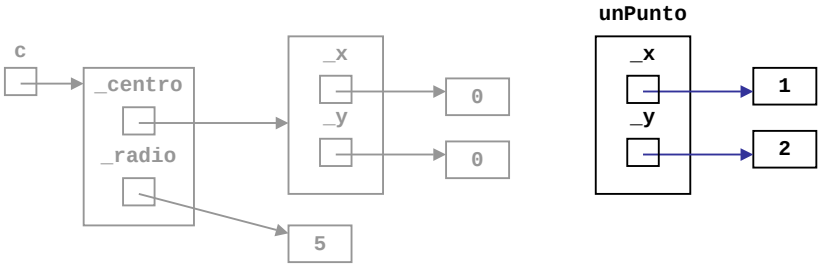
Un caso en el que se ejecuta el constructor de copia de la clase es cuando se pasa un objeto por valor a una función. Se crea el parámetro como una copia del argumento.

```
void Circulo::centro(Punto p) { *_centro = p; }
```

Para crear el objeto parámetro **p** se ejecuta el constructor de copia de la clase **Punto** tomando como punto de partida el argumento.

Atributos dinámicos

```
Circulo *c = new Circulo(5);
Punto unPunto(1,2);
```

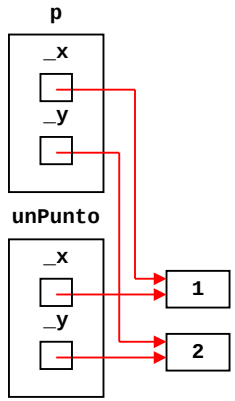


Atributos dinámicos

```
Circulo *c = new Circulo(5);
Punto unPunto(1,2);
...
c->centro(unPunto);
```

El constructor de copia por defecto crea el punto **p** (parámetro de **centro()**) y copia los atributos igualando los valores de los punteros (copia bit a bit).

```
p._x = unPunto._x;
p._y = unPunto._y;
```



Al terminar la ejecución de la función se destruye el objeto parámetro, ejecutándose el destructor que libera los atributos dinámicos.

Atributos dinámicos

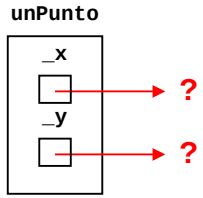
```
Circulo *c = new Circulo(5);
Punto unPunto(1,2);
...
c->centro(unPunto);
```

Como el parámetro y el argumento compartían memoria (los atributos dinámicos), el argumento se queda sin atributos dinámicos válidos.

El problema no se detecta hasta que llega el momento de destruir el objeto que se usó como argumento:

¿
```
delete _x;
delete _y;
```
?

¡¡¡ No hay nada que liberar !!!



Atributos dinámicos

Cuando las clases definen atributos dinámicos el constructor de copia se vuelve imprescindible. Debe crear los atributos dinámicos propios del objeto que construye y luego copiar contenidos:

```
Punto::Punto(const Punto& p) { // Constructor de copia
  _x = new int; *_x = *p._x;
  _y = new int; *_y = *p._y;
}
Circulo::Circulo(const Circulo& c) { // Const. de copia
  _radio = new int; *_radio = *c._radio;
  _centro = new Punto(); *_centro = *c._centro;
}
```

Como se ve, copiar contenidos significa hacer copia entre lo que se apunta y no simplemente copiar los punteros.

Si en todas las clases se procede así, no habrá problemas de compartición de datos. ¡A no ser por el operador de asignación!

Atributos dinámicos

Operadores de asignación para atributos dinámicos

Cuando se copian objetos se ejecuta el operador de asignación de la clase.

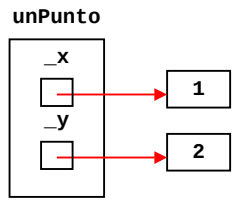
Si en la clase están definidos atributos dinámicos, el operador de asignación que añade el compilador automáticamente no va a funcionar, ya que copia bit a bit los atributos.

Como nos podemos esperar, el problema del operador de asignación por defecto (el que añade el compilador) es similar al del constructor de copia por defecto, el problema de la compartición de datos. Pero en este caso el efecto es incluso peor.

Veamos por qué.

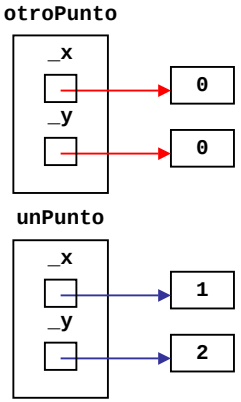
Atributos dinámicos

```
int main() {
  Punto unPunto(1,2);
```



Atributos dinámicos

```
int main() {
  Punto unPunto(1,2);
  Punto otroPunto;
```



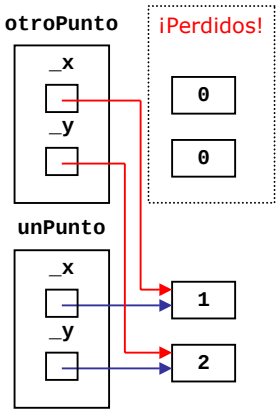
Atributos dinámicos

```
int main() {
  Punto unPunto(1,2);
  Punto otroPunto;
  otroPunto = unPunto;
```

El operador de asignación por defecto copia los atributos igualando los valores de los punteros (copia bit a bit).

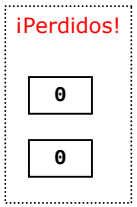
```
otroPunto._x = unPunto._x;
otroPunto._y = unPunto._y;
```

En este caso, además de llegarse a una situación de compartición de datos, también se han dejado datos perdidos en la memoria dinámica, lo que no debemos permitir que ocurra nunca.



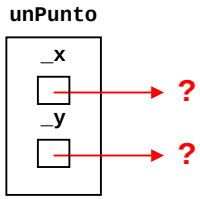
Atributos dinámicos

```
int main() {
  Punto unPunto(1,2);
  Punto otroPunto;
  otroPunto = unPunto;
  ...
  return 0;
}
```



Nuevamente, los problemas no los detectaremos hasta más adelante. El programa fallará cuando proceda con la destrucción de los objetos.

Se destruye primero **otroPunto**, el segundo objeto creado. Se libera la memoria de sus atributos dinámicos.



Cuando se intenta a continuación destruir **unPunto**, no hay atributos dinámicos que liberar y el `delete` falla, arrastrando al programa.

Atributos dinámicos

Cuando las clases definen atributos dinámicos el operador de asignación propio se vuelve imprescindible. Debe copiar contenidos, los objetos apuntados por los atributos, no simplemente los punteros:

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

Circulo& Circulo::operator=(const Circulo& c) {
  *_radio = *c._radio;
  *_centro = *c._centro;
  return *this;
}
```

Como el objeto sobre el que se copia ya tiene creados los atributos dinámicos, simplemente se copia lo apuntado.

Manipulación de objetos con atributos dinámicos

Implementando adecuadamente el constructor de copia y el operador de asignación en las clases con atributos dinámicos, nos aseguramos de que la manipulación de objetos de esas clases (construcción, copia y destrucción) funciona correctamente.

- ✓ El constructor por copia se ejecuta cuando se crean copias de forma automática, como cuando se pasa un objeto como argumento de un parámetro por valor de una función.
- ✓ El operador de asignación sobrecargado se ejecuta cuando se realizan explícitamente copias de objetos (haciendo uso del operador de asignación).

El encadenamiento adecuado de los procesos de creación, copia y destrucción desde los objetos hacia sus atributos dinámicos garantizará la ausencia de problemas de memoria dinámica debidos a la compartición o ausencia de datos dinámicos.

Manipulación de objetos con atributos dinámicos

El archivo **punto.h**

```
#ifndef punto_h
#define punto_h

// Todos los métodos implementados fuera
class Punto {
public:
  Punto(int = 0, int = 0); // Constructor (predeterminado)
  Punto(const Punto&); // Constructor de copia
  Punto& operator=(const Punto&) // Operador de asignación
  ~Punto(); // Destructor
  int x() const;
  int y() const;
  void x(int);
  void y(int);
private:
  int *_x, *_y;
};

#endif
```


Manipulación de objetos con atributos dinámicos

El archivo **punto.cpp**

```
#include "punto.h"

Punto::Punto(int x, int y) {
  _x = new int; *_x = x;
  _y = new int; *_y = y;
}

Punto::Punto(const Punto& p) {
  _x = new int; *_x = *p._x;
  _y = new int; *_y = *p._y;
}

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

Punto::~Punto() { delete _x; delete _y; }
```

(*continúa*)

Manipulación de objetos con atributos dinámicos

```
int Punto::x() const { return *_x; }

int Punto::y() const { return *_y; }

void Punto::x(int i) { *_x = i; }

void Punto::y(int i) { *_y = i; }
```

Los accedentes y mutadores trabajan con lo apuntado por el atributo puntero (con el atributo dinámico)

El archivo **circulo.h**

```
#ifndef circulo_h
#define circulo_h
#include "punto.h" // Inclusión de la clase Punto

class Circulo {
public:
  Circulo(int radio = 0); // Constructor (predeterminado)
  Circulo(const Circulo&); // Constructor de copia
  Circulo& operator=(const Circulo&); // Asignación
  ~Circulo(); // Destructor
  int radio() const;
  Punto centro() const;
  void radio(int);
  void centro(Punto);
private:
  int *_radio;
  Punto *_centro;
};

#endif
```


El archivo **circulo.cpp**

```
#include "circulo.h"

Circulo::Circulo(int radio) {
  _radio = new int; *_radio = radio;
  _centro = new Punto();
}

Circulo::Circulo(const Circulo& c) {
  _radio = new int; *_radio = *c._radio;
  _centro = new Punto(); *_centro = *c._centro;
}
```

Constructor de la clase **Punto**

Operador de asignación de la clase **Punto**

Constructor de la clase **Punto**

(*continúa*)


```
Circulo& Circulo::operator=(const Circulo& c) {
  *_radio = *c._radio; *_centro = *c._centro;
  return *this;
}
Circulo::~Circulo() { delete _radio; delete _centro; }

int Circulo::radio() const { return *_radio; }

Punto Circulo::centro() const { return *_centro; }

void Circulo::radio(int r) { *_radio = r; }

void Circulo::centro(Punto p) { *_centro = p; }
```

Operador de asignación de la clase **Punto**

Destructor de la clase **Punto**

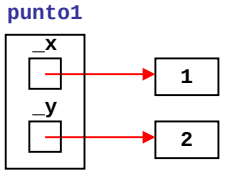
Constructor de copia de la clase **Punto**

Constructor de copia de la clase **Punto**

Operador de asignación de la clase **Punto**


```
int main()
{
  Punto punto1(1,2);
```

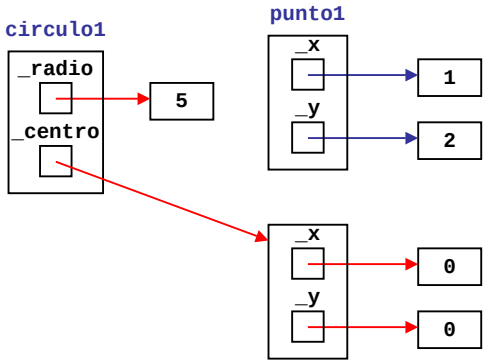
Se ejecuta el constructor de la clase **Punto**.



Manipulación de objetos con atributos dinámicos

```
int main()
{
  Punto punto1(1,2);
  Circulo circulo1(5);
```

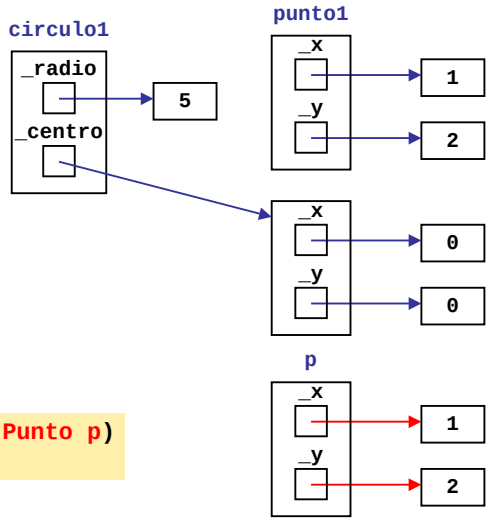
Se ejecutan el constructor de la clase **Circulo** y el de la clase **Punto**.



Manipulación de objetos con atributos dinámicos

```
int main()
{
  Punto punto1(1,2);
  Circulo circulo1(5);
  circulo1.centro(punto1);
```

Se ejecuta el constructor de copia de la clase **Punto** para crear el parámetro **p** del mutador como una copia del argumento.




```
void Circulo::centro(Punto p)
{ *_centro = p; }
```

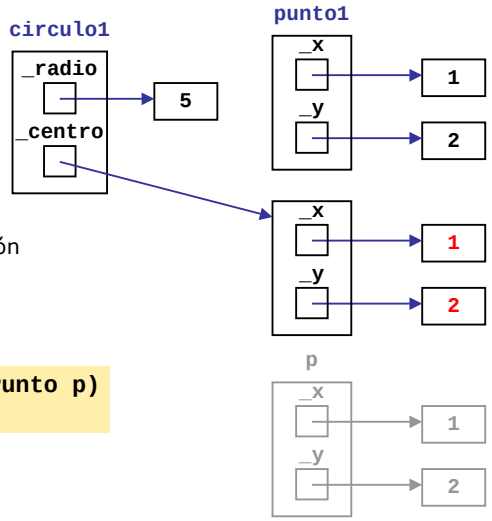

Manipulación de objetos con atributos dinámicos

```
int main()
{
  Punto punto1(1,2);
  Circulo circulo1(5);
  circulo1.centro(punto1);
```

Se ejecuta el operador de asignación de la clase **Punto** para copiar el parámetro **p** del mutador sobre el atributo dinámico.

```
void Circulo::centro(Punto p)
{ *_centro = p; }
```

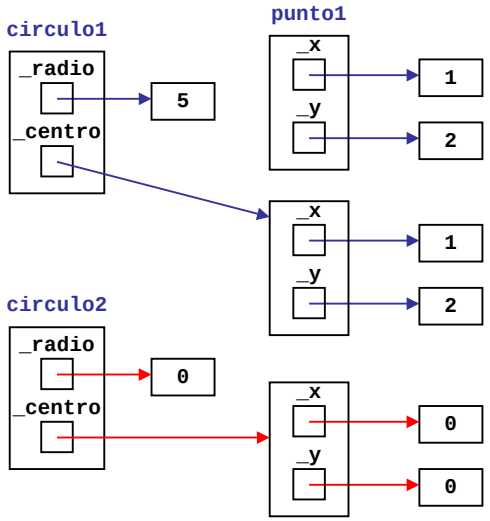
Y se destruye el parámetro.



Manipulación de objetos con atributos dinámicos

```
int main()
{
  Punto punto1(1,2);
  Circulo circulo1(5);
  circulo1.centro(punto1);
  Circulo circulo2;
```

Se ejecuta el constructor predeterminado de la clase **Circulo**.

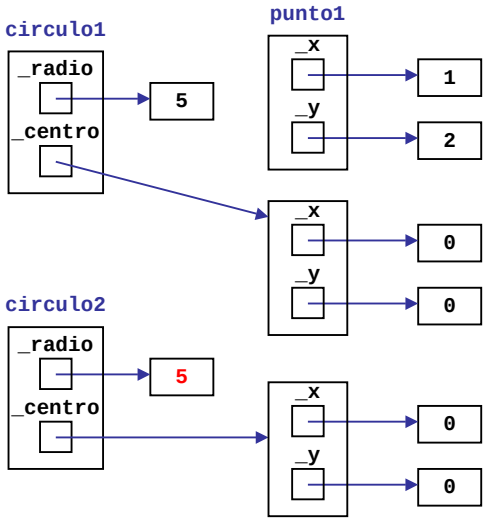


Manipulación de objetos con atributos dinámicos

```
int main()
{
  Punto punto1(1,2);
  Circulo circulo1(5);
  circulo1.centro(punto1);
  Circulo circulo2;
  circulo2 = circulo1;
```

Asignación de
la clase **Circulo**.

```
*_radio = *c._radio;
```

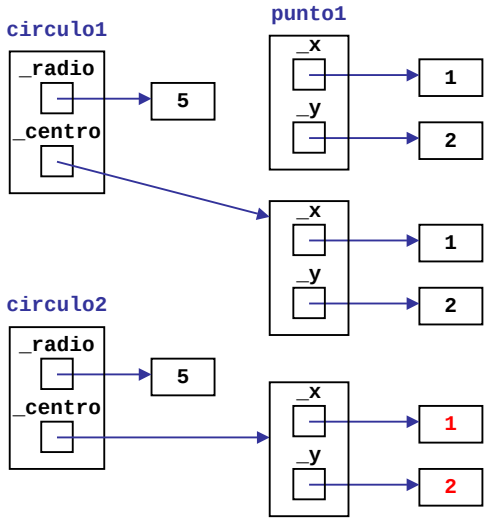


Manipulación de objetos con atributos dinámicos

```
int main()
{
  Punto punto1(1,2);
  Circulo circulo1(5);
  circulo1.centro(punto1);
  Circulo circulo2;
  circulo2 = circulo1;
```

Asignación de la clase **Circulo**

```
*_radio = *c._radio;
*_centro = *c._centro;
```



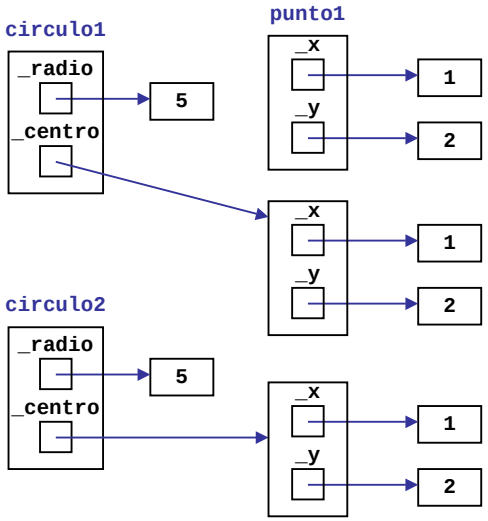
Manipulación de objetos con atributos dinámicos

```
int main()
{
  Punto punto1(1,2);
  Circulo circulo1(5);
  circulo1.centro(punto1);
  Circulo circulo2;
  circulo2 = circulo1;
  return 0;
```

Proceso de destrucción
de objetos.
Para **circulo2**:

```
delete _centro;
```

ejecuta el destructor de **Punto**.



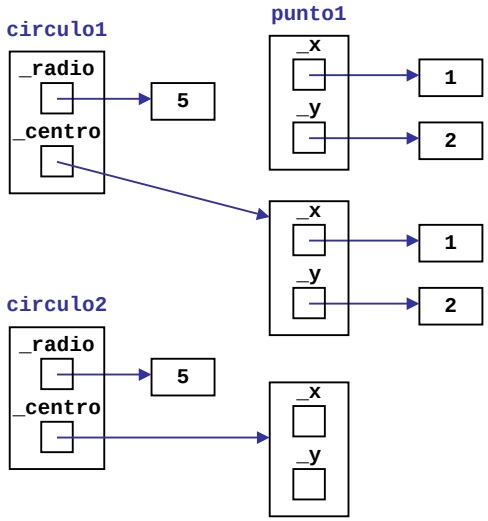
Manipulación de objetos con atributos dinámicos

```
int main()
{
  Punto punto1(1,2);
  Circulo circulo1(5);
  circulo1.centro(punto1);
  Circulo circulo2;
  circulo2 = circulo1;
  return 0;
```

Proceso de destrucción
de objetos.
Para **circulo2**:

```
delete _centro;
```

y libera la memoria.

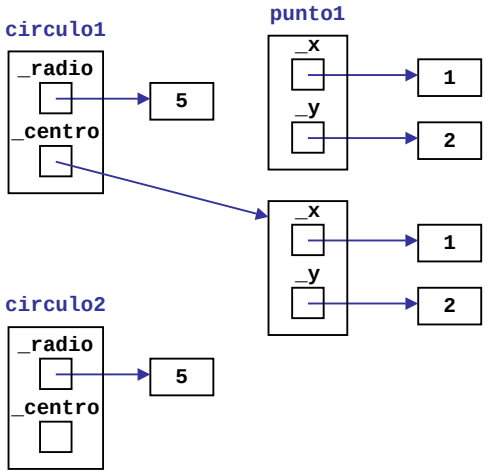


Manipulación de objetos con atributos dinámicos

```
int main()
{
  Punto punto1(1,2);
  Circulo circulo1(5);
  circulo1.centro(punto1);
  Circulo circulo2;
  circulo2 = circulo1;
  return 0;
```

Proceso de destrucción de objetos
Para **circulo2**:

```
delete _radio;
```

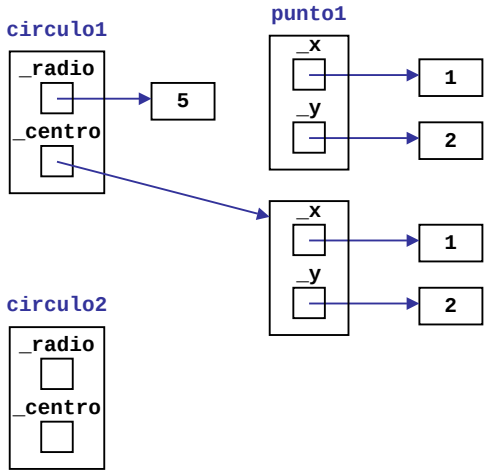


Manipulación de objetos con atributos dinámicos

```
int main()
{
  Punto punto1(1,2);
  Circulo circulo1(5);
  circulo1.centro(punto1);
  Circulo circulo2;
  circulo2 = circulo1;
  return 0;
```

Proceso de destrucción de objetos.
Para **circulo1**:

Mismo proceso.

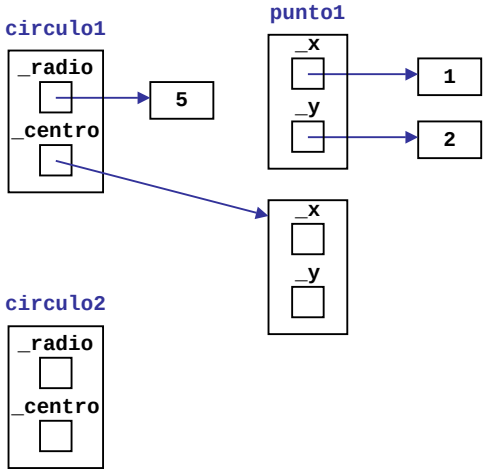


Manipulación de objetos con atributos dinámicos

```
int main()
{
  Punto punto1(1,2);
  Circulo circulo1(5);
  circulo1.centro(punto1);
  Circulo circulo2;
  circulo2 = circulo1;
  return 0;
```

Proceso de destrucción de objetos.
Para **circulo1**:

Mismo proceso.

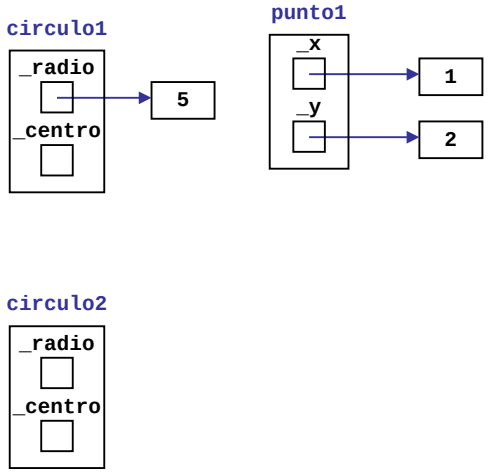


Manipulación de objetos con atributos dinámicos

```
int main()
{
  Punto punto1(1,2);
  Circulo circulo1(5);
  circulo1.centro(punto1);
  Circulo circulo2;
  circulo2 = circulo1;
  return 0;
```

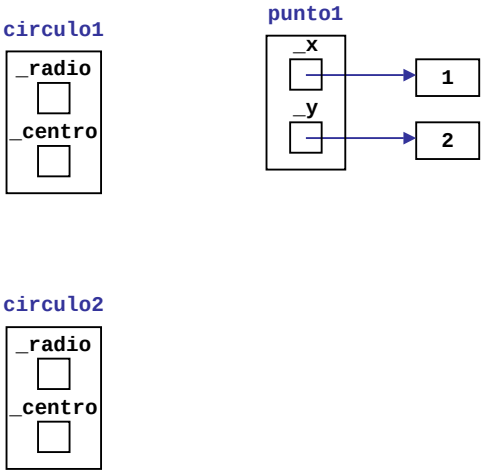
Proceso de destrucción de objetos.
Para **circulo1**:

Mismo proceso.



Manipulación de objetos con atributos dinámicos

```
int main()
{
  Punto punto1(1,2);
  Circulo circulo1(5);
  circulo1.centro(punto1);
  Circulo circulo2;
  circulo2 = circulo1;
  return 0;
```

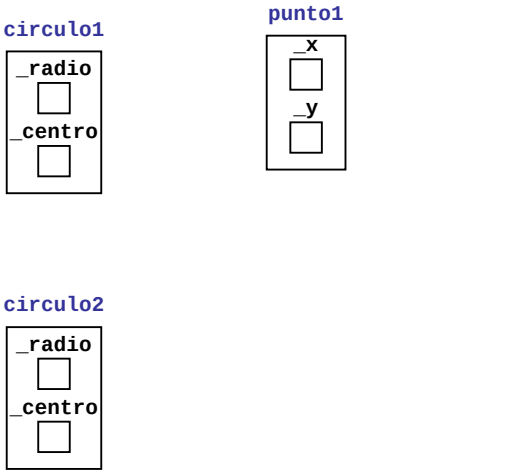


Proceso de destrucción de objetos.
Para **punto1**:

Destructor de su clase.

Manipulación de objetos con atributos dinámicos

```
int main()
{
  Punto punto1(1,2);
  Circulo circulo1(5);
  circulo1.centro(punto1);
  Circulo circulo2;
  circulo2 = circulo1;
  return 0;
```



Finalmente se libera la memoria principal (datos estáticos).

Manipulación de objetos con atributos dinámicos

```
int main()
{
  Punto punto1(1,2);
  Circulo circulo1(5);
  circulo1.centro(punto1);
  Circulo circulo2;
  circulo2 = circulo1;
  return 0;
}
```

La ejecución concluye.

Atributos relaciones

Relaciones con objetos "externos"

El atributo implementa una relación entre los dos objetos.

```
class Alumno : public Persona {
  ...
private:
  Persona *_profesor;
  ...
};
```

El objeto **Persona** al que apunta el atributo **_profesor** de los objetos **Alumno** es un objeto con existencia propia, de forma que no se ve afectado por la existencia o no de los objetos **Alumno**.
La **Persona** existirá independientemente de que en objetos **Alumno** se haga referencia a la misma, indicando una relación entre ellos.

Los constructores no deben crear el objeto al que se referirá el atributo, como tampoco debe ser liberado en el destructor.
Y el operador de asignación copiará la relación (punteros).

Atributos relaciones

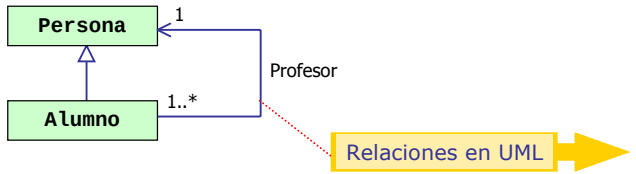
El archivo **alumno.h**

```
#ifndef alumno_h
#define alumno_h

#include "persona.h" // Inclusión de la clase Persona

class Alumno : public Persona {
public:
  // Constructor (predeterminado):
  Alumno(Persona* = NULL, string = "", int = 0,
         string = "", string = "", int = 1);
  Alumno(const Alumno&); // Constructor de copia
  Alumno& operator=(const Alumno&); // Asignación
  ~Alumno(); // Destructor
  void curso(int);
  void profesor(Persona*);
  int curso() const;
  Persona* profesor() const;
  void leer();
  void mostrar() const;
```

(*continúa*)

Atributos relaciones

```
private:
  int _curso;
  Persona *_profesor; // RELACIÓN con una Persona
};                    // La que es profesor de ésta

#endif
```



Atributos relaciones

Relaciones en UML

Se representan con líneas, pudiendo indicar con flechas el sentido de la relación:

Los **Alumnos** tienen algún atributo relación que es un puntero a **Persona**.

La relación se llama *Profesor* (el profesor del alumno es una **Persona**).

En este caso, además, se indica que cada **Alumno** se relaciona con una sola **Persona** (un solo profesor) y que muchos **Alumnos** pueden tener el mismo profesor.

Atributos relaciones

El archivo **alumno.cpp**

```
#include "alumno.h"
#include <iostream>
#include <string>
using namespace std;

Alumno::Alumno(Persona* profe, string nif, int edad,
   string nombre, string apellidos, int curso) :
   _curso(curso), _profesor(profe),
  Persona(nif, edad, nombre, apellidos) { }

Alumno::Alumno(const Alumno& otro) : Persona(otro)
{
  _curso = otro._curso;
  _profesor = otro._profesor;
}
```

Los atributos relaciones se manejan igual que los atributos que no son punteros. No requieren operaciones de reserva (**new**) o liberación (**delete**) de memoria.

(*continúa*)

Atributos relaciones

```
Alumno& Alumno::operator=(const Alumno& otro)
{
  Persona::operator=(otro);
  _curso = otro._curso;
  _profesor = otro._profesor;
  return *this;
}

Alumno::~Alumno() {}

void Alumno::curso(int num) { _curso = num; }

void Alumno::profesor(Persona* profe)
{ _profesor = profe; }

int Alumno::curso() const { return _curso; }

Persona* Alumno::profesor() const { return _profesor; }
```

No hay **delete**

(*continúa*)

Atributos relaciones

```
void Alumno::mostrar() const
{
  Persona::mostrar();
  cout << "Curso: " << _curso << endl;
  cout << "Profesor:" << endl;
  if(_profesor == NULL) cout << "No asignado" << endl;
  else _profesor->mostrar();
}

void Alumno::leer()
{
  Persona::leer();
  cout << "Curso: ";
  cin >> _curso;
}
```

Los datos del **_profesor**
ya se habrán obtenido en alguna otra parte

Atributos relaciones

```
#include "persona.h"
#include "alumno.h"
int main()
{
  Persona *p; // Puntero a objeto Persona
  p = new Persona(); // Creado el objeto Persona dinámico
  p->leer();
  p->mostrar(); // Uso del operador flecha
  Alumno *a; // Puntero a objeto Alumno
  a = new Alumno(); // Creado el objeto Alumno dinámico
  a->leer();
  a->profesor(p);
  a->mostrar(); // Uso del operador flecha
  delete a; // Liberada la memoria del objeto Alumno
  delete p; // Liberada la memoria del objeto Persona

  return 0;
}
```

2 variables estáticas (**p** y **a**)
2 variables dinámicas (objetos **Persona** y **Alumno**
a los que apuntan las variables estáticas)