Tema VII
POO y lenguajes de programación no Orientados a Objetos (R-1.1)

Programación en Entornos Interactivos.

16 de febrero de 2012

Dpto. Lenguajes y Sistemas Informáticos
Universidad de Alicante

Resumen

Cómo realizar Programación orientada a Objetos con lenguajes no orientados a objetos. Representación de clases. Paso de argumentos a métodos. Reserva de memoria para objetos. Representación de la herencia. Resolución de métodos.

Preliminares.

- La utilización del paradigma de programación orientada a objetos requiere cierta disciplina por parte del programador y por parte del lenguaje empleado.
- Con los lenguajes no orientados a objetos esto último no es posible y todo queda en manos del programador.
- Por tanto, la realización de la POO con un Lenguaje no orientado a objetos se consigue mediante el uso de una serie de normas auto-impuestas que nos permiten simular el uso de: clases, herencia, etc. . .
- A lo largo del tema vamos a ver cuáles son y cómo se aplican estas normas.

¿Qué vamos a simular?

- Representación de clases.
- Paso de argumentos a métodos.
- Reserva de memoria e iniciación de los objetos.
- Representación de la herencia simple.
- Resolución de métodos en tiempo de ejecución.

¿Con qué lenguaje?

- Nos vamos a centrar en lenguaje 'C' dado que la biblioteca GTK+[1]. está escrita en él y emplea las técnicas que veremos.
- Vamos a aprovechar de él la buena gestión de los punteros y de la reserva de memoria dinámica que nos facilita.
- Debido a estas peculiaridades la simulación de las características antes comentadas se llevará a cabo con poca o nula pérdida de eficiencia.

[1] No confundir con Gtkmm.

Representación de las ‘clases’. (I)

- Emplearemos aquello que más se le parece: ‘struct’
- En ‘C’ cada clase del diseño se ‘convierte’ en un ‘struct’ del lenguaje.
- Cada atributo definido en la clase es una campo del ‘struct’ creado.

Representación de las ‘clases’. (II)

- Un objeto podrá estar ubicado en el almacenamiento global, en la pila o en memoria dinámica.
- La referencia a un objeto se puede representar mediante un puntero a su ‘struct’:

Referencia a un objeto

```
typedef float Length
struct Ventana {
  Length xmin;
  Length ymin;
  Length xmax;
  Length ymax;
}
...
struct Ventana* ventana;
...
Lenght x1 = ventana->xmin;
```

Paso de argumentos a métodos. (I)

- Es aconsejable seguir un convenio consistente para dar nombre a los métodos:
  Nombre–clase__nombre–operacion
- Asumimos que cada método tiene al menos un argumento, al cual llamaremos self, representa al objeto receptor del mensaje y. . . se ha de declarar explícitamente siempre.
- El tipo de este primer parámetro será puntero a la clase receptora del mensaje, veámoslo con un ejemplo:

Paso de argumentos a métodos. (II)

- Ejemplo:

Nombre de método y paso argumentos

```
Ventana__anyadir_a_seleccionados(struct Ventana* self,
                                 struct Figura*  figura)
```

- El paso de objetos como parámetros se hará con punteros:
  - Es más eficiente.
  - Podemos emplear el objeto dentro del método tanto para operaciones de consulta como para operaciones de actualización del mismo.

Reserva de memoria para objetos. (I)

- Los objetos se pueden crear estáticamente en tiempo de compilación , dinámicamente (heap) o localmente en la pila (stack).
- Los objetos creados estáticamente se representan por variables globales y su tiempo de 'vida' coincide con el de la duración del programa. No es aconsejable el uso de variables globales: Polución del espacio de nombres.
- Los objetos globales se pueden declarar como variables globales de algún tipo de 'struct' –*su clase*–, además pueden iniciarse en tiempo de compilación.
- A la hora de llamar a un método que utilice un objeto global, debemos pasar la dirección de la variable.

Reserva de memoria para objetos. (II)

- Podemos declarar objetos temporales como variables locales a una función o a un bloque, exactamente igual que cualquier otra variable.
- La creación de objetos en memoria dinámica se realiza con malloc o calloc. Podemos aprovechar este instante para iniciar el objeto. . . constructor.
- Debemos liberar la memoria que ocupa con free.

Reserva de memoria para objetos. (III)

Creación/Iniciación

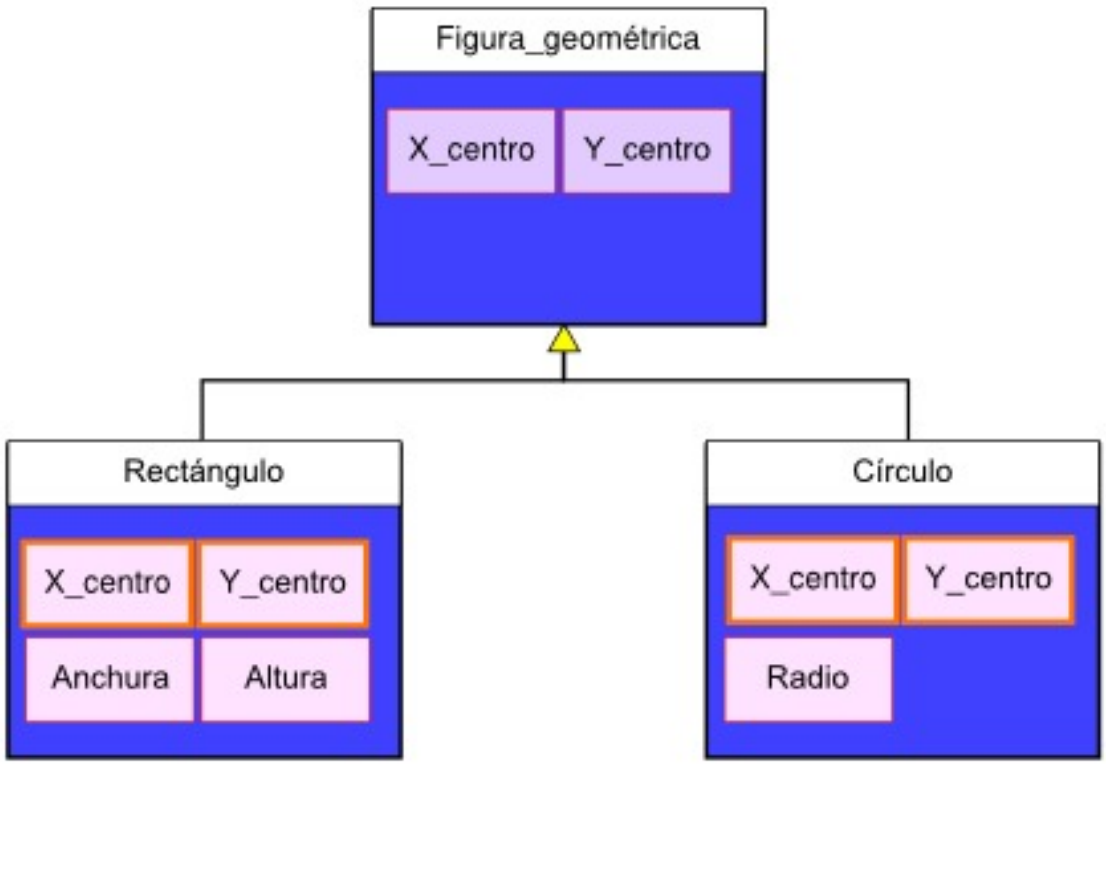
```
/* Objeto global */
struct Ventana ventana_global = {0.0,0.0,8.5,7.0};
/* Objetos en memoria dinamica */
struct Ventana* crear_ventana(Length xmin,  Length ymin,
                              Length width, Length height)
{
  struct Ventana* ventana;

  ventana = (struct Ventana*) malloc(sizeof(struct Ventana));
  ventana -> xmin = xmin;
  ventana -> ymin = ymin;
  ventana -> xmax = xmin + width;
  ventana -> ymax = ymin + height;
  return (ventana);
}
```

Representación de la herencia. (I)

- La simulación de la herencia simple se basa en una idea muy sencilla:
  1. Agrupar en un 'struct' o registro los atributos heredados de la clase base, a continuación...
  2. Insertar este registro al principio de cada clase derivada de ella.
  3. Añadir la parte específica de la clase derivada.
- Veámoslo con un ejemplo muy sencillo:

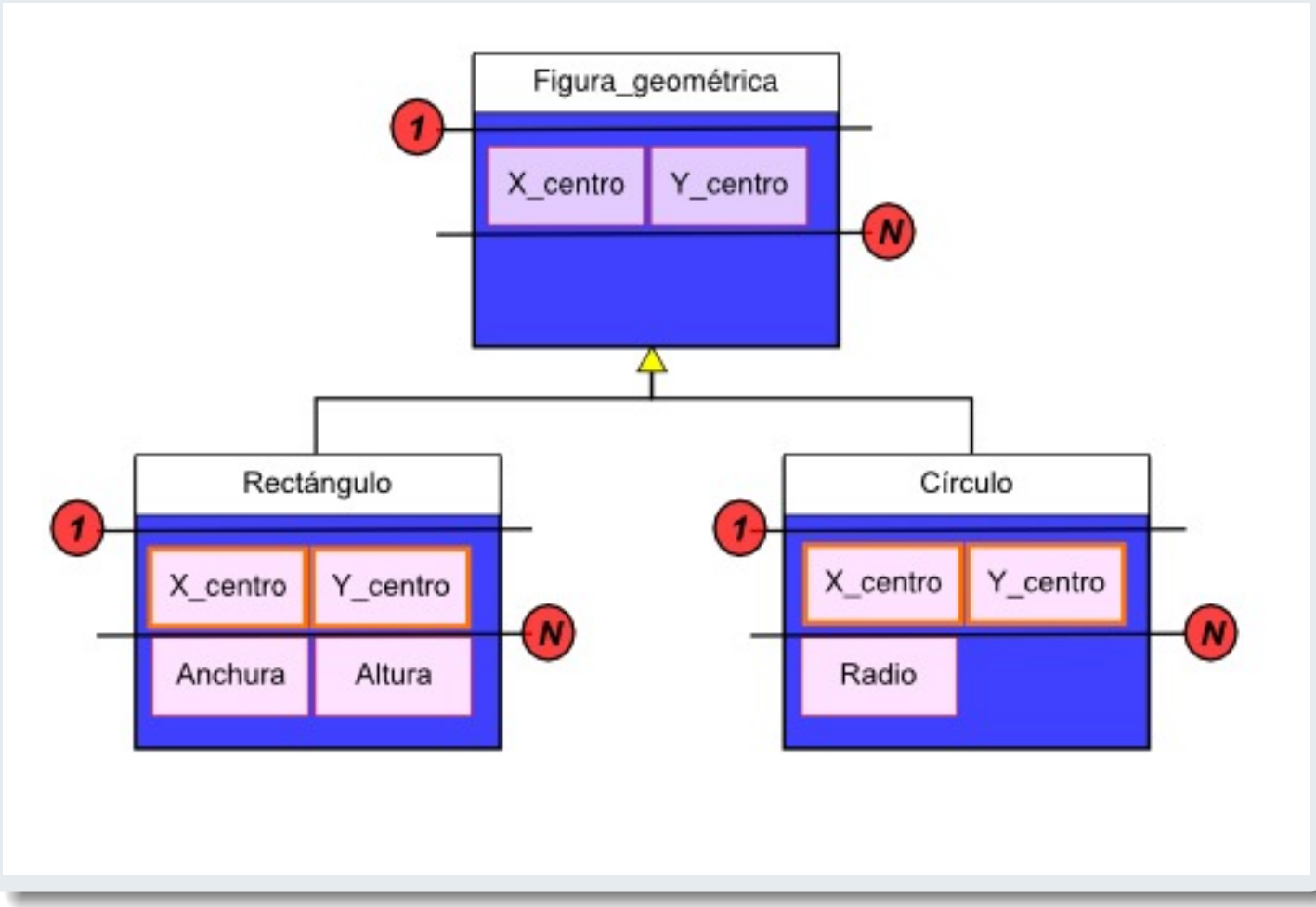
Representación de la herencia. (II)

Simulación de la herencia: figuras geométricas



Representación de la herencia. (III)

Simulación de la herencia: figuras geométricas

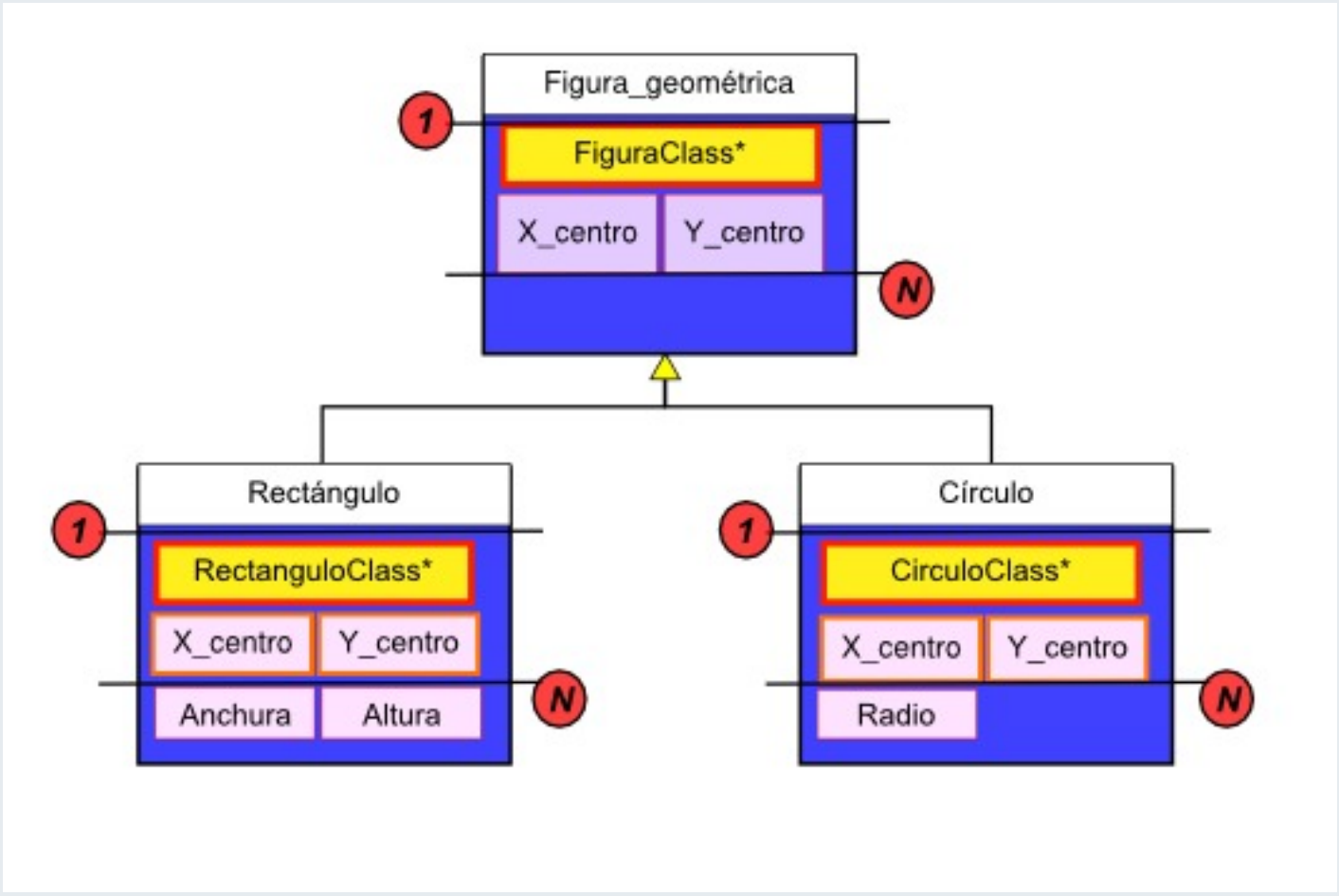


Representación de la herencia. (IV)

- Para que esta simulación sea completa y podamos añadir posteriormente los '*métodos*' a la clase, necesitamos añadir un primer campo a cada '*struct*' de las anteriores.
- Se trata de un puntero a su descriptor de clase. Se llamará 'dc'.
- Los descriptores de clase son otro tipo de struct que incluyen los métodos y variables de clase.
- Por tanto el gráfico anterior quedaría así:

Representación de la herencia. (V)

Simulación de la herencia: figuras geométricas



Representación de la herencia. (VI)

- Utilizando esta técnica podemos pasar un puntero a un objeto de tipo Rectangulo o Circulo a una función que espere un puntero a un objeto de tipo Figura.
- Esto es posible en virtud de la relación is a que hay entre una clase derivada y su base.
- En el siguiente ejemplo un puntero a un Rectangulo se interpreta como un puntero a una Figura:

Representación de la herencia. (VII)

Paso de punteros

```
struct Rectangulo* rec;
struct Ventana*    ventana;

/* prototipo de: Ventana__anyadir_a_seleccionados */
void Ventana__anyadir_a_seleccionados(struct Ventana* ventana,
                                      struct Figura*  fig);


...
Ventana__anyadir_a_seleccionados(ventana, rec);
```

Resolución de métodos. (I)

- Se trata de elegir en tiempo de ejecución qué método invocar en respuesta a un mismo mensaje enviado a distintos objetos, cada uno de los cuales lo hace a su manera.
- En lenguaje 'C' vamos a emplear punteros a funciones para realizarlo.
- Para implementarlo de forma sencilla vamos a hacer uso del descriptor de clase comentado anteriormente.

Resolución de métodos. (II)

- El Descriptor de Clase es una estructura que contiene:
  - Una cadena con el nombre de la clase que representa.
  - Un puntero a cada método de la clase, incluídos los heredados.
  - Las variables de clase que pueda tener la clase que representa.
- Los *descriptores de clase* sólo son necesarios para aquellas clases que vayan a tener instancias y no para clases abstractas tales como 'Figura'.
- El nombre de la estructura que representa al *descriptor de clase* es el mismo que el de la clase añadiéndole el sufijo '*Class*': FiguraClass, CirculoClass...
- Veamos un ejemplo de descriptores de clase:

Resolución de métodos. (III)

Descriptores de clase

```
struct FiguraClass {
  char*   nombre_clase;
  void    (*mover)        ();
  Boolean (*seleccionar) ();
  void    (*desagrupar)  ();
  void    (*dibujar)     ();
};

struct CirculoClass {
  char*   nombre_clase;
  void    (*mover)        ();
  Boolean (*seleccionar) ();
  void    (*desagrupar)  ();
  void    (*dibujar)     ();
};
```

```
struct RectanguloClass {
  char*   nombre_clase;
  void    (*mover)        ();
  Boolean (*seleccionar) ();
  void    (*desagrupar)  ();
  void    (*dibujar)     ();
};
```

Resolución de métodos. (IV)

- La estructura, 'struct', del descriptor de clase define los nombres de las operaciones 'visibles' de la clase, pero sólo eso.
- Todavía tenemos que definir e iniciar un 'objeto descriptor de clase' para cada clase.
- Cada uno de estos 'objetos descriptores de clase' es una única variable global, la cual será la única instancia de la clase 'descriptor de clase' correspondiente.
- Cada 'campo' del objeto descriptor de clase debe ser iniciado con el nombre de la función de 'C' definida o heredada por la clase:

Resolución de métodos. (V)

Iniciación del descriptor de clase

```
struct RectanguloClass RectanguloClass = {
        "Rectangulo",
        Figura__mover,
        Rectangulo__seleccionar,
        Figura__desagrupar,
        Rectangulo__dibujar
};

struct CirculoClass CirculoClass = {
        "Circulo",
        Figura__mover,
        Circulo__seleccionar,
        Figura__desagrupar,
        Circulo__dibujar
};
```

Resolución de métodos. (VI)

- Cuando se crea un objeto, guardamos en su primer campo ‘dc’, que es de tipo ‘puntero a su descriptor de clase’, la dirección del objeto descriptor de la clase.
- De este modo, y en tiempo de ejecución, podemos obtener:
  - La clase a la que pertenece el objeto.
  - El nombre de esta clase.
  - Los métodos asociados a esta clase.
- Por ejemplo, la creación de un objeto de clase ‘Circulo’ se haría así:

Resolución de métodos. (VII)

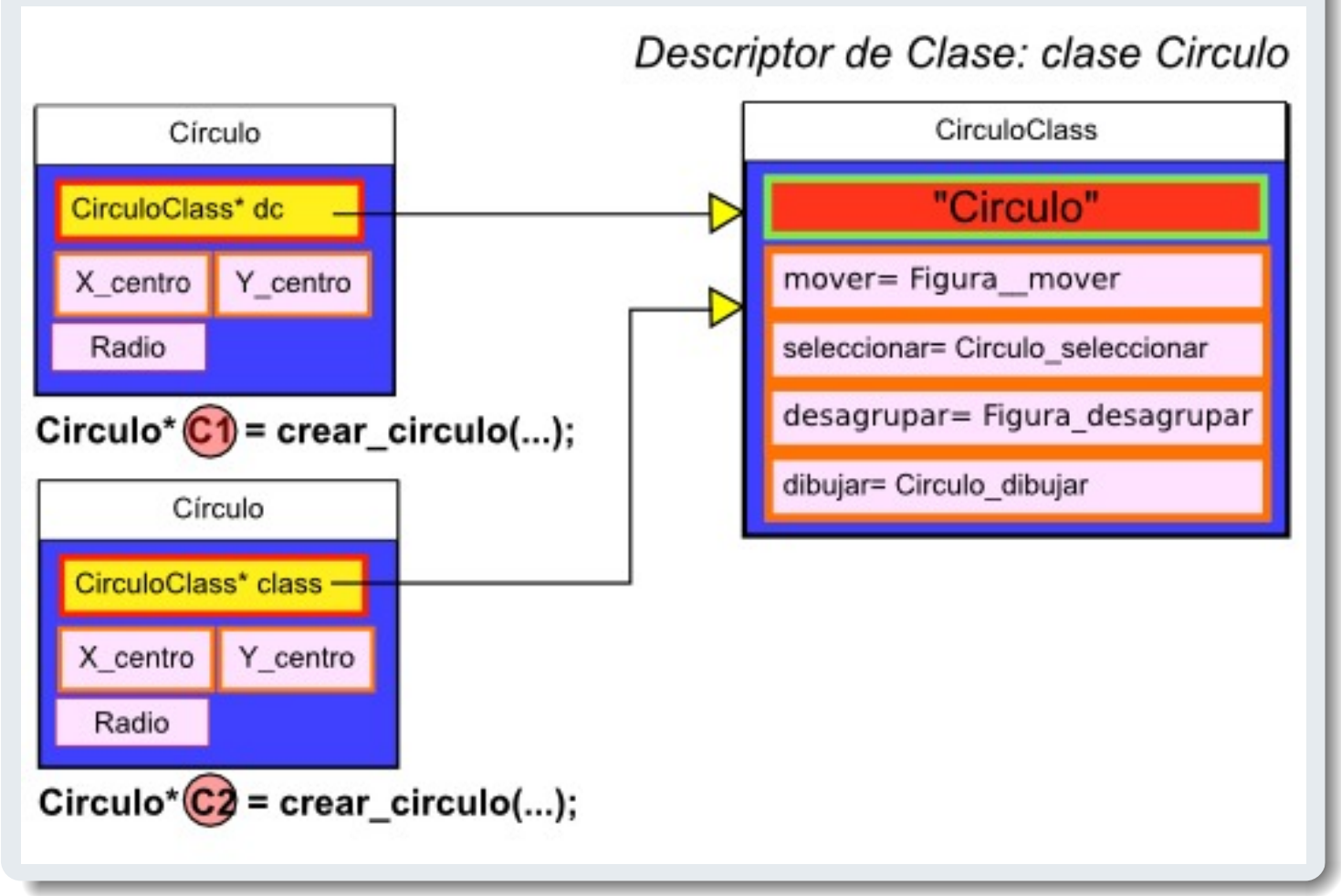
Creación de objetos / Descriptor de clase

```
struct Circulo*
crear_circulo(Length x0,Length y0,Length radio0)
{
  struct Circulo* nuevo_circulo;

  nuevo_circulo = (struct Circulo*)
                  malloc(sizeof(struct Circulo));
  nuevo_circulo -> dc    = &CirculoClass; /* desc. clase */
  nuevo_circulo -> x     = x0;
  nuevo_circulo -> y     = y0;
  nuevo_circulo -> radio = radio0;
  return (nuevo_circulo);
}
```

Resolución de métodos. (VIII)

Visualmente tendríamos esta situación:

Objetos y descriptores de clase



Resolución de métodos. (IX)

- La resolución en tiempo de ejecución de un método se realiza a partir del objeto descriptor de clase al que apunta su campo 'dc', ¿cómo?:
- accedemos al campo del descriptor de clase al que se refiere la operación que queremos realizar así:

Enlace dinámico

```
/* Primero: Ya se han creado los descriptores de clase */

  struct Figura* f;
  struct Circulo* c1 = crear_circulo(...);
  f = c1;

  f->dc->mover(f, ...);    /* Invoca Figura::mover */
  f->dc->dibujar(f, ...); /* Invoca Circulo::dibujar */
```