

## IV.1. Registros

### IV.1.1. El tipo de dato struct

Un *registro* o *struct* permite almacenar varios elementos de (posiblemente) distintos tipos y gestionarlos como uno sólo. Cada uno de los datos agrupados en un *struct* es un *campo*.

Los campos de un *struct* se suponen relacionados, que hacen referencia a una misma entidad. Cada campo tiene un *nombre*.

Un ejemplo típico es la representación en memoria de un *punto* en un espacio bidimensional. Un punto está caracterizado por dos valores: abscisa y ordenada. En este caso, tanto abscisa como ordenada son del mismo tipo, supongamos que sea *int*.

```
struct Punto2D {  
    double abscisa;  
    double ordenada;  
};
```

Se ha definido un tipo de dato registro llamado *Punto2D*. Los datos de este tipo están formados por dos campos llamados *abscisa* y *ordenada*, ambos de tipo *double*.

Cuando se declara un dato de este tipo:

```
Punto2D un_punto;
```

se crea una variable llamada *punto*, y a través de su nombre se puede acceder a los campos que la conforman mediante el operador punto (.). Por ejemplo, podemos establecer los valores de la abscisa y ordenada de *un\_punto* a 4 y 6 respectivamente con las instrucciones:

```
un_punto.abscisa = 4.0;  
un_punto.ordenada = 6.0;
```

**Es posible declarar variables struct junto a la definición del tipo (aunque normalmente lo haremos de forma separada):**

```
struct Punto2D {  
    double abscisa;  
    double ordenada;  
} un_punto, otro_punto;
```

**Un ejemplo de struct heterogéneo:**

```
struct Persona {  
    string nombre;  
    string apellidos;  
    string NIF;  
    char categoria;  
    double salario_bruto;  
};
```

**No se supone ningún orden establecido entre los campos de un struct y no se impone ningún orden de acceso a éstos (por ejemplo, puede iniciarse el tercer campo antes del primero).**

**No puede leerse/escribirse un struct, sino a través de cada uno de sus campos, separadamente. Por ejemplo, para leer los valores de un\_punto desde la entrada estándar:**

```
cin >> un_punto.abscisa;  
cin >> un_punto.ordenada;
```

**y para escribirlos en la salida estándar:**

```
cout << "(" << un_punto.abscisa << ", " << un_punto.ordenada << ")";
```

**Es posible asignar un struct a otro y en consecuencia, un struct puede aparecer tanto como lvalue y rvalue en una asignación.**

```
otro_punto = un_punto;
```

**Los campos de un struct pueden emplearse en cualquier expresión:**

```
dist_x = abs(un_punto.abscisa - otro_punto.abscisa);
```

**siendo, en este ejemplo, un\_punto.abscisa y otro\_punto.abscisa de tipo double.**

## IV.1.2. struct y funciones

**Un dato struct puede pasarse como parámetro a una función y una función puede devolver un struct.**

**Por ejemplo, la siguiente función recibe dos struct y devuelve otro.**

```
Punto2D PuntoMedio (Punto2D punto1, Punto2D punto2){  
    Punto2D punto_medio;  
  
    punto_medio.abscisa = (punto1.abscisa + punto2.abscisa) / 2;  
    punto_medio.ordenada = (punto1.ordenada + punto2.ordenada) / 2;  
  
    return (punto_medio);  
}
```

### IV.1.3. Ámbito de un struct

Una variable **struct** es una variable más y su ámbito es de cualquier variable:

- ▷ Puede ser una variable global, aunque ya sabe que el uso de variables globales no está permitido en esta asignatura.
- ▷ Como cualquier variable local a una función, puede usarse desde su declaración hasta el fin de la función en que ha sido declarada. Por ejemplo, en la función `PuntoMedio` la variable `punto_medio` es una variable local y se comporta como cualquier otra variable local, independientemente de que sea un **struct**.
- ▷ El ámbito más reducido es el nivel de bloque: si el **struct** se declarara dentro de un bloque sólo podría usarse dentro de él. La variable `punto_medio` es una variable local de la función `PuntoMedio` y se comporta como cualquier otra variable local, independientemente de que sea un **struct**. Lo mismo podría decirse si el **struct** se declarara dentro de un bloque: sólo podría usarse dentro de ese bloque.

.....

```
while (hay_datos_por_procesar) {
    Punto2D punto;

    // punto sólo es accesible dentro del ciclo while.
    // Se crea un struct nuevo en cada iteración y "desaparece"
    // el de la anterior.
    // Evítelo por la recarga computacional ocasionada
}
```

## IV.1.4. Inicialización de los campos de un struct

Un struct puede inicializarse en el momento de su declaración. El objetivo es asignar un valor inicial a sus campos evitando que pueda usarse con valores basura.

El procedimiento es muy simple, y debe tenerse en cuenta el orden en que fueron declarados los campos del struct. Los valores iniciales se especifican entre llaves, separados por comas.

Por ejemplo, para inicializar un struct Persona podríamos escribir:

```
Persona una_persona = {"", "", "", 'A', 0.0};
```

que asigna los valores **cadena vacía** a los campos de tipo string (nombre, apellidos y NIF), el carácter A al campo categoria y el valor 0.0 al campo salario\_bruto.