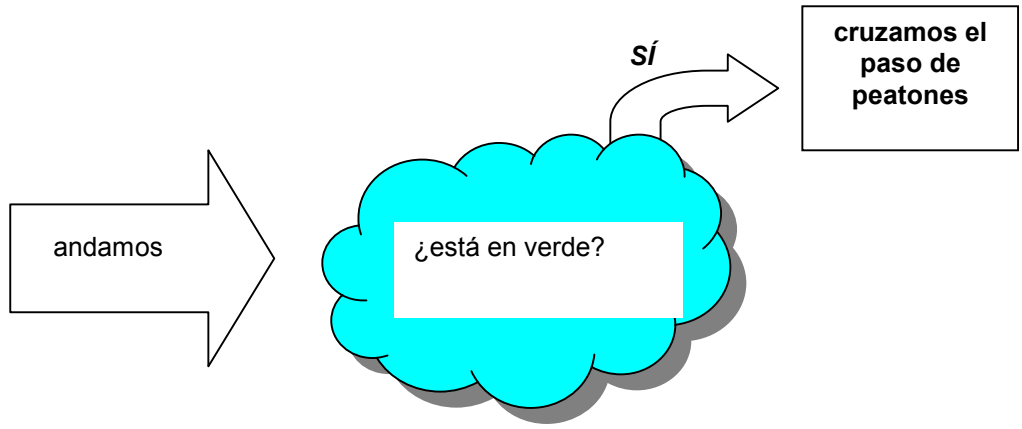
TEMA 3
ESTRUCTURAS CONDICIONALES

1 INTRODUCCIÓN

Hasta el momento hemos visto que un programa ejecuta acciones de forma seguida (secuencial), podemos mostrar valores por pantalla, pedir datos, realizar operaciones matemáticas sencillas…
Pero, como en la vida, no todo se reduce a un procedimiento fijo, siempre en el mismo orden, ni tan siquiera siempre las mismas acciones. Las prestaciones de las estructuras condicionales posibilitan construir programas más potentes, versátiles y dinámicos.
Ahora bien, una buena programación eficiente y comprensible implica un buen diseño del programa, por lo que conocer cómo *razonan* las expresiones resulta fundamental.

1.1 ¿Todo funciona de modo secuencial?

Imaginemos que estamos andando por una acera y nos acercamos a un paso de peatones con semáforo. ¿Qué hacemos? La respuesta es: preguntar. Podríamos preguntar si el color del semáforo es verde, en cuyo caso cruzaríamos.
Dicho de otra manera:



2 LAS CONDICIONES

A la hora de representar la situación anterior, debemos expresar la condición que queremos preguntar. Una posible manera de hacerlo es mediante pseudocódigo o pseudo-lenguaje natural:

SI *semáforo es de color verde*
ENTONCES *cruzamos el paso de peatones*

Así, podríamos pensar en una variable de tipo char llamada *color_semaforo* cuyos valores puedan ser 'r' para el color rojo y 'v' para el color verde.
Nos falta la herramienta para preguntar. Si de forma natural diríamos "**si** color_semaforo **es verde**", en lenguaje C esto se expresaría como:

```
if ( color_semaforo == 'v') {
      printf("ahora cruzamos el paso de peatones);
}
```

Veamos:

- ***if*** equivale a un "**si**" (conjunción condicional). Nótese que se escribe en minúsculas y no de otra forma (el lenguaje C es sensible a las minúsculas/mayúsculas, diferencia entre ambas, por lo que para el compilador 'C' y 'c' son *caracteres distintos*)
- **==** es el operador de igualdad, para comprobar si el primer operando es exactamente igual al segundo.
- En este caso comprobará si el valor de la variable *color_semaforo* es exactamente el carácter '*v*'.
- *la pregunta se escribe entre paréntesis*
- la instrucción o instrucciones se escriben entre llaves, independientemente del número de sentencias implicadas.

Ahora bien, la pregunta que nos surge es qué significa realmente la expresión color_semaforo == 'v', esto es, cómo se interpreta realmente.

Cuando se evalúa una expresión, se traduce por un valor en lógica binaria: verdadero o falso. En lenguaje C, realmente lo que sucede es que cualquier expresión, al ser evaluada, toma un valor. Si ese valor es 0, entonces se interpreta como *falso*; cualquier otro valor distinto de cero equivale a *verdadero*.
Por ejemplo, tenemos dos variables:

puntosEquipo1	puntosEquipo2
5	*10*

La expresión *puntosEquipo1 == 4* toma un valor falso, o lo que es lo mismo, un valor numérico equivalente a 0. Si realizamos la siguiente pregunta:

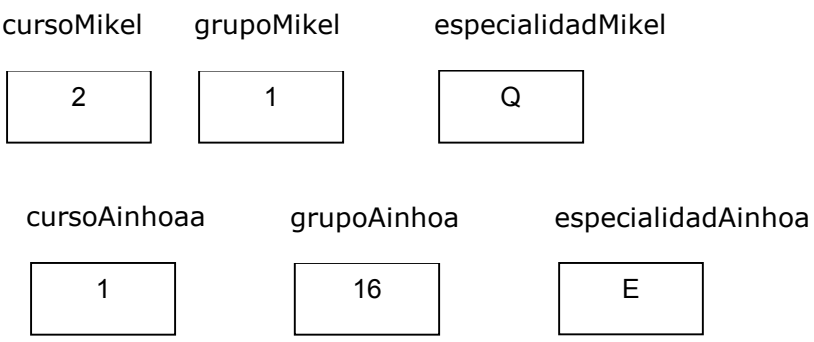
```
if (puntosEquipo1 == 4) {
      printf("El primer equipo no tiene 10 puntos.");
}
```

La respuesta es *falso*, por lo que no se tendrá que imprimir ningún texto por pantalla. Coloquialmete se suele decir que "*no entra por la rama del if"*, o simplemente, que "*no entra en el if"*.

Si lo que se pretende es comprobar que su valor es distinto a otro entonces el operador a utilizar es *!=* (exclamación de fin más signo igual).

```
if (puntosEquipo1 !=  10) {
      printf("El primer equipo no tiene 10 puntos.");
}
```

Ejercicios:
Tenemos almacenada la información de todos los alumnos y alumnas de la EUITI.



1.- Expresa, mediante instrucciones *if* las siguientes comprobaciones:

a.- Que Mikel no está en tercer curso
b.- Que Ainhoa es de la sección electrónica.
c.- Que Ainhoa y Mikel son de la misma especialidad
d.- Que Mikel es de la especialidad de Mecánica ('M')
e.- Que Mikel y Ainhoa están en distinto curso.

2.- Indica la respuesta correcta en las condiciones siguientes

a.- if (cursoAinhoa != cursoMikel)
b.- if (grupoMikel == 2)
c.- if (cursoAinhoa != grupoMikel)
d.- if (especialidadAinhoa != 'M')

También podemos realizar preguntas que impliquen a varios operandos. Así, podemos saber cuál de los dos equipos tiene más puntuación:

```
if (puntosEquipo1 > puntosEquipo2) {
        printf("El primer equipo tiene más puntos que el segundo");
}
```

Existen otros operadores para evaluar numéricamente dos valores:

Símbolo	Descripción	Ejemplo
>	Mayor que	*puntosEquipo1 > 3* (comprueba si el primer equipo tiene más de 3 puntos)
>=	Mayor o igual	*puntosEquipo1 >= 3* (comprueba si el primer equipo tiene 3 ó más puntos)
<=	Menor o igual	*puntosEquipo1 <= 3* (comprueba si el primer equipo tiene 3 puntos o menos)
<	Menor	*puntosEquipo1 < 3* (comprueba si el primer equipo tiene menos de 3 puntos)

Nota: A diferencia de los operadores matemáticos (≥,≤, ≠) los operadores en lenguaje C se escriben con dos símbolos (>=, <=, *!*=).

Además, mientras en el lenguaje matemático se permiten expresiones del tipo *1< x ≤ 20*, los operadores en C son binarios, esto es, únicamente permiten dos operandos. La expresión anterior debería construirse, entonces, en base a dos expresiones binarias más sencillas:
Por una parte, *1<x*, y por la otra *x ≤ 20*, estableciéndose además la condición de que ambas condiciones se cumplan necesariamente.

2.1 Cuestión de lógica…

Como se ha expuesto anteriormente, la lógica binaria consiste en la formulación de expresiones que al ser evaluadas deriven en un resultado interpretable en términos de verdadero o falso. Por otro lado, las expresiones lógicas pueden ser simples o compuestas, es decir, que impliquen un solo término o varios.
Por ejemplo, si decimos "el fin de semana iré a Paris si hace sol y si tengo dinero para pagar el viaje", entonces para poder ir a París tendrán que darse como buenas las dos condiciones a la vez. Es decir, se tendrá que cumplir que:

Hace_sol es verdadero **Y** dinero_disponible mayor que precio_billete

de otra manera

tiempo_soleado == 'S' **AND** dinero_disponible > precio_billete

En lenguaje C se expresaría como

tiempo_soleado == 'S' **&&** dinero_disponible > precio_billete

Así, para cualquier expresión AND con 2 términos, el resultado es el siguiente

		Segundo término	
		Verdadero	Falso
Primer término	Verdadero	***verdad***	*falso*
	Falso	*falso*	*falso*

O lo que es lo mismo, para que una expresión AND (más estrictamente, *operador lógico AND*) sea cierta, verdadera deberán ser verdad sus dos términos.

Sin embargo, podemos tener otro tipo de relación entre términos dentro de una expresión lógica. Veamos:

Si decimos "el fin de semana iré a París si hace sol o si tengo dinero suficiente para pagar el viaje", querrá decir que podré ir a París si:

hace_sol es verdadero **O BIEN** dinero disponible es mayor que precio del billete

de otra forma
hace_sol == 'S' **OR** dinero_disponible > precio_billete

en C

(hace_sol == 'S') **||** (dinero_disponible > precio_billete)

2.2 El operador unario de negación

Aunque hemos visto dos operadores lógicos binarios (AND y OR), existe un operador especial unario, que es el de negación. De forma muy simple podemos decir que este operador establece el opuesto lógico de una expresión. Por ejemplo, si tenemos una expresión, *x>=8* (x es mayor o igual a 8), negar dicha expresión lo representamos como *!(x>=8)*. Esto quiere decir que x no es mayor o igual a 8 ó dicho de otra manera, x es menor que 8 (*x<8*).
Si al evaluar la expresión x>=8 nos diera verdadero, entonces al evaluar la contraria, obtendríamos falso; y viceversa.
Resumiendo

expresión	!expresión
Verdadero	***Falso***
Falso	***verdadero***

Combinación y transformación de expresiones con operadores

```
El Athletic desciende a segunda división si
!( athletic_gana && real_pierde) ➔ (!athletic_gana) || (!real_pierde)

(verdadero si A o B son falsos)

El Athetic gana la Copa del Rey si
!(madrid_gana || barcelona_gana) ➔ (!madrid_gana) && (!barcelona_gana)

(verdadero si A y B son falsos)

El Athletic saca algún punto si
!( puntos_athletic == 0) ➔ (puntos_athletic != 0)

La Real queda por delante del Athletic si
!( puntos_athletic <=puntos_real) ➔ (puntos_athletic >= puntos_real)
```

Práctica

A partir de la siguiente situación de puntos, y con las variables athletic_pierde, real_gana, madrid_pierde, barcelona_pierde expresa las condiciones para que:

a) El Athetic gane la Liga
b) El Athetic no quede primero

Athletic	42 puntos
Barcelona	44 puntos
Real	41 puntos
Madrid	40 puntos

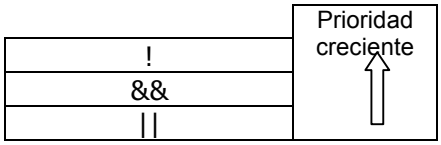
3 EVALUACIÓN DE LAS EXPRESIONES COMPLEJAS

Las expresiones complejas describen situaciones que se construyen a partir de condiciones más sencillas. Por lo tanto, podemos decir que la evaluación de las expresiones complejas se pueden componer/descomponer en base a la evaluación de expresiones simples.

La forma más sencilla de escribir e interpretar expresiones compleja es utilizando paréntesis que eliminen posibles ambigüedades. Pero esto no es necesario, por lo que el lenguaje C tiene su propia manera –por defecto- de interpretar las expresiones.

Si tenemos una condición compleja y no utilizamos paréntesis tendremos que tener en cuenta dos circunstancias:

1. Las expresiones se evalúan de izquierda a derecha si no existe diferencia de prioridad entre ellas
2. Si existe prioridad entre ellas, entonces es necesario considerar el orden de prioridades.



Las expresiones

(45 > 10) || (num > 100 + 5) || (12 = = -63)

y

((45 > 10) || (num > 100 +5)) || (12 = = -63)

son equivalentes.

Los paréntesis no son necesarios si se utiliza en la condición compleja el mismo operador
Si se utilizasen diferentes operadores tendremos que tener cuidado con las prioridades y seguramente tendremos que utilizar paréntesis**.**

Ejercicios:
3- Representar las expresiones siguientes
ej:
4.- Simplificar en expresiones simples las siguientes expresiones complejas.
ej:

4 ELSE (O MEJOR DICHO: *ELSE*)

En ocasiones necesitamos, no sólo realizar acciones determinadas en el caso de que se cumpla la condición a validar, sino justamente en el caso contrario.

```
if (esta_lloviendo == 1) {
      printf("Llevamos paraguas");
} else {
      printf ("Llevamos sombrilla");
}
```

5 LAS INSTRUCCIONES CONDICIONADAS Y "LO QUE QUEDA FUERA DE LAS LLAVES"

Al construir una instrucción condicional estamos representando qué acciones se llevan a cabo cuando la condición se cumple (en cuyo caso entrará en el if) y cuando no se cumple (se ejecutará el else). En ambos casos, las acciones a ejecutar son las delimitadas entre llaves o, dicho de otra manera, no se ejecutarán las sentencias que no estén entre llaves. Veamos un ejemplo:

```
1     if (calificacionFI >=9 ) {
2           printf ("Has sacado sobresaliente");
3     } else {
4           printf ("No has sacado sobresaliente");
5     }
6     calificacionFI = 7;
```

Al final de todas las instrucciones tendremos que: aparece en pantalla el texto segundo, y además el valor de la variable calificacionFI es 7 independientemente de su valor inicial.

Veamos este otro ejemplo

```
1     if (calificacionFI >=9 ) {
2           printf ("Has sacado sobresaliente");
3           calificacionFI = 5;
4     } else {
5           printf ("No has sacado sobresaliente");
6     }
7     printf("ta luego!!");
```

Al final de todas las instrucciones tendremos dos posibilidades: si la calificación inicial es nueve o más, aparecerá en pantalla el texto segundo, y además el valor de la variable calificacionFI es 5. Si no se da la condición, sacará el mensaje "No has sacado sobresaliente" y calificacionFI tendrá el valor inicial. Lo que sí se escribirá en todos los casos es el texto "ta luego!!".

6 SECUENCIA Y ANIDAMIENTO DE CONDICIONES

Las instrucciones condicionales pueden agruparse de varias maneras: en secuencia o bien de forma anidadas. Se escriben de forma seguida cuando tras evaluar una condición, e independientemente de los resultados de esa evaluación, sea necesario realizar otra evaluación.
Por ejemplo:

```
if ( alturaArmario >= 100) {
    alto = 'S';
} else {
    alto = 'N';
}

if ( anchuraArmario >= 200) {
    ancho = 'S';
} else {
    ancho = 'N';
}
```

Al final de todo tenemos dos variables (alto y ancho) que son modificadas una independientemente de la otra.

Sin embargo, en el caso de las instrucciones

```
if ( alturaArmario >= 100) {
    alto = 'S';
    if (anchuraArmario > 1000) {
          altoYsuperAncho ='S';
    } else {
    altoYsuperAncho = '1';
} else {
    alto ='N';
    altoYsuperAncho = 'N';
}
```

tenemos que la variable altoYsuperAncho se modifica de dos maneras distintas, en función de si el armario es alto o no (primer *if*) y de la anchura del armario.
Los if se pueden anidar, tanto en la rama verdadera (cuando se cumple) como en la falsa (cuando no se cumple).

En lenguaje C existe un número máximo de 256 anidamientos permitidos, nivel de complejidad al que no llegaremos en esta asignatura, principalmente porque en ese caso el diseño del programa no sería ni sencillo ni probablemente eficiente.

Por otra parte, dado que un conjunto de condiciones puede estar anidado varias veces, nos debe quedar claro qué es lo que estamos escribiendo, por lo que para evitar indeseadas ambigüedades, se establece que un else queda ligado automáticamente a su if anterior más cercano.
Ahora bien, una forma sencilla de escribir código de manera que nos evitemos dificultades tanto durante la programación como en la depuración de los programas es utilizando una guía de estilo para la programación. Las guías de estilo varían mucho de unos programadores a otros (cada una de ellas con sus ventajas e inconvenientes), pero en cualquier caso evitan errores indeseados y facilitan la comprensión de los programas.
Por ejemplo, aunque en lenguaje C se permite escribir todas las instrucciones seguidas (incluso en una sola línea), lo cierto es que separarlo en diferentes líneas facilita su lectura. El uso de sangrías, espacios, paréntesis ayuda también en este mismo sentido.

Para las mismas instrucciones, estilos diferentes:

```
if (color ='v') {
      printf("El color es verde");
} else {
      printf ("No es de color verde");
}
```

Menor número de líneas. El control del cierre de las llaves se hace visualmente bajo la vertical del if

```
if (color ='v')
{
      printf("El color es verde");
}
else
{
      printf ("No es de color verde");
}
```

Más líneas. En la vertical de la llave "{" deberá aparecer "}"

```
if (color ='v')
      {
      printf("El color es verde");
      }
else
      {
      printf ("No es de color verde");
      }
```

Más líneas. La sangría de llaves y de las sentencias de if y else permite distinguir los bloques if y else.

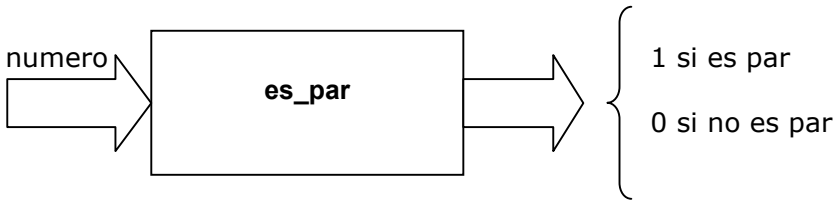
Ejercicios:
Preparando material para camping
Expresa mediante mensajes qué material llevaremos para el próximo fin de semana, en función de que haga sol, llueva, haga viento, etc.
Si llueve, llevamos paraguas. SI además hace frío, cogeremos abrigo. Pero si no llueve, llevaremos pantalón. Si además de no llover hace sol, llevamos sombrilla, crema protectora. Si hace sol y viento cogeremos una gorra.

7 ¿SOLO PODEMOS EVALUAR CONDICIONES NUMÉRICAS O ALFABÉTICAS?

Evidentemente, la respuesta es no. De ser así, todos los programas –del tipo que fueran- tendrían que reducirse a expresiones matemáticas, lo cual no siempre es posible. Dicho de otra manera, podemos realizar preguntas más allá de los operadores. ¿Cómo?
Supongamos que tenemos un *mecanismo* o especie de cajita negra que, diciéndole qué queremos, él nos devuelve un resultado como respuesta a nuestra pregunta. Lo que podremos hacer en ese caso es evaluar numéricamente cuál es el resultado que nos da.

Por ejemplo, tenemos la condición

```
if ( es_par(numero) == 1) {
      printf ("El número es par");
}
```



Pues bien, estos mecanismos llamados funciones (porque cumplen una función, en este caso decirnos si es par o no) también pueden ser evaluados o preguntados. Aunque las funciones se estudian en capítulos posteriores, es importante conocer que al utilizar una función podemos conocer qué resultado nos devuelve dicha función. Pero no solo eso, sino que a partir de dicho resultado tenemos la posibilidad de preguntar sobre dicho resultado.

8 ANEXO A – CUADRO COMPLETO DE PRIORIDADES Y ASOCIATIVIDADES

☐Precedencia	Operador										Asociatividad ☐
1	**( )**	**[ ]**	**++**	**--**							☐
2	**!**	**+**	**-**	**++**	**--**	**&**					☐
4	*****	**/**	**%**	Operadores aritméticos: multiplicación, división y módulo							☐
7	**<**	**<=**	**>**	**>=**	Operadores relacionales, mayor-que, menor-que, etc.						☐
8	**==**	**!=**	Operadores relacionales igual y desigual								☐
12	**&&**	Operador Y lógico									☐
13	**||**	Operador O lógico									☐
15	**=**	***=**	**/=**	**%=**	**+=**	**-=**	**&=**	**^=**	**|=**	**<<=**	**>>=** ☐

Nótese que el indicador de precedencia no sigue un orden secuencial. Esto se debe a la existencia de otros operadores (por ejemplo para operaciones con números binarios) que no aparecen en la tabla, pero se deja constancia de su prioridad.