5. Tipos de datos estructurados

Agrupando datos

Los tipos que conocemos (enteros, reales, `bool`, `char`, `string`) son tipos simples: una variable de ese tipo almacena un único valor de ese tipo.

Pero en muchos problemas es adecuado agrupar varios valores en una misma estructura (referenciada por una variable), i.e., una misma variable almacena varios valores!

La información a agrupar puede ser:

- Homogénea (del mismo tipo). Por ejemplo: temperaturas mínimas alcanzadas en una población a lo largo de una semana $\rightsquigarrow$ agrupamiento de valores de tipo `double`'s
  - Arrays (vectores o matrices)
- Heterogénea (de distinto tipo). Por ejemplo: datos de una persona (nombre, teléfono, sexo, dirección, edad, ...) $\rightsquigarrow$ agrupamiento de valores de tipo `string`'s, `char`'s, `int`'s, ...
  - Structs (estructuras o registros)

Arrays

Arrays: un primer ejemplo

Ejemplo: temperaturas mínimas alcanzadas en una población a lo largo de una semana.

Para leer esas temperaturas podríamos utilizar 7 variables:

```
double temp1, temp2, temp3, temp4, temp5, temp6, temp7;
temp1 = pideDato("Temperatura: ", -20, 50); // método del cap. anterior
temp2 = pideDato("Temperatura: ", -20, 50);
 ...  // temp3, temp4, temp5, temp6
temp7 = pideDato("Temperatura: ", -20, 50);
```

Con un array podemos hacer:

```
double [ ] temp;       // array de eltos de tipo double
temp = new double[7]; // con 7 elementos

for (int i=0; i<7; i++) // el elemento i-esimo se referencia como temp[i]
  temp[i] = pideDato("Temperatura: ", -20, 50);
```

Se compacta y clarifica el código. Esencialmente es como un vector de 7 elementos $(temp_0, \ldots, temp_6)$ (ojo, empieza en 0!)

Un array es una estructura de almacenamiento de valores del mismo tipo.

Con una sola variable podemos:

- referenciar todo el conjunto de valores
  (`temp` representa el conjunto de temperaturas de la semana)

- y también referenciar elementos individuales
  (`temp[0]`: temperatura del día 1,
  `temp[1]` temperatura del día 2,
  ...,
  `temp[6]` temperatura del día 7)

Arrays: declaración, creación, inicialización

Tres pasos:

- Declaración: `double [ ] temp;`
  Los corchetes `[ ]` indican que `temp` es una variable de tipo *array de...* en este caso de elementos de tipo `double`

- Creación: `temp = new double[7];`
  Se crea (se solicita hueco en memoria) para albergar 7 elementos de tipo *double* en la variable `temp`.

- Inicialización:

  `temp[i]=«expresión de tipo double»`

  para cada índice `i` del conjunto 0,...,6
  A cada una de las componentes del vector (o elementos del vector) se le asigna un valor.

Cada componente `temp[0],...,temp[6]` se comporta a todos los efectos como una variable de tipo `double`.

Arrays: declaración+creación+inicialización

Es posible compactar la declaración y la creación en una sola línea:

```
double[] temp = new double[7];
```

Declarara un array `temp` de (elementos de) tipo `double` y lo crea con 7 "huecos".

Es posible incluso declararlo e inicializarlo a la vez (la creación queda implícita):

```
double[] temp = {2.5, 3.8, 4, 5, 9, 4, 2.3};   // new double[7] queda implícito
```

- En este caso el tamaño (7) queda determinado por el número de elementos con que se inicializa.
- Esto es similar a la *declaración+inicialización* de una variable convencional.

Arrays: declaración y creación

En general, la declaración de un array se hace como:

`«tipo definido» [ ] «identificador»;`

donde `«tipo definido»` puede ser cualquier tipo válido (`int`, `long`, `double`, `float`, `string`, `char`, `bool`,....

Es decir, podemos declarar:

- arrays de cualquier cosa,
- pero con todos los elementos del mismo tipo (definido)

La creación se hará de la forma:

`«identificador» = new «tipo»[«entero»];`

Crea un array de tamaño `«entero»` (reserva hueco para `«entero»` valores de tipo `«tipo»`).

Componentes de un vector

Si definimos:

```
«tipo» [ ] a;
```

cada componente `a[i]` se comporta a todos los efectos como una variable de tipo `«tipo»`, i.e., admite todas las operaciones de dicho tipo.

Por ejemplo, si hacemos;

```
int [] vec = new int[10];
```

`vec[6]` corresponde al elemento 7 del array y se comporta a todos los efectos como una variable de tipo `int`: se le puede asignar un valor entero, comparar con otro entero, escribir en pantalla, incrementar, leer su valor de teclado, etc. (Se puede incluso, pasar como parámetro a un método que espere un `int`... volveremos sobre ello).

Teniendo:

```
int [] vec = new int[10];
```

Qué pasa si hacemos:

- `vec[-1], vec[11], vec[10]` no tienen sentido ... pero el programa compila!!
  - No da ningún error en tiempo de compilación
  - Pero da un error en tiempo de ejecución:
    `Unhandled Exception:`
    `System.IndexOutOfRangeException: Index was outside the bounds of the array. at arrays.MainClass.Main () ...`
    `...`
- `vec[3.2] = 4;`? ⇝ error de tipo en tiempo de compilación
- `vec[(6+8) % 3] = 8;` qué error da? ninguno!
- `vec[(6+8) % 7] = 3.2;` qué error da? error de tipo en tiempo de compilación
- y `vec[ vec[1] ]`? depende. Ninguno en tiempo de compilación. En ejecución, depende de si el valor de `v[1]` está en el rango 0..9

Usando arrays

Recogida de temperaturas de una semana y cálculo de la media:

```
double[] temp = new double[7];
double suma;

// pide temperatura de cada día (indicando el día):
for (int i=0; i<7; i++)
  temp[i] = pideDato("Temperatura del día "+ (i+1), -20, 50);

suma = 0; // acumular de suma
for (int i=0; i<7; i++)
  suma += temp [i];

Console.Write ("Media: " + suma/7);
```

por qué esta mal? está bien!

```
Temperatura del día 1 [-20,50]: 21,5
Temperatura del día 2 [-20,50]: 22,4
...
Temperatura del día 7 [-20,50]: 20,3
Media: 21,0428571428571
```


Temperatura máxima y mínima

Para la máxima:

```
double max = -100; // inicializamos con valor fuera del rango
for (int i = 0; i < 7; i++)
  if (temp [i] > max) max = temp [i];
  else  max = max;
```

Es correcto? Sí... pero, cuántas malas ideas hay aquí?

- la asignación `max = max;` denota un pésimo estilo de programación
- ahora, para calcular la mínima hacemos otro bucle semejante? $\rightsquigarrow$ mejor aprovechar el mismo recorrido
- también puede ser buena idea inicializar `max = temp[0]` y luego arrancar el bucle desde `i=1`

```
double max = min = temp [0];
for (int i = 1; i < 7; i++) {
  if (temp [i] > max) max = temp [i];
  if (temp [i] < min) min = temp [i];
}
```

Arrays de strings

Podemos mejorar la interfaz de usuario del programa anterior fácilmente con un array de strings:

```
double[] temp = new double[7];
string[] diaSemana = {"lunes", "martes", "miércoles", "jueves", "viernes",
                      "sábado", "domingo"};

for (int i=0; i<7; i++)
  temp[i] = pideDato("Temperatura del "+ diaSemana[i], -20, 50);
```

```
Temperatura del lunes [-20,50]: 12
Temperatura del martes [-20,50]: 15
Temperatura del miércoles [-20,50]: 16
Temperatura del jueves [-20,50]: 1
Temperatura del viernes [-20,50]: 8
Temperatura del sábado [-20,50]: 17
Temperatura del domingo [-20,50]: 21
```

Este vector con los nombre de los días de la semana se puede reutilizar cada vez que queramos escribir un día de la semana.

Ejemplos

En los siguientes ejemplos consideramos el vector `v` declarado como:

```
int [] v = new int[10];
```

Inicializar `v` con los valores 0,1,...,9:

```
for (int i=0; i<10; i++)
  v[i] = i;
```

Inicializarlo con los valores en orden inverso 9,8,...,0:

```
for (int i=9; i>0; i--)
  v[i] = i;
```

Por qué está mal? ...porque sigue inicializando en orden ascendente y además no alcanza la componente 0 la condición de parada debería ser `i>=0`. Corregimos:

```
for (int i=0; i<10; i++)
  v[i] = 10-i-1;  // 9-i
```


Está bien ahora? sí

Contar el número de positivos, negativos y de ceros de `v`:

```
int pos, neg, ceros;
pos = neg = ceros = 0;

for (int i=0; i<10; i++)
  if (v[i]<0) neg++;
  else if if (v[i]>0) pos++;
  else ceros++;
```

Para el mismo vector, determinar si todos sus elementos son positivos:

```
bool todosPos = true;

for (int i=0; i<10; i++)
  if (v[i]>0) todosPos = true;
  else todosPos = false;
```

Incorrecto! Si `v[9]>0`, quedará `todosPos = true` aunque todos los demás sean negativos!
Corregimos...


```
bool todosPos = true;

for (int i=0; i<10; i++)
  if (v[i]<=0) todosPos = false
```

Correcto? Sí ... eficiente? mejorable?

Qué pasa si el vector tiene 10^6 componentes y es de la forma `[-1,...]`? $\rightsquigarrow$ mejoramos al algoritmo con un *while*:

```
bool todosPos;
int i = 0;
// OJO: orden de las condiciones para no salirnos del rango del vector
while (i<10 && v[i]>0) i++;

// si sale con i=10 -> todos positivos; si no (i<10) no todos positivos
if (i<10) todosPos = false;
else todosPos = true;
```

Este patrón de búsqueda es muy utilizado (aprender bien!!)

- Es crítico el orden de las condiciones en el *while* (evaluación de circuito corto o perezosa): primero la comprobación de rango `i<10` (después resto de condiciones)

Determinar si el vector `v` está ordenado en orden creciente?

bucle *for* o *while*? hay que recorrer el vector entero?
no necesariamente (paramos en cuanto detectemos dos elementos contiguos desordenados) $\rightsquigarrow$ `while`

```
bool ordenado;
// i recorre 1..9
int i = 1;

// comparamos v[0]<=v[1], luego v[1]<=v[2] ... hasta v[8]<=v[9]
// vital el orden de las condiciones!
while (i<10 && v[i-1]<=v[i]) i++;
// si sale con i<10 es porque v[i-1]>v[i] para algún i -> no ordenado

if (i<10) ordenado=false;   // if (i==10) ordenado=true;
else ordenado = true;       // else ordenado = false;
```


Tabla de frecuencias de notas de alumnos

```
Random rnd = new Random();

const int NUM_ALS = 10;
float [] notas = new float[NUM_ALS];

// tabla de frecs. tabla_frec[0]: contador de SS... tabla_frec[4]: contador de MH
int[] tabla_frec = {0,0,0,0,0}; // contadores inicializados a 0

// inicialización con valores aleatorios entre 0 y 10 con un decimal
for (int i = 0; i < NUM_ALS; i++) notas[i] = rnd.Next(0, 101) / 10f;
// vista del array de notas
for (int i = 0; i < NUM_ALS; i++) Console.Write(notas[i] + " ");

// construcción de la tabla de frecuencias
for (int i=0; i<NUM_ALS; i++)
  if (notas[i]<5) tabla_frec[0]++;        // SS
  else if (notas[i]<6.5) tabla_frec[1]++; // AP
  else if (notas[i]<8.5) tabla_frec[2]++; // NT
  else if (notas[i]<10) tabla_frec[3]++;  // SB
  else tabla_frec[4]++;                   // 10, MH

// Aprobados totales (excluyenso suspensos)
int aps = 0;
for (int i = 1; i < 5; i++) aps += tabla_frec [i];

Console.Write("\n\nResumen de resultados: \n");
Console.Write("SS: " + tabla_frec[0] + "\n");
Console.Write("AP: " + tabla_frec[1] + "\n");
Console.Write("NT: " + tabla_frec[2] + "\n");
Console.Write("SB: " + tabla_frec[3] + "\n");
Console.Write("MH: " + tabla_frec[4] + "\n\n");
Console.Write("Aprobados totales: " + aps + "\n");
```

Ejemplo de ejecución:

```
7,3   0,7   7,1   4,6   1,9   3,1   3,5   8,9   0,7   7

Resumen de resultados:
SS: 6
AP: 0
NT: 3
SB: 1
MH: 0

Aprobados totales: 4
```


Arrays: en memoria

```
double[] temp = new double[7];
```

Esta declaración es equivalente a declarar 7 variables de tipo `double` (`temp[0]`,...,`temp[6]`) que se almacenan en posiciones contiguas en memoria:

temp[0]	temp[1]	temp[2]	temp[3]	temp[4]	temp[5]	temp[6]
...	...	...	...	...	...	...

O también:

	0	1	2	3	4	5	6
temp	...	...	...	...	...	...	...

Cuánto ocupa este array en memoria? 7*tamaño(`double`)
En general: *numElems* * tamaño(*tipo*)

Por qué es importante que las posiciones de las componentes de un array sean contiguas en memoria?

$\rightsquigarrow$ porque permite referenciar el array completo con una sola dirección de memoria (la de la variable que lo referencia).

Esto implica:

- El acceso a los elementos de un vector, internamente se hace con una sencilla operación aritmética (dirección base del vector + componente accedida) en tiempo constante $\rightsquigarrow$ estructura de acceso directo (en contraposición al acceso secuencial, más ineficiente)
- El paso de un array como parámetro a un método implica el paso de una sola dirección de memoria!

$\rightsquigarrow$ la estructura `array` es muy potente, muy cómoda de manejar y muy eficiente. Se utiliza constantemente en programación.

Arrays y paso de parámetros

En C# los arrays son tipos referencia: a todos los efectos las componentes de un array siempre pasan por referencia a los métodos de modo implícito (sin `ref`), i.e., todas las modificaciones que se hagan a una componente tendrán efecto fuera del método en cuestión.

Por qué?

- Es una decisión de diseño (adoptada por la muchos lenguajes) para mejorar el rendimiento en memoria y en tiempo de ejecución.
- Si el paso de array fuese por valor (por copia) habría que hacer una copia suya en el registro de activación, i.e., duplicar el array en memoria:
  - Consumo de memoria
  - Consumo de tiempo

Si en algún momento se necesita una copia de un array, hay que hacerla explícitamente (conscientes del coste en tiempo y memoria que supone; hay métodos predefinidos para hacerlo).

Arrays y paso de parámetros. Ejemplos

Método que escribe todas las componentes de un vector de enteros

```
static void escribeVector(int [] v){
  for (int i = 0; i < v.Length; i++)
    Console.Write (v[i] + " ");
}
```

Nótese:

- El parámetro del método: `int [] v` (array de enteros)
- La expresión `v.Lenght` que devuelve el tamaño del array (muy útil!!)

Arrays y paso de parámetros. Ejemplos

Método que dado un array obtiene otro con las componentes en orden inverso (supuestos ambos del mismo tamaño):

```
static void reverse(int[] v, int[] w){
  for (int i=0; i<v.Length; i++)
    w[i] = v[v.Length-i-1]; // w[0] = v[len-1], w[1] = v[len-2] ...
}

public static void Main () {
  int[] v = {0,1,2,3,4,5,6,7,8,9};
  int[] w = new int[v.Length];

  reverse (v, w);
  escribeVector(w);
  ...
```

En `reverse`:

- los arrays `v` y `w` pasan por referencia.
- $\rightsquigarrow$ todos los cambios dentro del método modifican los vectores (fuera del método)

```
9 8 7 6 5 4 3 2 1 0
```

Arrays y paso de parámetros. Ejemplos

Método que dado un array invierte el orden de sus elementos (sin vector auxiliar):

```
static void reverse2(int [] v){
  int len = v.Length;
  for (int i=0; i<len; i++) {
    int tmp = v[i];
    v[i] = v[len-i-1];
    v[len-i-1] = tmp;
  } }

public static void Main () {
  int[] v = {0,1,2,3,4,5,6,7,8,9};
  reverse2 (v);
  escribeVector(v);
```

Por qué está mal? Escribe: `0 1 2 3 4 5 6 7 8 9` No invierte! Trazar la ejecución. Corrección en el bucle:

```
  for (int i=0; i<len/2; i++) { ...
```

Es correcto?

Arrays y paso de parámetros. Ejemplos

Las componentes de un vector se comportan a todos los efectos como variables del tipo correspondiente $\rightsquigarrow$ pueden ser de entrada, salida (`out`) o entrada-salida (`ref`).
Utilizando `swap` en el método `reverse`:

```
static void reverse3(int [] v){
  int len = v.Length;
  for (int i=0; i<len/2; i++)
    swap (ref v[i], ref v[len-i-1]);
}

static void swap(ref int i, ref int j){
  int tmp = i;
  i = j;
  j = tmp;
}
```

Está bien? Sí

- Las componentes del vector pasan por referencia a `reverse3` (implícitamente, como siempre)
- Las componentes `v[i]` y `v[len-i-1]` pasan por referencia explícitamente a `swap`

Arrays bidimensionales

Podemos crear arrays de cualquier tipo definido... y además con cualquier número de dimensiones

Por ejemplo, para una matriz (array bidimensional) de 3x4, podemos declarar:

```
int [,] v;
v = new int[3,4];
```

- En la declaración, si en vez de `[,]` escribimos `[,,]` declararíamos un array de 3 dimensiones, i.e., podemos declarar arrays de tantas dimensiones utilizando comas (dimensión = comas +1)
- La inicialización debe ser coherente con el número de dimensiones y proporcionar el tamaño en cada dimensión.

Se puede pensar en un array bidimensional como una matriz estándar, donde el primer índice referencia la fila y el segundo, la columna. Por ejemplo, para el vector `v` anterior (de dos dimensiones, con tamaños 3 y 4):

	0	1	2	3
0	_	_	_	_
1	_	_	_	_
2	_	_	_	_

(En realidad "fila" y "columna" es nuestra forma de asimilar las dimensiones; C# internamente no hace esta distinción. Trabaja con "primer índice", "segundo índice", etc).

- Igual que antes, la componente `v[i,j]` se comporta a todos los efectos como una variable de tipo `int`).

Arrays bidimensionales: inicialización

Inicializar un vector `v[3,4]` con los números 1 a 12:

```
int cont = 1;
for (int i = 0; i < 3; i++)
  for (int j = 0; j < 4; j++) {
    v [i, j] = cont;
    cont++;
  }
```

Implementar un método para escribirlo en pantalla:

```
static void escribeMat(int [,] v, int fils , int cols){
  for (int i = 0; i < fils ; i++) {
    for (int j = 0; j < cols; j++) {
      Console.Write ("{0,3}", v [i, j ]);
    }
    Console.WriteLine ();
  }
}
```

```
  1  2  3  4
  5  6  7  8
  9 10 11 12
```


Se podría hacer sin pasar las dimensiones del vector:

- `v.GetUpperBound(i)` $\rightsquigarrow$ índice de la última componente en la dimensión i.

Haríamos:

```
static void escribeMat2(int [,] v){
  for (int i = 0; i < v.GetUpperBound(0)+1; i++) {
    for (int j = 0; j < v.GetUpperBound(1)+1; j++) {
      Console.Write ("{0,3}", v [i, j ]);
    }
    Console.WriteLine ();
  }
}
```

Incluso se puede hacer sin pasar el número de dimensiones, pasando el vector como tipo `Array` (predefinido en el sistema), y obtenerlas con `v.Rank` (en el ejemplo anterior `v.Rank` vale 2).

Suma de dos matrices del mismo tamaño (asumimos `m1`, `m2` y `m3` declaradas de dimesión 2 y del mismo tamaño):

```
static void sumaMat(int [,] m1, int [,] m2, int [,] m3){

  int fils = m1.GetUpperBound(0) + 1,
      cols = m1.GetUpperBound(1) + 1;

  for (int i=0; i<fils; i++)
    for (int j=0; j<cols; j++)
      m3[i,j] = m1[i,j] + m2[i,j];
}
```


Buscar la posición del primer elemento negativo de una matriz:

```
static void buscaNeg(int [,] v, out int x, out int y){
  int i, j;
  int fils = v.GetUpperBound (0) + 1,
      cols = v.GetUpperBound (1) + 1;

  bool cont = true;
  // recorrido por filas
  i = 0;
  while (i<fils && cont) {
    // recorrido por columnas. Para cada fila columna a 0
    j = 0;
    while (j<cols && cont) {
      if (v[i,j]<0) cont = false;
      else j++;
    }
    if (cont) i++;
  }

  // negativo encontrado, devolvemos coordenadas
  if (!cont) {
    x = i; y = j;
    // si no, devolvemos coordenadas negativas (no válidas)
  } else x = y = -1;
}
```

Distancias kilométricas: arrays escalonados

Queremos almacenar la distancia entre las principales ciudades:

	Distancias kilométricas									
	Albacete									
Alicante	171	Alicante								
Almería	369	294	Almería							
Avila	366	537	663	Avila						
Badajoz	525	696	604	318	Badajoz					
Barcelona	540	515	809	717	1022	Barcelona				
Bilbao	646	817	958	401	694	620	Bilbao			
Burgos	488	659	800	243	536	583	158	Burgos		
Cáceres	504	675	651	229	89	918	605	447	Cáceres	
Cádiz	617	688	484	618	342	1284	1058	900	369	Cádiz
Castellón	256	231	525	532	805	284	607	524	701	873	Castellón

No necesitamos una matriz cuadrada: la distancia de Ávila a Bilbao es la misma que la de Bilbao a Ávila $\rightsquigarrow$ matriz triangular:

```
string[] ciudades = {"Albacete", "Alicante", "Almería", "Ávila", "Badajoz", "Barcelona", ...};

int [][] distancias = new int[46][]; // 46 ciudades

// No guardamos distancia de Albacete a Albacete -> empezamos en 1
for (int i=1; i<46; i++) distancias [i] = new int[i];

distancias [1] [0] = 171; // Alicante-Albacete
distancias [2] [0] = 369; // Almería-Albacete
distancias [2] [1] = 294; // Almería-Alicante
...
```


Tres nuevas construcciones de C#

Tres nuevas construcciones de C#

Hay tres construcciones de uso limitado, pero que conviene conocer:

- Condicional múltiple `switch`
- Bucles `do-while`
- Bucles `foreach-in`

Selección múltiple: construcción `switch`

Problema: dado un entero que representa el mes del año, escribir el mes en letras.

Y distinguiríamos casos con *if*, *else-if*:

```
if (mes == 1) Console.WriteLine ("Enero");
else if (mes == 2) Console.WriteLine ("Febrero");
...
else if (mes ==12) Console.WriteLine ("Diciembre");
```

Con la construcción `switch` podemos escribir:

```
switch (mes) {
  case 1:
    Console.WriteLine ("Enero");
    break;
  case 2:
    Console.WriteLine ("Febrero");
    break;
  ...
  case 12:
    Console.WriteLine ("Diciembre");
    break;
}
```

Cómo funciona?

- toma el valor de `mes`
- busca el caso que corresponda entre los `case`
- ejecuta el código de ese case
- y la instrucción `break` finaliza la búsqueda de casos

Construcción `switch`: caso por defecto

Entre los casos, podemos también incluir al final el caso `default`.
Si en el ejemplo anterior no controlamos que el mes este en el rango `[1-12]` tendría sentido lo siguiente:

```
switch (mes) {
  case 1:
    Console.WriteLine ("Enero");
    break;
  ...
  case 12:
    Console.WriteLine ("Diciembre");
    break;
  default:
    Console.WriteLine ("Mes no válido");
    break;
}
```

Lo razonable es poner al caso por defecto como último caso por legibilidad (pero C# no obliga a ello).

Construcción `switch`: tipo de la expresión

La *expresión* sobre la que se hace la selección puede ser de tipo entero, `char`, `string` o incluso `bool`... no puede ser real (en coma flotante).

Conversión inversa de mes en letra a número:

```
string mesLetra = ...;
switch (mesLetra) {
  case "Enero":
    Console.WriteLine (1);
    break;
  case "Febrero":
    Console.WriteLine (2);
    break;
  ...
  case "Diciembre":
    Console.WriteLine (12);
    break;
}
```

Como es natural, el tipo de la expresión del `switch` debe coincidir con el tipo de los `case`'s

Construcción `switch`: agrupando casos

Cuando en distintos casos se ejecuta el mismo código es útil poder agrupar esos casos. Por ejemplo: escribir el número de días de un mes dado (dado el año)

```
int anio = ..., mes = ...;
switch (mes) {
    case 2: // en febrero distinguimos bisiestos
        if (bisiesto(anio)) // suponemos implementado el método bisisieto
          Console.WriteLine (29);
        else
          Console.WriteLine (28);
        break;
    case 4: // agrupamos abril, junio, septiembre y noviembre
    case 6:
    case 9:
    case 11: // todos ellos con 30 días
        Console.WriteLine (30);
        break;
    default: // el resto 31
        Console.WriteLine (31);
        break;
}
```

Bucle *do-while*

Hemos visto dos construcciones para bucles

- `while («cond») ...`
- `for («inic» ; «cond» ; «incr») ...`

La construcción `while` sería suficiente: podríamos hacer todos los bucles con `while` ... pero la construcción `for` aporta claridad y expresividad al lenguaje.

Por la misma razón, hay otra construcción más para bucles:

```
do {
  << código >>
} while (cond);
```

Significado: ejecutar «código» mientras cond sea true

Uso: cuando el cuerpo del bucle se repita al menos una vez.
Por ejemplo, en la petición de datos de teclado:

```
do {
  Console.Write ("Altura del triángulo: ");
  alturaT = double.Parse(Console.ReadLine());
} while (alturaT < 0 || alturaT > 10000);
```

Mejor o peor que estas versiones?

```
// leemos fuera del bucle y luego otra vez dentro
Console.Write ("Altura del triángulo: ");
alturaT = double.Parse(Console.ReadLine());

while (alturaT < 0 || alturaT > 10000) {
  Console.Write ("Altura del triángulo: ");
  alturaT = double.Parse(Console.ReadLine());
}
```

```
alturaT = 0; // para forzar entrada " artificial " en bucle
while (alturaT < 0 || alturaT > 10000) {
  Console.Write ("Altura del triángulo: ");
  alturaT = double.Parse(Console.ReadLine());
}
```

En este ejemplo, es más natural la construcción *do-while* . . . pero esta petición de datos es uno de los pocos casos de utilidad del *do-while*.

do-while es muchísimo menos frecuentes *while*.

Por qué es más útil y frecuente la construcción *while*? Cúantas veces se ejecuta el cuerpo de los bucles?

- En el *while* puede ejecutarse $n = 0, 1, \ldots$ veces
- En el *do-while* puede ejecutarse $n = 1, \ldots$ veces

$\rightsquigarrow$ es más general la construcción *while*.

Traducción de *do-while* a *while*

Es muy fácil expresar una construcción *do-while* como *while*:

```
do {
  <<código>>
} while (cond);
```

Se expresa como:

```
<<código>>
while (cond) {
  <<código>>
}
```

En el bucle pricipal de un videojuego, cuál es más natural?

Bucle `foreach-in`

Ahí aun otro tipo de bucle para hacer recorridos en arrays (en general en colecciones o estructuras *enumerables*).

El uso es bastante inmediato:

```
int[] fibarray = { 0, 1, 1, 2, 3, 5, 8, 13 };
foreach (int elem in fibarray)
{
  Console.Write(elem + "  ");
}
```

La sintaxis general es:

```
foreach (<<tipo>> variable in <<var_array>>)
{
  <<código>>
}
```

Sirve para recorrer un vector y ver su contenido sin modificarlo (no es tan potente como el `for` o el `while`).

Calcular la suma de los elementos de un vector bidimensional:

```
int[,] v = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

int sum = 0;
foreach (int elem in v) sum += elem;
```

En general, sirve para vectores de cualquier dimensión. Ventaja: no maneja índices explícitos de recorrido.

Archivos

Archivos y flujos (*streams*)

Hasta ahora nuestros programas pueden procesar datos de entrada y producir datos de salida en pantalla. . . pero no pueden almacenar datos de modo persistente (en almacenamiento secundario, disco duro) $\rightsquigarrow$ necesitamos escribir y leer archivos.

En C# (y otros lenguajes) se trabaja con archivos a través de flujos:

- son un enlace entre el programa y la *fuente* de datos.
- los datos *fluyen* a través de los flujos para lectura o escritura.
- establecen capa de abstracción entre el programa y el sistema operativo: C# convierte las instrucciones de acceso a archivos en otras de más bajo nivel para el SO $\rightsquigarrow$ simplifica y facilita mucho el trabajo con archivos porque uniformiza el tratamiento en distintos SO's.
- Ya hemos utilizado flujos: la clase `Console` está implementada mediante un flujo!

C# proporciona diversos tipos de *streams*. Nos interesan dos en particular: `StreamReader`, `StreamWriter`

Flujos de salida

```
using System;
using System.IO;
...
public static void Main (string[] args) {
  // declaración de flujo de salida
  StreamWriter salida;

  // creación y asociación a un archivo concreto
  salida = new StreamWriter ("prueba.txt");

  // escritura en el flujo (va al archivo), igual que en consola
  salida.WriteLine ("Archivo de prueba:");
  for (int i = 0; i < 5; i++)
    salida.Write(i + " ");

  // cierre, importantísimo!!
  salida.Close ();
```

En el archivo `.... /bin/Debug/prueba.txt` tendremos:

```
Archivo de prueba
0 1 2 3 4
```

En este caso se crea el archivo en el directorio donde corre el ejecutable, pero se puede escribir cualquier archivo, en cualquier directorio (al que nos deje acceso el SO), colocando la ruta (path) adecuado al crear el flujo con `new`).

Cierre de flujo (y archivo asociado)

El flujo se cierra con:

```
salida.Close ();
```

Este cierre es muy importante porque hace varias acciones cruciales:

- flush de datos: el flujo (variable lógica) vuelca en archivo (físico) todos los datos que queden pendientes (por motivos de eficiencia, la escritura en el archivo físico no es inmediata)
- se destruye el flujo (la variable lógica correspondiente) y se liberan recursos
- se cierra el archivo (físico) y ya puede ser accedido por cualquier otro programa

Si no se hace `Close` pueden surgir errores y problemas incómodos: pueden *desaparecer* datos que se pensaban salvados en archivo (en realidad nunca llegaron a escribirse), otros programas no podrán acceder al archivo escrito, ...

Flujos y espacios de nombres

Para que todo esto funcione hemos incluido la sentencia:

```
using System.IO;
```

C# es un lenguaje con

- un pequeño repertorio de instrucciones
- y una enorme biblioteca de recursos
  - organizados o catalogados en espacios de nombres

Un espacio de nombres es un espacio donde los nombres tienen sentido (están definidos). Al especificar un espacio de nombres en el programa indicamos dónde buscar esos recursos.

- Los espacios de nombres pueden anidarse: en un museo podemos tener un espacio dedicado a Roma y dentro de ese espacio un subespacio dedicado a vasijas romanas.
- $\rightsquigarrow$ dentro de `System` está el espacio `System.IO` que proporciona acceso a los flujos de entrada/salida.

Flujos y archivos de entrada

Se sigue la misma filosofía que con los de salida:

```
using System.IO;
 ...

// declaración de flujo de entrada
StreamReader entrada;

// creación de flujo y asociación a archivo
entrada = new StreamReader ("sal.txt");

// lectura de línea de texto
string s = entrada.ReadLine ();

// cierre de flujo
entrada.Close ();

// se puede ver en consola la línea leída
Console.WriteLine (s);
```

Flujo de entrada. Fin de archivo

En el programa anterior hemos leído una línea del archivo de entrada. . . y si queremos leer más líneas? Cómo detectamos cuándo acaba el archivo? ⇝ propiedad `EndOfStream`

Leer un archivo línea a línea y sacar su contenido en consola:

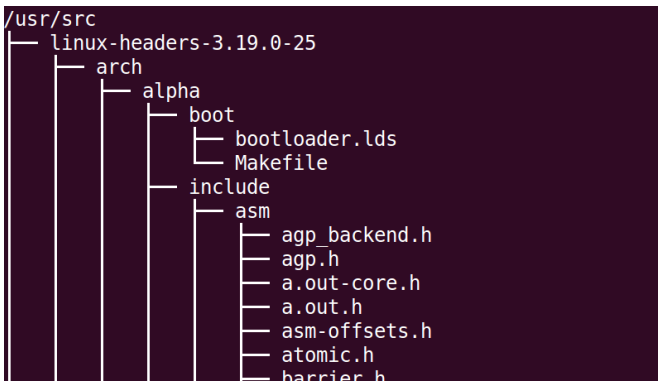
```
StreamReader entrada;

entrada = new StreamReader ("sal.txt");
while (!entrada.EndOfStream){
  string s = entrada.ReadLine ();
  Console.WriteLine (s);
}
entrada.Close ();
```

También se puede leer carácter a carácter en vez de línea a línea, haciendo en el bucle: `char c = (char) entrada.Read();`

Rutas de archivos (paths)

En cualquier sistema operativo (Linux, Windows, . . .) los archivos se organizan en una estructura de directorios (o carpetas) con forma de árbol del estilo:



La ruta o path para acceder a un archivo es una concatenación de los nombres de directorio que llevan hasta ese archivo, y finalmente el propio nombre del archivo. Por ejemplo, de acuerdo al árbol anterior son paths válidos (en Linux):

- `/usr/src/linux-headers-3.19.0-25/arch/alpha/boot/bootloader.lds`
- `/usr/src/linux-headers-3.19.0-25/arch/alpha/include/asm/a.out.h`

Hay *directorios especiales* como "." que se refiere al directorio actual, o ".." que se refiere al *padre* del directorio actual.
En Windows los paths son análogos, pero con "\" en vez de "/".
Por ejemplo, en Windows podríamos declarar:

```
string ruta = @"c:\datos\2016\enero\ventas.txt"
```

Nótese que especificamos *cadena literal* con "@" para que no interprete los símbolos "\" como secuencias de escape.

- Para mantener la compatibilidad entre distintos SO's, C# proporciona la clase `Path` con diferentes propiedades para cambiar los separadores de los paths.