Tema 11: POO y lenguajes no OO.

¿Es necesario un LOO para hacer POO?

- La utilización del paradigma de *programación orientada a objetos* requiere cierta disciplina por parte del programador y elementos sintácticos por parte del lenguaje empleado.
- Con los lenguajes no orientados a objetos (*LNOO*) esto último *no lo tenemos disponible* pero aún así, sí que es posible hacer POO con un LNOO.
- No vamos a disponer de los elementos que convierten a un lenguaje de programación en un LOO, pero sí que vamos a poder *simularlos* de manera bastante efectiva.
- En este tema explicaremos cómo hacerlo en el lenguaje **C**.

¿Qué podemos simular?

- Representación de clases.
- Paso de mensajes.
- Constructores y destructor de una clase.
- Representación de la herencia simple.
- Resolución de métodos en tiempo de ejecución.

¿Cómo es posible hacerlo?

- Dado que trabajaremos en **C**, vamos a aprovechar la gestión de punteros y la reserva de memoria dinámica de que dispone.
- Esto permitirá que la simulación de las características antes comentadas se lleve a cabo con *poca* o *nula* pérdida de eficiencia.
- De hecho veremos *ejemplos reales* que emplean esta técnica.

- De manera que cada clase del diseño se *convierte* en un `struct` del lenguaje.
- Cada atributo (*datos*) definido en la clase es un campo del `struct` creado.
- Los objetos serán instancias de estos `struct` y podrán estar ubicados en el almacenamiento global, en la pila o en memoria dinámica (preferiblemente en este último).
- Por lo tanto la referencia a un objeto se puede representar mediante un puntero a su `struct`.

Ejemplo de una clase en **C**.

```
typedef float Length;
struct Window {
  Length xmin;
  Length ymin;
  Length xmax;
  Length ymax;
}
...
struct Window* w;
...
Lenght x1 = w->xmin;            /* Mejor emplear setters/getters */
```

Paso de mensajes.

- Debemos seguir un convenio para nombrar a las funciones de manera que las identifiquemos con métodos de una clase: `NombreClase_nombreMetodo`.
- En el caso de constructor y destructor emplearemos: `NombreClase_create` y en el del destructor `NombreClase_destroy`.
- Cada método de instancia recibirá un primer parámetro, lo llamaremos *self*, que representará al objeto receptor del mensaje. Hemos de hacerlo de manera explícita.

```
/************************/
/* Clase: Window        */
/* Metodo: addToSelected */
/************************/
Window_addToSelected (struct Window* self, struct Shape* s);
```

- El paso de parámetros lo hacemos por puntero ya que es más eficiente.
- Como en cualquier función, tenemos accesibles sus parámetros para realizar cualquier operación que queramos con ellos.

Constructores y destructor de una clase.

```
/* Objetos en memoria dinamica */
struct Window* Window_create(Length x, Length y, Length w, Length h) {
  struct Window* ventana;
  ventana = (struct Window*) malloc(sizeof(struct Window));
  ventana -> xmin = x;
  ventana -> ymin = y;
  ventana -> xmax = x + w;
  ventana -> ymax = y + h;
  return ventana;
}
```

```
void Window_destroy (struct Window* self) {
   if (self != NULL) free (self);
}
```

Representación de la herencia simple.

Preliminares.

- La simulación de la herencia simple se basa en una idea muy sencilla:

2. Añadimos a continuación la parte específica de la clase derivada, es decir, los atributos nuevos que añade la clase derivada.

3. Veamos un ejemplo:

Simulando la herencia simple.

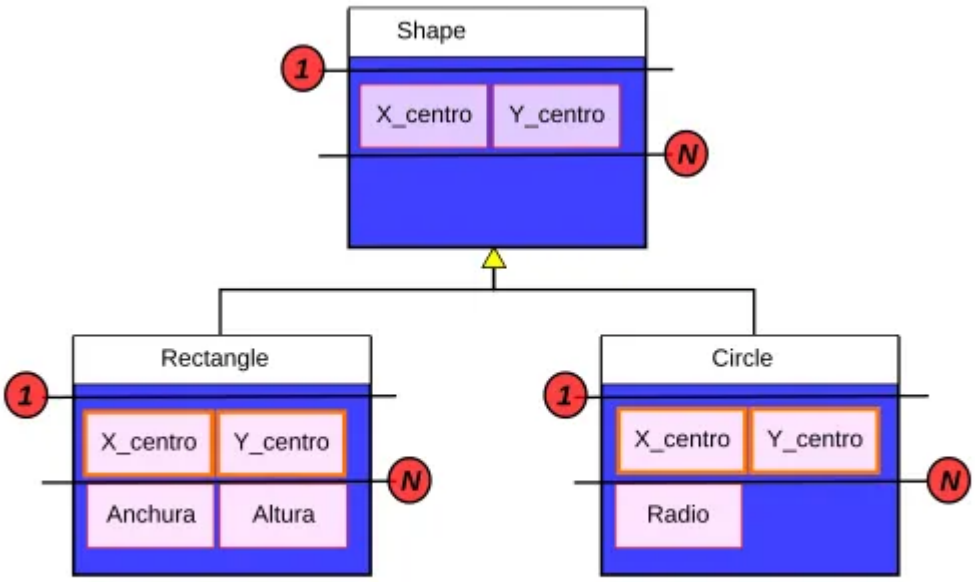

Figura 1: Simulando herencia simple.

Descriptores de clase.

- Para que esta simulación sea completa y podamos añadir posteriormente los *métodos* a la clase, necesitamos añadir un campo más a cada una de las *struct* anteriores. **Lo añadiremos al principio**.

- Se trata de un puntero a su **descriptor de clase**. Se llamará **dc**.

- Los *descriptores de clase* son otro tipo de *struct* que incluyen a los métodos y variables de clase.

- Por tanto el gráfico anterior quedaría así:

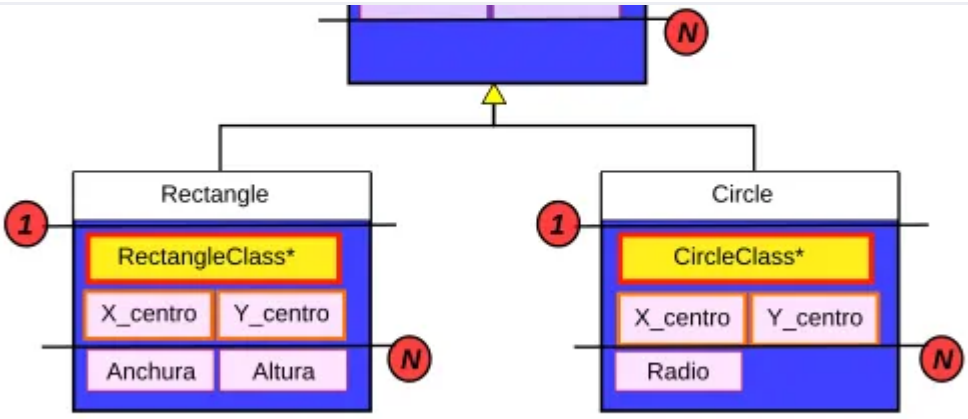

Figura 2: Simulando herencia simple (con descriptores).

- Utilizando esta técnica podemos pasar un puntero a un objeto de clase Rectangulo o Circulo a una función que espere un puntero a un objeto de tipo Figura.
- Así conseguimos que entre dos clases relacionadas por herencia, la representación en memoria de sus primeros *N*-bytes sea igual. Esta es la base de la representación de la relación **Es Un** (*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*:

```
struct Rectangle* r;
struct Window*    w;

/* prototipo de: Window_addToSelected */
void Window_addToSelected (struct Window* self, struct Shape* s);
...
Window_addToSelected (w, r);
```

Resolución dinámica de métodos.

- Se trata de elegir en tiempo de ejecución qué método invocar en respuesta a un mismo mensaje enviado a distintos objetos.

- 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 a la 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**: *ShapeClass*, *CircleClass*, etc...
- Veamos un ejemplo de descriptores de clase:

```
/*********************************/
/* Descriptor de clase para Shape */
/*********************************/
struct ShapeClass {
  char*   classname;
  void    (*move)    ();
  Boolean (*selected)();
  void    (*ungroup) ();
  void    (*draw)    ();
};

/* Descriptor de clase para Circle */
struct CircleClass {
  char*   classname;
  void    (*move)     ();
  Boolean (*selected) ();
  void    (*ungroup)  ();
  void    (*draw)     ();
};

/* Descriptor de clase para Rectangle */
struct RectangleClass {
  char*   classname;
  void    (*move)    ();
  Boolean (*selected)();
  void    (*ungroup) ();
  void    (*draw)    ();
};
```

- 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** (su dirección) definida o heredada por la clase, por ejemplo:

```
struct RectangleClass RectangleClass = {
   "Rectangle",
   Shape_move,         /* void    (*move)    () */
   Rectangle_selected, /* Boolean (*selected)() */
   Shape_ungroup,      /* void    (*ungroup) () */
   Rectangle_draw      /* void    (*draw)    () */
};

struct CircleClass CircleClass = {
   "Circle",
   Shape_move,      /* void    (*move)    () */
   Circle_selected, /* Boolean (*selected)() */
   Shape_ungroup,   /* void    (*ungroup) () */
   Circle_draw      /* void    (*draw)    () */
};
```

- 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 *global* descriptor de la clase.

- De este modo, y en tiempo de ejecución, podemos obtener:

    - El nombre de esta clase.

    - Sus variables de clase.

    - Los métodos asociados a esta clase.

- Por ejemplo, la creación de un objeto de clase *Circle* se haría así:

```
struct Circle*
```

```
    nc -> dc     = &CircleClass; /* descriptor de clase */
    nc -> x      = x0;
    nc -> y      = y0;
    nc -> radius = r;
    return nc;
}
```

- ¿Y si crearámos varios círculos?
- Visualmente podríamos representarlo así:

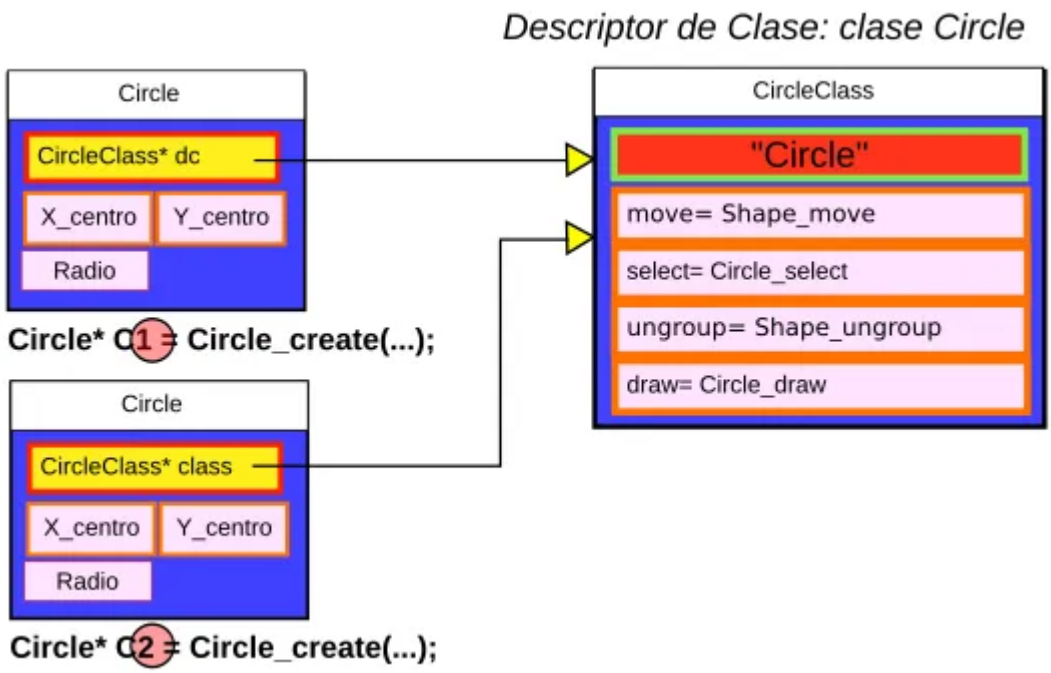

Figura 3: Creación de varios objetos.

- Como vamos a ver, la resolución en tiempo de ejecución de un método para un objeto se realizará a partir del objeto *descriptor de clase* al que apunta su campo **dc**.
- Para ello accedemos al campo del *descriptor de clase* al que se refiere la operación que queremos.
- Si crearamos un *círculo* y le enviáramos mensajes:

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

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

Casos de uso.

- Estas técnicas se emplean (y se amplían) en proyectos software reales.

    1. Uno de esos proyectos es la biblioteca de creación de interfaces de usuario Gtk+. Con ella está construída la interfaz de usuario del *IDE* geany o de nemiver (ambos instalados en la máquina virtual empleada en prácticas).

    2. Veamos algunos de los archivos de cabecera que reflejan su jerarquía de clases.

- También es la base del código **C** generado por el compilador de **Vala** , se puede ver con este sencillo ejemplo:

```
// Compilar con valac -C valaoop.vala

// Clase base
public class Droid {
        public Droid (string n) {
          name = n;
        }

        public string name {get; set;}
        public virtual void move (int x, int y) {
                this.x = x;
                this.y = y;
        }

        protected int x;
        protected int y;
}
// Clase derivada
public class AquaDroid : Droid {
        public AquaDroid(string n, int md = 100) {
```

```
        public override void move (int x, int y) {
                this.x = x/2;
                this.y = y/2;
        }

        private int depth;
}
```

Aclaraciones.

- **Este contenido no es la bibliografía completa de la asignatura**, por lo tanto debes estudiar, aclarar y ampliar los conceptos que en ellas encuentres empleando los enlaces web y bibliografía recomendada que puedes consultar en la página web de la ficha de la asignatura y en la web propia de la asignatura.