Structs (registros)

Structs (registros)

Con los arrays podemos agrupar información homogénea (del mismo tipo) de una forma eficiente y fácil de manejar
. . . y si la información es heterogénea (de distinto tipo)?

Los registros (structs) sirven precisamente para esto:
$\rightsquigarrow$ permiten agrupar valores individuales de tipos distintos y tratar el bloque como una entidad simple: a través de una única variable.

Por ejemplo, para agrupar la información referente a un coche:

```
using System;
namespace ... {
  class MainClass {

    struct Coche {
      public string marca, modelo, color;
      public string matricula;
      public int año_compra;
      public long num_bastidor;
      public int potencia;
      public char carburante; // 'g': gasolina, 'd': diesel, 'e': eléctrico, 'h': híbrido
      public long dni_propietario;
      public char letra_dni;
    };

    public static void Main (string[] args) {
      ...
```

La sintaxis para definir un registro es:

```
struct <<nombre_struct>> {
  <<declaracion de campos>>
}
```

- Esta definición se hace fuera de los métodos (pero dentro de la clase)
- La declaración de los campos del registro es una secuencia de declaraciones de variables convencionales.
- El tipo de los campos puede ser cualquier tipo válido (predefinido o definido por el programador). Utilizaremos el modificador `public` para que sean accesibles fuera del propio `struct`.

Nota:

- Los `structs` son tipos *normales*, tipos valor (no son tipos referencia como los arrays) y no necesitan creación (new).

Structs: declaración de variables del tipo

Al definir un `struct` hemos creado un nuevo tipo en nuestro programa (un *tipo definido por el programador*) $\rightsquigarrow$ podemos declarar variables de ese tipo, igual que `int`, `char` o `bool`.

Por ejemplo, podemos declarar:

```
public static void Main () {
  Coche un_coche;
  ...
```

`un_coche` es una variable de tipo `Coche`, que agrupa un conjunto de valores (campos): virtualmente hemos declarado 4 variables de tipo `string`, 2 `int`, 2 `long` y 2 `char` de golpe!

El acceso a cada campo individual del registro se hace mediante el operador de acceso "." Si `r` es una variable de tipo registro y `c` uno de sus campos, la expresion `r.c` representa ese campo `c`. Por ejemplo, se puede hacer:

```
un_coche.marca = "honda";
un_coche.modelo = "civic";
un_coche.color = "amarillo fosforito";
un_coche.matricula = "...";
```

- Las variables de tipo registro (`struct`) no pueden leerse ni escribirse directamente: hay que procesar los campos de uno en uno.
- Pero sí pueden pasarse como parámetros a los métodos (por valor, `ref`, `out`)
- Los campos individuales también pueden pasarse como parámetros a los métodos: se comportan a todos los efectos como variables simples del tipo correspondiente.

Structs: paso de parámetros

Las variables de tipo registro pueden pasar a los métodos como cualquier otra variable de tipo simple (por valor, `ref`, `out`).

Un método para leer los campos de una variable de tipo `Coche`:

```
static void leeCoche(out Coche c){
  Console.WriteLine("Introduzca información del coche: ");

  Console.Write ("Marca: ");
  c.marca = Console.ReadLine ();

  Console.Write ("Año de compra: ");
  c.año_compra = int.Parse(Console.ReadLine ());

  Console.Write ("Num bastidor: ");
  c.num_bastidor = long.Parse(Console.ReadLine ());

  Console.Write ("Carburante: ");
  c.carburante = char.Parse(Console.ReadLine ());
   ...
}
```

Nota: la declaración `out` exige asignación para todos los campos del registro.

Structs: paso de parámetros

Un método para escribir la información de un coche:

```
static void escribeCoche(Coche c){
  Console.WriteLine("Información del coche: ");

  Console.WriteLine ("Marca: " + c.marca);
  Console.WriteLine ("Modelo: " + c.modelo);
  Console.WriteLine ("Año de compra: " + c.año_compra);
  Console.WriteLine ("Num bastidor: " + c.num_bastidor);

  Console.Write ("Carburante: ");
  switch (c.carburante) {
    case 'g':
      Console.WriteLine ("gasolina");
      break;
    case 'd':
      Console.WriteLine ("diesel");
      break;
    case 'e':
      Console.WriteLine ("electrico");
      break;
    case 'h':
      Console.WriteLine ("hibrido");
      break;
    default:
      Console.WriteLine ("carburante no reconocido");
      break;
    }
     ...
}
```


Al igual que ocurre con las componente de un array, cada campo concreto de un registro se comporta a todos los efectos como una variable del tipo correspondiente.

Método para cambiar un campo concreto del registro:

```
static void cambiaColor(out string col, string nuevo){
   col = nuevo;
}

// llamada
cambiaColor (out un_coche.color, "rojo");
```

También podemos rejuvenecer el coche quitándole años (parámetro de entrada salida: paso por referencia):

```
static void quitaAños(ref int x){
   x = x+5;
}

// llamada
quitaAños (ref un_coche.año_compra);
```

Lo más natural para cambiar un campo de un registro es pasar el registro entero en vez del campo concreto, i.e., tratar el coche como una unidad:

```
static void pintaCoche(Coche c, string nuevoCol){
  c.color = nuevoCol;
}
// llamada
pintaCoche(un_coche, "rojo");
```

por qué está mal? ⇝ se modifica el color del coche:

```
static void pintaCoche(out Coche c, string nuevoCol){
  c.color = nuevoCol;
}
```

sigue estando mal! ⇝ no se modifican los demás campos: el coche, como unidad (como registro) se modifica, pero hay campos que no cambian y quedan con su valor de entrada ⇝ es un parámetro de entra/salida:

```
static void pintaCoche(ref Coche c, string nuevoCol){
  c.color = nuevoCol;
}
```


Números complejos

Registro con dos campos:

```
struct Complejo {
  public double re, im;
}
```

Lectura y escritura:

```
static void leeComplejo(out Complejo c){
  Console.Write("Parte real: ");
  c.re = double.Parse (Console.ReadLine ());

  Console.Write ("Parte imaginaria: ");
  c.im = double.Parse (Console.ReadLine ());
}

static void escribeComplejo(Complejo c){
  Console.Write(c.re);
  if (c.im > 0)
     Console.Write (" + {0}i", c.im);
  else if (c.im<0)
     Console.Write (" - {0}i", Math.Abs(c.im));
}
```

Suma de complejos:

```
static void sumaComplejos(Complejo c1,Complejo c2,out Complejo c3){
  c3.re = c1.re+c2.re;
  c3.im = c1.im+c2.im;
}
```

Otra forma:

```
static Complejo sumaComplejos(Complejo c1,Complejo c2){
  Complejo c3;
  c3.re = c1.re+c2.re;
  c3.im = c1.im+c2.im;
  return c3;
}
```

Resta, multiplicación, conjugado... $\rightsquigarrow$ creamos un nuevo tipo numérico con su representación y sus operaciones (se puede hacer todavía más *integrada* utilizando clases).

Anidando tipos estructurados

Hemos dicho:

- las componentes de un array pueden ser de cualquier tipo válido (predefinido o definido por el programador).
- los campo de un registro pueden ser de cualquier tipo válido (predefinido o definido por el programador).

Entonces... podemos definir un array de registros o un registro que contenga campos de tipo array:

- Se puede definir un array de componentes de tipo `Coche`
- o incluir en el registro `Coche` un campo `propietarios` de tipo array que contenga el historial de propietarios del coche
  - que a su vez pueden definirse como un registro con `Nombre, Apellido1, Apellido2, DNI,...`

Podemos anidar arrays y registros sin límite $\rightsquigarrow$ de este modo podemos dar una representación estructurada para cualquier tipo de información.

Ejemplo: representación y manipulación de polinomios

Queremos escribir un programa para trabajar con polinomios que incluya las operaciones típicas: suma, resta, multiplicación, división, evaluación en un valor dado, lectura, escritura...

Antes de nada: ¿qué representación elegimos?

Primera idea: array de reales, donde el contenido de la componente i-ésima representa el coeficiente de x^i. Por ejemplo: el polinomio

$$3x^4 - 5.23x^2 + 6x - 7.9$$

se representará como:

0	1	2	3	4	5	6	7	8	9	10
−7.9	6.0	−5.23	0.0	3.0	0.0	0.0	0.0	0.0	0.0	0.0

Esta representación es sencilla ... pero

- el grado del polinomio está limitado por el tamaño del array (de qué tamaño declaramos el vector)
- puede hacer muy mal aprovechamiento de la memoria: para representar el polinomio $2x^{3456}$ se necesita un vector de 3456 componentes ... de las cuales sólo se utiliza realmente una!!

Representación alternativa?

- Definimos el tipo monomio como un registro con dos componentes: coeficiente y exponente
- Un polinomio será un array de monomios ... de qué tamaño?
  - El tamaño puede definirse en el momento de crear el polinomio conociendo el número de monomios.
  - O bien puede fijarse de antemano constante `N` y *llevar cuenta del tamaño* `tam` *del polinomio (número de monomios)*

Definimos el tipo `Monomio` y `Polinomio`:

```
const int N = 100; // tamaño del vector de monomios

struct Monomio {
  public double coef;
  public int exp;
}

struct Polinomio {
  public Monomio [] mon; // array de monomios
  public int tam; // número de monomios
                  // último monomio en tam-1
}
```


Lectura de polinomios:

```
static void leeMonomio(out Monomio m){
  Console.Write ("Coeficiente: ");
  m.coef = double.Parse (Console.ReadLine ());
  Console.Write ("Exponente: ");
  m.exp = int.Parse (Console.ReadLine ());
}


static void leePolinomio(out Polinomio p){
  p.mon = new Monomio[N];

  Console.Write ("Número de monomios: ");
  p.tam = int.Parse (Console.ReadLine ());

  Console.WriteLine ("Introduce monomios");
  for (int i=0; i<p.tam; i++)
    leeMonomio (out p.mon[i]);
}
```

Escritura de polinomios:

```
static void escribePolinomio(Polinomio p){
  for (int i = 0; i < p.tam; i++)
    Console.Write (" + " + p.mon [i].coef + "x^" + p.mon [i].exp);
}
```

Esta escritura se puede mejorar mucho: eliminando signos "+" innecesarios, exponentes 0 ó 1, etc

La propia representación de los polinomios es muy mejorable:

- Invariante de la representación 1: unicidad de exponentes en vector de monomios, para que no aparezca más de un monomio del mismo grado).
- Invariante de la representación 2: que no aparezca ningún monomio con coeficiente 0.

Ejercicio: mejorar la representación con estas ideas e implementar las operaciones básicas de suma, resta, multiplicación, evaluación en un valor, etc

Tipos enumerados

Tipos enumerados

¿Cómo podemos representar los días de la semana en C#?

Posibles soluciones:

Variable de tipo `string` con los valores

`"lunes"`, `"martes"`,..., `"domingo"`

Inconvenientes de esta solución:

- tenemos que definir el *siguiente* explícitamente (siguiente a `"lunes"` es `"martes"`...).
- es fácil cometer errores e introducir *valores no previstos* (por ejemplo, `"miecoles"`) que producirán después errores "incómodos".

Otra representación?

Otra alternativa: variable de tipo entero, de modo que 0 represente *lunes*, 1 *martes*,..., 6 *domingo*.
(O también numerando de 1 a 7: es viable, pero es preferible empezar en 0 para facilitar aritmética modular (como veremos) o para indexar en vectores).

Mejora las cosas?

- Las operaciones *siguiente* y *anterior* son más fáciles, básicamente sumar o restar 1, con la salvedad de que después del 6 va el 0, y antes del 0 va el 6

$$next = (day + 1)\%7 \qquad prev = (day + 6)\%7$$

- Es más fácil controlar los valores no previstos ($0 \leq day \leq 6$), pero nada impide $day = 17$
- Tenemos que *memorizar* esta codificación a lo largo del programa (y comentarla adecuadamente)

Otra alternativa?

Estaría bien poder definir un tipo simple que tuviese exactamente los valores `lunes`, `martes`, ... `domingo` ... que además incluya las operaciones de *anterior* y *siguiente*

El tipo que queremos es un tipo enumerado y es inmediato definirlo en C#:

```
enum Dias { Lunes, Martes, Miercoles, Jueves, Viernes, Sabado, Domingo};
```

Ahora podemos declarar variables, escribir y leer su valor:

```
// declaración
Dias d;
// asignación
d = Dias.Domingo;
// escribir en pantalla (otros lenguajes no lo admiten)
Console.WriteLine (d);

// leer de teclado. Ojo a la info de tipos incluida
// (otros lenguajes no admiten lectura directamente)
Didas d2 = (Dias) Enum.Parse (typeof(Dias),Console.ReadLine ());
```


También admiten los operadores de comparación (valores ordenados):

Se puede hacer:

```
static bool finde(Dias d){
  return d>=Dias.Sabado;
}
```

Y además

Anterior y siguiente (incremento y decremento):

```
// incremento circular
d = Dias.Domingo;
d++;
Console.WriteLine (d); // escribe Lunes


// recorrido de valores del tipo
foreach (Dias d2 in Enum.GetValues(typeof(Dias)))
  Console.WriteLine (d2);
```

Podríamos hacer lo análogo con los meses del año.

Internamente C# representa los enumerados con valores numéricos... pero podemos abstraernos de ello.

Uso de enumerados

En general son muy útiles para almacenar información de estado. Por ejemplo:

- Para el juego de hundir la flota, las casillas del tablero pueden tener los estados: *Libre, Agua, Buque, Tocado, Hundido, No_valido,...*
- Estado de un semáforo: *Verde, Rojo, Ámbar, AmbarIntermintente, Estropeado*
- Tamaño de la hamburguesa: *Extra, Grande, Mediana, Pequeña*
- Palos de la baraja española: *Oros, Copas, Espadas, Bastos*
- ...