

CURSO DE C

Características

- **Programación imperativa estructurada:** una **secuencia claramente definida de instrucciones para un ordenador.**
- Relativamente de “**bajo nivel**” (dentro de los lenguajes de alto nivel, es de los de menos nivel de abstracción).
- **Lenguaje simple, la funcionalidad está en las bibliotecas.**
- Básicamente **maneja números, caracteres y direcciones de memoria.**
- **No tiene tipado fuerte.**

Hola mundo: un vistazo rápido

```
#include <stdlib.h>
#include <stdio.h>

/* Comentario */

int
main(int argc, char *argv[])
{
    printf("hola mundo");
    exit(EXIT_SUCCESS);
}

// comentario en una línea
/* comentario en
más de una línea*/
```

La línea: int main ... es la cabecera y lo que hay entre llaves ({}) es el cuerpo.

Antes de nada: cómo compilar

La compilación se compone de tres fases. El programa gcc se encarga de las tres, en realidad es un front-end que invoca a otros programas:

- **Preprocesado:** incluye ficheros de cabecera (**#includes**: pega el fichero que describe el path), quita comentarios, etc.
El preprocesado es un programa que se llama cpp, no lo llamaremos directamente y antes de compilar se va a ejecutar.
Ejemplo de dist.h (declaración de la cabecera de la función y sirve para que cuando compile, no de un error de tipos).

¿Cómo descubre cuál es el tipo de la función dist?

Porque en realidad **el compilador no recibe el fichero main.c sino que recibe el fichero main.c preprocesado.**

El fichero main que ve ha pasado por el preprocesador (**cpp main.c**)

- **Compilación** (compiling): **genera el código objeto** a partir del código fuente, pasando por código ensamblador (aunque no nos demos cuenta).
- **Enlazado** (linking): **a partir de uno o varios ficheros objeto y bibliotecas, genera un único binario ejecutable.**

Siempre seguir este proceso cuando se quiere compilar un programa:

- Preprocesado y compilación:

gcc -c -Wall -Wshadow -Wvla -g hello.c

OJO: ¡los warnings hay que tratarlos como errores!

- Enlazado:

gcc -o hello hello.o

Para **conseguir ayuda** hay que usar **man** y **apropos** (en el libro pdf hay más información al respecto)

Hola mundo: disección

```
#include <stdlib.h>          /* Instrucciones para el preprocesador */  
#include <stdio.h>  
  
/* Comentario */  
  
int  
main(int argc, char *argv[]) /* Definición de función.  
                           Punto de entrada.*/  
{  
    printf("hola mundo");    /* Inicio de bloque */  
    exit(EXIT_SUCCESS);     /* Sentencia, llamada a función */  
    /* Sentencia, llamada a función */  
}  
                           /* Fin de bloque */
```

- **Expresión:** trozo de código que evalúa un código en tiempo de ejecución.
- En c es **importante el tamaño del registro de activación**, por eso hay que tenerlo en cuenta cuando se crea un programa. (trozo que queda en la pila).
- **main()** es la función por la que se comienza a ejecutar el programa, es lo que se denomina "punto de entrada". Recibe dos parámetros, retorna un valor, que es el status del programa (más adelante veremos las funciones). El **retorno de main no lo vamos a manejar**.
- **Llamaremos a exit a mano:** EXIT_SUCCESS (ha acabado bien) o EXIT_FAILURE (lo contrario)
- Es importante **declarar las variables al principio de una función** y así poder ver de una pasada el tamaño de registro de activación.
- Los **números con coma flotante** hay que usarlos con **mesura** (si lo necesitamos lo dirá el profe)
- Los **#include** tienen que **seguir un orden**, especificado en la página de manual correspondiente.
- Los **comentarios no pueden estar anidados**.

- Todas las **sentencias acaban** con un “;”. Ej: printf(“hola mundo”);
- Un bloque o sentencia compuesta es un grupo de sentencias que se trata sintácticamente como una única sentencia. Los **bloques** se **determinan mediante llaves {}**. Las sentencias de una función se engloban en un bloque.

Tipos de datos fundamentales

Dejamos fuera los tipos reales, nos quedamos con los enteros:

- **char**: carácter con signo (1 byte: -128 a 127), p.e. 'a' , c = 12 (literal de tipo char).
¿Qué significa que son enteros de 8 bits/1 byte?
Si meto un valor más grande se va a desbordar.

Ejemplos:

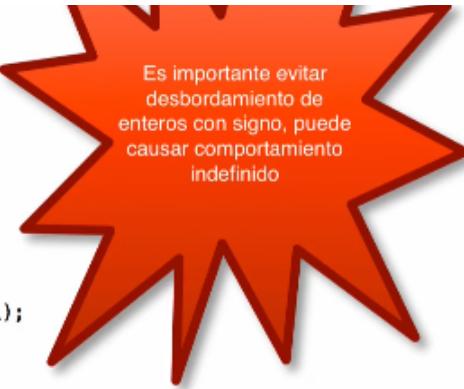
```
$ cc -o main main.c
$ ./main
valor de i:-1
$ cat main.c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    char i;

    i = 255;

    printf("valor de i:%d\n", i);
    exit(0);
}

$
```



Las variables en C **guardan su valor en complemento a 2** y por lo tanto i es igual a -1

Cuando llegan a 127, vuelven a empezar a -128. Por lo tanto, si el valor de i es 128, el valor de la salida será -128.

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    unsigned char i;

    i = 255;

    printf("valor de i:%d\n", i);
    exit(0);
}

~
```

```
$ cc -o main main.c
$ ./main
valor de i:255
$
```

Si usamos el valor 256, da un warning y se produce un desbordamiento y el valor de i es 0.

Unsigned (es un cualificador) significa sin signo y ese entero no se va a comportar como un complemento a 2 sino sin signo; con lo cual **va a ir de 0 a 255** y después se desbordaría. Se le puede añadir a char, entero y demás.

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    unsigned char i;

    i = 255;

    printf("valor de i:%d\n", i);
    exit(0);
}
```

Imprime 255

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    unsigned char i;

    i = -1;

    printf("valor de i:%d\n", i);
    exit(0);
}
```

Imprime 255 porque va a guardar un entero sin signo, interpreta el -1 con su valor en complemento a 2 como un entero sin signo y lo utiliza.

```
$ vi main.c
$ cc -c -Wshadow -Wall main.c
$ cc -o main main.o
$ ./main
i > j: 255 > -1
$ cat main.c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    unsigned char i;
    signed char j;

    i = -1;
    j = -1;
    if(i > j){
        printf("i > j: %d > %d \n", i, j);
    }
    exit(0);
}
```

Cuando se comparan, se comparan sus valores reales.

Aplicaciones de los enteros con signo y sin signo: **comparaciones y meter valores más grandes con menos bits.** **HAY QUE TENER MUCHO CUIDADO.**

Las diferentes formas de escribir enteros en c:

Decimal	10	Decimal	10
Hexadecimal	0x10	Decimal	16
Octal	010	Decimal	8

- **int**: entero con signo (4 bytes), p.e. 77 -11

```
int main(int argc, char *argv[])
{
    char i;

    i = 'a';

    printf("valor de i:%d\n", i);
    printf("valor de i:%c\n", i);
    exit(0);
}

$ cc -c -Wshadow -Wall main.c
$ cc -o main main.o
$ ./main
valor de i:97
valor de i:a
```

El tipo de datos char no es especial, en realidad es un entero de 8 bits.

Modificamos haciendo algo FEO:

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int|i;
    i = 'a';

    printf("valor de i:%d\n", i);
    printf("valor de i:%c\n", i);
    exit(0);
}

valor de i:97
valor de i:a
```

Obtenemos mismo output

C no define tamaños concretos para los tipos de datos solo define un tamaño mínimo. No hay garantías sobre de qué tamaño va a ser cada entero, sólo su mínimo. Si queremos saber el tamaño de datos en bytes de cualquier tipo usamos el built-in: **sizeof**

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    i = 'a';

    printf("valor de i:%lu\n", sizeof i);
    printf("valor de i:%lu\n", sizeof(char));
    exit(0);
}

$ ./main
valor de i:4
valor de i:1
$
```

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    i = 10;
    printf("El valor de i:%d\n", i);
    exit(0);
}
```

Imprime 10

Las operaciones con enteros se obtendrá valores enteros

```
$ ./main
El valor de i:10 j:3 i/j 3
$ vi main.c
e
```

- **unsigned char, uchar**: carácter sin signo (1 byte), usado para operar sobre bits como xor.
- **unsigned int, uint**: entero sin signo (4 bytes), p.e. 77
- **long**: entero largo, p.e. 431414341L
Es un cualificador que se le puede poner a un entero para que sea más ancho
- **long long**: entero más largo, p.e. 432423432423LL

Y otro:

- **void**: vacío (ya veremos para qué sirve).

Tamaño

- **C tiene tipado débil y no se queja si intentas asignar una variable a otra de distinto tamaño.**
- Si se asigna a una de menor tamaño, se truncá. Ejemplo: c = 358, es mayor que un byte y se truncará y no será el valor de 358.
- **Si se desborda una variable, nadie te lo va a decir.**
- **Puedes asignar una con signo a una sin signo (y viceversa).**
- **Que el tipo tenga signo solo se tiene en cuenta en las comparaciones**

- Las dimensiones de una variable no está definido, si queremos saber su **tamaño** es usando **sizeof**
- 035 (es octal) pero en C NO TIENE SENTIDO)

El uso de void es polisémica.

Una función que no devuelve nada es un procedimiento

Un puntero que apunta a void, apunta a cualquier cosa .

Un **identificador** es una **palabra que se inventa el programador para dar nombre a algo y tienen un ámbito** (espacio donde se pueden usar): **global** (desde donde lo inicias hasta el final del fichero) o **local** (desde las llaves más cercanas hasta la que los cierran)

Declaración e inicialización de variables

```
#include <stdlib.h>
#include <stdio.h>

int x = 1;           /* variables globales */
int k;

int
main(int argc, char *argv[])
{
    int i, q=1, u=12;   /* variables locales */
    char c;
    char p = 'o';

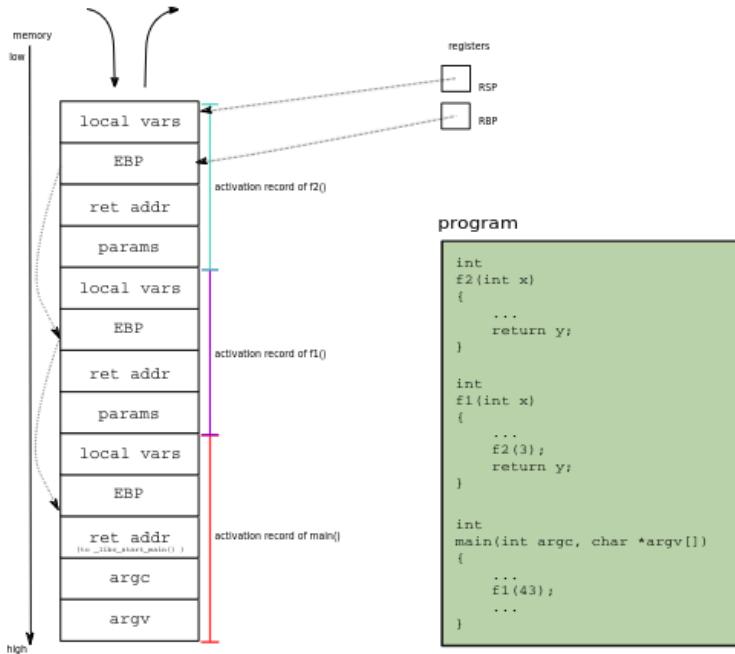
    c = 'z';
    i = 13;
    exit(EXIT_SUCCESS);
}
```

Una variable tiene un identificador y se trata de una abstracción que te da el lenguaje para llamar a un espacio de memoria.

Declaración de variables:

- Las **variables globales o externas** se declaran fuera de cualquier función. Se “ven” desde cualquier función del fichero, su ámbito es todo el fichero. Se localizan en el segmento de datos. **Si no se inicializan explícitamente, se inicializan a 0.**
- Las **variables locales o automáticas** se declaran dentro de una función y sólo se pueden “ver” dentro de su ámbito, que es el bloque en el que se han declarado y sus bloques anidados. Se localizan en la pila. Si no se inicializan explícitamente, tienen un valor indeterminado.

La pila



pila es memoria de la pila y tiene un puntero de pila que va apuntando ahí y va cambiando según se meten cosas

Declaración e inicialización

- Una variable declarada dentro de una función como static conserva el valor entre distintas invocaciones. Es así porque no se localizan en la pila, sino en el segmento de datos.
- Unas variables pueden ocultar a otras. Para enterarnos, usamos el modificador -Wshadow del compilador.
- Para inicializar, usamos valores constantes o literales:
 - Entero Decimal: 777
 - Entero Hexadecimal: 0x777
 - Entero Octal: 0777
 - Carácter: 'a', '\92'

Ejemplo:

int x = 1

No es una **asignación** (trozo de código que cuando se ejecuta, mete un valor en una variable) es una **inicialización**.

Una variable local no inicializada tiene basura. Cada vez que se llama a esa función tiene un nuevo registro de activación y si no lo haces, tendrá el valor que se quede ahí.

Declaración -> eje: int x; x sería de tipo int.

Importante recordar: **1º Inicialización y 2º asignación.**

Cuando retorne main, todos los registros de activación desaparecerán y nos quedaremos sin pila.

Las **variables globales** no se encuentran en la pila, sino en el **segmento de datos** y persisten a lo largo de toda la vida del programa.

El **shadowing** es cuando el **id local es igual al id global**. Si es un lenguaje con shadowing, generalmente se referirá al id local y sino al global; por eso, mejor no hacerlo y gracias a -Wshadow podemos recibir un warning para saber si hacemos shadowing.

Importante:

A c le da igual si es 0 o nulo
carácter nulo '\0' (caracter ASCII)

Existen concurs de ofuscación

Constantes

En c no existen constantes como tal sino una mezcla entre constantes y enumerados.

Declaración de **constantes enteras** mediante el uso de tipos enumerados con **enum**. Si no se da valor, se adjudican valores consecutivos desde el último definido:

Se pueden declarar local o global.

El primer valor es 0 ya que el módulo da vueltas con 0.

Ejemplo:

```
enum{  
    Lun,  
    Mar,  
    Mier,  
    Jue,  
    Vier,  
    Ndias,  
    SalarioBase = 2580,  
};
```

Hay que saber diferenciar entre :

- **constantes**: valor inmutable en tiempo de compilación. no necesariamente ocupan memoria.
- **variable inmutable**: variable que no puedes cambiar el valor (const) y ocupan memoria.

Ejemplos que **NO SON CONSTANTES**:

static const int VAR = 5;

- static: variable que va al segmento de datos; como las globales aunque solo se restringe su uso para un cierto espacio.

- const: va a un segmento de datos de solo lectura y no es constante porque tiene memoria reservada. Existe tanto en tiempo de compilación como de ejecución. Se puede obtener su dirección y hacer cosas con ellas que no se debería poder hacer comuna constante.

#define CONST 17

No es una constante, es una sustitución que va a hacer el preprocesador donde el compilador vea el nombre definido por define (que en realidad es un tipo de macro del preprocesador), lo va a sustituir por 17. Esto es antes del compilador y no sabe nada de esto y se producirá una sustitución textual.
Útil para strings y float.

Ejemplo de Sí son CONSTANTES:

Ponerle nombre a una enum no define un tipo de datos sino que define un alias de un entero. Lo podemos ver en este programa:



```
#include <stdlib.h>
#include <stdio.h>

enum Nx{
    A = 2,
    B,
};

int main(int argc, char *argv[])
{
    enum Nx a;

    a = 20;
    printf("%d %d\n", A, a);
    exit(0);
}
```

\$./prog
2 20

Ponerle nombre a los enum no tiene mucho sentido; en ocasiones puede ayudarnos para documentar parámetros de alguna función. RECOMENDACIÓN: NO USARLO

SÍ RECOMIENDA USAR ENUM PARA DEFINIR CONSTANTES.

Operadores aritméticos (pueden ser: prefijos, infijos y sufijos)

+	Suma. Operandos enteros o reales
-	Resta. Operandos enteros o reales
*	Multiplicación. Operandos enteros o reales
/	División. Operandos enteros o reales
%	Módulo. Operandos enteros

Existen operadores unarios, binarios,...

Módulo: devuelve el resto de la división entera (se usa para dar vueltas)

Ejemplo:

```
for(i = 0 i < 5 i++)
    i = (i+1) % 5
```

Output :

Nº de ejecución -> Valor obtenido

```
0 -> 0
1 -> 1
2 -> 2
3 -> 3
4 -> 4
5 -> 0
6 -> 1
7 -> 2
```

Los operadores aritméticos salvo el módulo valen para: int, char, long y long long.

Los **booleanos en c no existen**, cualquier entero puede funcionar como booleano.

Operadores lógicos

Las operaciones lógicas devuelven un entero

0 = FALSE

0 != TRUE

a && b	AND. 1 si "a" y "b" son distintos de 0
a b	OR. 0 si "a" y "b" son iguales a 0
!a	NOT. 1 si "a" es 0, 0 si es distinto de 0

```
$ cat main.c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

    i = 33;

    if(i > 8 && 0 && i < 35) {
        printf("esto es claramente falso\n");
    }

    if(i < 8 || i > 35) {
        printf("esto es claramente cierto\n");
    }
    exit(0);
}

$ ./main
esto es claramente cierto
$
```

El 0 es falso, por lo tanto, no entraría.

Operadores de relación

Devuelven un entero

Recuerda: 0 si es falso, cualquier otro valor si es verdadero:

a < b	"a" menor que "b"
a > b	"a" mayor que "b"
a <= b	"a" menor o igual que "b"
a >= b	"a" mayor o igual que "b"
a != b	"a" distinto que "b"
a == b	"a" igual que "b"

Ejemplo:

```
if (x + 3 == 0)
if (eslunes)
if (!eslunes)
```

Los **booleanos en c no existen**, cualquier entero puede funcionar como booleano.

```
$ cat main.c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    int j;

    i = 6;
    j = 5;
    printf("i > j, %d\n", i > j);
    exit(0);
}

$ ./main
i > j, 1
```

El resultado no tiene por qué ser 1, puede ser cualquier número distinto de 0.

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    int j;

    i = 1;
    j = 5;
    printf("i > j, %d\n", i > j);
    exit(0);
}
```

```

$ -----
| $ ./main
| i > j, 0
| if (int main (int argc, char *argv[])
| {
|     if(0)
|         printf("algo falso");
|     if(12234){
|         printf("algo verdadero");
|     }
|     exit(0);
| }

$ 

```

IMPORTANTE: Si es una sola línea, lo normal es que no se pongan las llaves.

```

$ ./mainif
algo verdadero$ 

```

If 0 no se imprime ya que 0 es falso, por eso imprime algo verdadero ya que 12234 es distinto de 0.

```

#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    i = 13;
    if(12 == i)
        printf("algo falso");
    if(12 != i)
        printf("algo verdadero");

    exit(0);
}

$ 

```

```

$ ./mainif
algo verdadero$ 

```

Operadores de asignación

Hacen 2 cosas a la vez: son expresiones y asignaciones

++	Incremento (pre o post)
--	Decremento (pre o post)
=	Asignación simple
*=	Multiplicación y asignación
/=	División y asignación
%=	Módulo y asignación
+=	Suma y asignación
-=	Resta y asignación

a = 0
d = a++ (post incremento) y es igual a : d = a , a = a + 1
a++ es a la vez un **operador sufijo + asignación**

```
$ cat main.c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;
    i = 3;
    i++;
    printf("valor de i:%d\n", i);
    printf("valor de i:%d\n", i++);
    printf("valor de i:%d\n", i);
    exit(0);
}

$ ./main
valor de i:4
valor de i:4
valor de i:5
$
```

Segundo ejemplo: el valor de 4 es porque printf imprime el valor actual y luego aumenta su valor al ser un post incremento.

Es como si se descompusiera en: printf("%d", i) y luego i = i +1

d = ++a (pre incremento) y es igual a: a = a + 1, d = a
En este caso printf imprimiría 5.

Ejemplo: b = (a + 3) asigna 3 a a y lo mete en b

if (a = 3) {} (bucle infinito)

$a^* = b \rightarrow a = a^*b$ (son útiles para no escribir paréntesis)

Operadores bit a bit y otros

Cada operador tiene significados distintos.

OJO: estas operaciones sobre tipos con signo pueden sorprendernos!

&	(unario) Dirección-de. Da la dirección de su operando
*	(unario) Indirección. Acceso a un valor, teniendo su dirección
~	(unario) Complemento
&	AND de bits
~	XOR de bits
	OR de bits
<<	Desplazamiento binario a la izquierda
>>	Desplazamiento binario a la derecha
?:	Operador ternario
sizeof	Operador de tamaño

*a (operador contenido)

a*b (multiplicador)

&a (dirección de a)

~a (complemento): negar los bits

a&b (and bit a bit) ¡cuidado con usarlo con signo. HAY QUE USARLO SIN SIGNO

!!Importante: Se caen por el lado que marca la flecha

>> : Se meten los bits por el lado **izquierdo**

<< : Se meten los bits por el lado **derecho**

Ejemplo:

```
uchar c, d;  
c = 2    00000010  
d = 3    00000011  
d = c&d 00000010
```

Desplazamientos binarios

Ejemplo:

c = 13 00001101

c = c >> 2 0000001101 (se eliminan estos 2 bits)

? : Los operadores ternarios NO USARLOS

mirar el código de **tests/bitbit.c**

Precedencia y asociatividad de operadores

Precedencia de operadores: el orden en la que se evalúa la expresión (hay que mirar la precedencia dependiendo del lenguaje que uses)

() [] -> .	izquierda a derecha
! ++ -- * & ~ sizeof (unarios)	derecha a izquierda
* / %	izquierda a derecha
+ -	izquierda a derecha
<< >>	izquierda a derecha
< <= > >=	izquierda a derecha
== !=	izquierda a derecha
& ^	izquierda a derecha
&&	izquierda a derecha
	izquierda a derecha
? :	derecha a izquierda
= += -= *= /= %= &= ...	derecha a izquierda
,	izquierda a derecha

No poner muchos paréntesis y tampoco no poner ninguno, mirar tabla

a[3]
*(a[3]) REPASARLO

Cuando hay varias multiplicaciones/divisiones seguidas es recomendable poner paréntesis.
Ejemplo:

a*b/c*d
(a*b)/c)*d

Definición y declaración de funciones

- Una función tiene que estar declarada antes de poder usarla en el código, pero puede estar definida después. Los tipos del prototipo y de la definición tienen que coincidir.
- Los argumentos siempre son por valor. Si queremos argumentos por referencia, tendremos que usar un puntero (lo veremos más adelante).
- Si una función no devuelve nada, es de tipo void.
- Si una función no tiene argumentos, entonces ponemos un argumento sin nombre de tipo void.

declarar variable: decir de qué tipo es la variable (int, char...local o global)

int (devuelve un int)
f(int a); (a partir de aquí se puede usar)

los include tiene declaraciones de tipos

En C el paso es siempre el **paso por valor**: la función recibe una copia del valor del argumento. Cualquier modificación realizada en el parámetro dentro de la función **no afecta al valor original del argumento**.

Paso por referencia: la función recibe una referencia o dirección de memoria del argumento. Cualquier modificación realizada en el parámetro **afecta directamente al valor original del argumento**.

```
int a  
a = 3  
  
f(&a)  
  
void  
f( int *b)  
  
*b = 8;  
  
printf(a) (el valor será de 8)
```

Librerías estándar

Las bibliotecas son colecciones de ficheros objeto pre-compilados que podemos usar. Las bibliotecas con funciones estándar:

<stdio.h>	Entrada y salida estándar: printf(), sprintf(), perror(), . . .
<stdlib.h>	Librería estándar de C: exit(), atoi(), getenv(), . . .
<string.h>	Operaciones con cadenas de caracteres: strlen(), strcat(), strcpy(), . . .
<unistd.h>	Llamadas al sistema de UNIX: fork(), read(), write(), close(), . . .
<fcntl.h>	Control de ficheros: open(), creat(), . . .

(Ahora sí) printf

```
int printf(const char * restrict format, ...);
```

- Tiene un número variable de parámetros.
 - El primer parámetro indica en una cadena de caracteres el formato de lo que se quiere imprimir por pantalla.
 - Cada % (es un **verbo**) en el formato se sustituirá con el parámetro que ocupa ese lugar después del formato.
 - Lo que viene después del % es la forma en la que se quiere imprimir el dato: %d es un entero, %c es un carácter, %x un entero hexadecimal sin signo, %o un entero octal sin signo, %s una cadena de caracteres o string (las veremos más tarde)...
- más info en man 3 printf.

Para depurar usamos: **fprintf(stderr, “ “, ...)**

Mirar lo de los porcentajes: %c = el carácter en ASCII (hay muchos en el manual)

If

```
if ( expresión ) {
    sentencias1...
} else {
    sentencias2...
}
```

Si la expresión evalúa a un entero distinto de 0, no se entra al if, se entra a else (si lo hay).

- Los paréntesis son obligatorios.
- Si sólo hay una sentencia, podemos prescindir de las llaves.

omitar los if o else innecesarios

Si no hay llaves, solo se toma la primera line de la sentencia

(cada nivel de tabulación son 2 estados posibles)

Switch

```
switch ( expresión ) {
case valor1:
    sentencias1...
case valor2:
    sentencias2...
default:
    sentencias3...
}
```

- El flujo pasa por el case que corresponde al valor de la expresión.
- La sentencia break rompe un bucle o un switch. Si no se rompe al final de un case, se entra a otros cases posteriores.
- Si no entra por ningún case, entrará en default (si lo hay).

si no hay break, hay un salto a una etiqueta

While/do-while

```

        while ( expresión ) {
            sentencias...
        }
y también
do {
    sentencias...
} while ( expresión );

```

- No se sabe más que por una condición las vueltas que va a dar.
- Se itera hasta que la expresión evalúa a 0.
- La sentencia **break** rompe un bucle y **continue** para saltar a la siguiente iteración del bucle y se puede aplicar para cualquier estructura de control.

Ejemplo break:

```

#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

    for(i = 0; i < 10; i++){
        if(i == 5)
            break;
        printf("%d ", i);
    }
    printf("\n");
    exit(0);
}

$ ./br
0 1 2 3 4
$ 

```

Ejemplo continue:

```

$ cat co.c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

    for(i = 0; i < 10;i++){  
        if(i == 5)  
            continue;  
        printf("%d ", i);
    }
    printf("\n");
    exit(0);
}

$ 

```

Se salta todo lo que hay desde el continue hasta el final del bloque en la iteración de $i = 5$.

Ejemplo:

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    while(1){
        printf(".\n");
    }
    exit(0);
}
```

Ejemplo de bucle infinito.

El do-while garantiza entrar en el bucle al menos una vez.

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

    i = 0;
    do{
        printf(".\n");
        i++;
    }while(i%7 != 0);
    exit(0);
}
```

Imprime puntos hasta que se encuentra múltiplos de 7. (del 0 al 6)

For

```
for ( inicialización ; condición ; actualización) {
    sentencias...
}
```

- Se usa para iterar un número de veces conocido.
- La inicialización sólo se ejecuta una vez antes de la primera iteración y antes de evaluar la condición.
- Se itera en el bucle hasta que se deja de cumplir la condición.
- Al final de cada iteración se ejecuta la actualización.
- Ojo, es muy versátil, pero mejor usar unos pocos **idioms**.
- Usarlo con mesura.
- En la inicialización, no declarar variables
- Puede no entrar ninguna vez si la condición es falsa

Idioms de c:

- Bucle **infinito**: **for();{}** (expresión vacía se evalúa a true)
- **for(i = 0; i < 10; i++){ }** (da igual si se pone i++ o ++i pero dejarlo como i++)
- **for(i =9; i >=0, - - i)**
- **for(p = a-> land; p != NULL; p = p->next)**
- **for(p=a; *a!='\0'; a++)** (recorrer el array hasta que encuentras un valor)

Ejemplo:

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

    for(i = 0; i < 10; i++){
        printf("%d\n", i);
    }
    exit(0);
}
```

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

    for(i = 0;;i++){
        printf("%d\n", i);
    }
    exit(0);
}
```

Imprime infinitamente números que se irán incrementando.

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int i;

    for(i = 0;0;i++){
        printf("%d\n", i);
    }
    exit(0);
}
```

No se ejecuta el bucle ninguna vez.

Punteros

- Las variables son una o varias direcciones de memoria contiguas con los bytes correspondientes.
- Un puntero es una variable que contiene una dirección de memoria.
- Para declarar una variable puntero a un tipo:

Ejemplo:

```
int **r; (r contiene la dir de un puntero) es puntero a puntero
int *q;
int a; (si es local está en la pila)
int *p; (dentro de p va a haber una dirección (cada vez en el tiempo) de una variable de tipo entero)
```

a = 3;

p = &a; saca la dirección de la variable en esa llamada a la función (recordar: si es local, su posición cambia) Da la dirección del primer byte de a

printf("%d\n", *p); (va a la variable a la que apunta p) da 3

*p = 28;

q = p; (el contenido de la variable)

printf("%d\n", *q)

q = &p (error de tipos)

r = &p; (puntero a puntero) (es 4)

*r = q; es 0

**r = 12 apunta donde apunta r que es modificar el valor de a (12) . atravesas r que va a p y luego atravesas a

print(*q)

tipodedato * nombre;

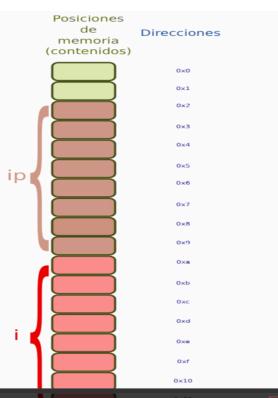
Por ejemplo:

int *ptr; /* un puntero a entero */

- En una sentencia, el operador * (dereference) delante de una variable de tipo puntero significa que queremos operar sobre el contenido de la dirección a la que apunta. (atravesamos el puntero)
- Con el operador & (address of) delante de una variable obtenemos su dirección de memoria.
- No se puede usar un puntero que no apunta a ningún sitio (NULL). Muere el programa
- %p imprimo la dirección en hexadecimal

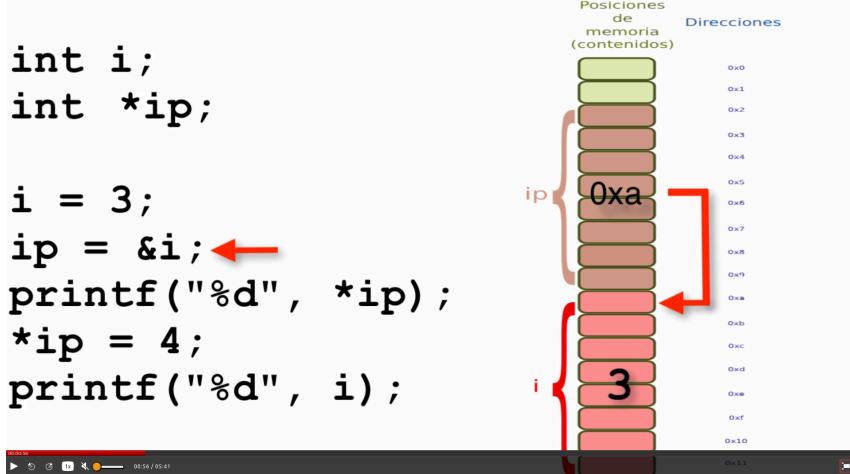
Explicación del video:

```
int i;  
int *ip; ←  
  
i = 3;  
ip = &i;  
printf("%d", *ip);  
*ip = 4;  
printf("%d", i);
```



int *ip (puntero a entero)

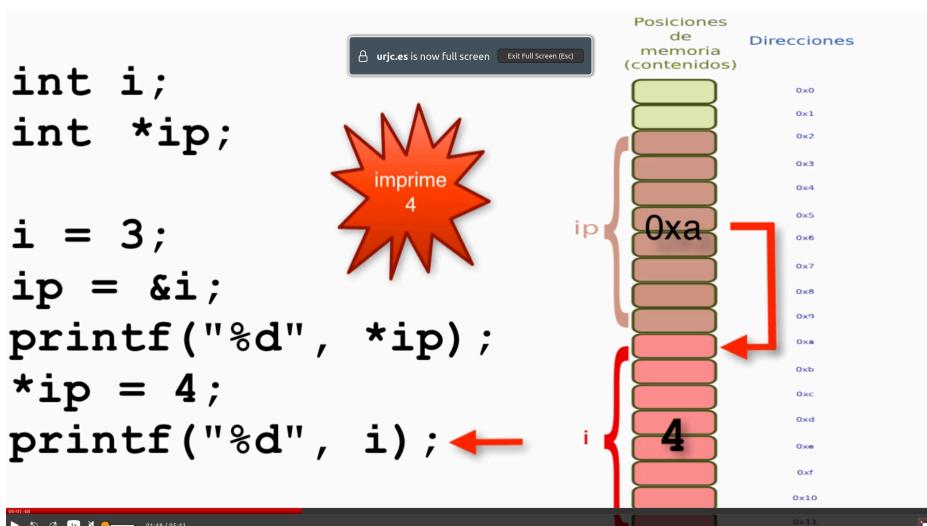
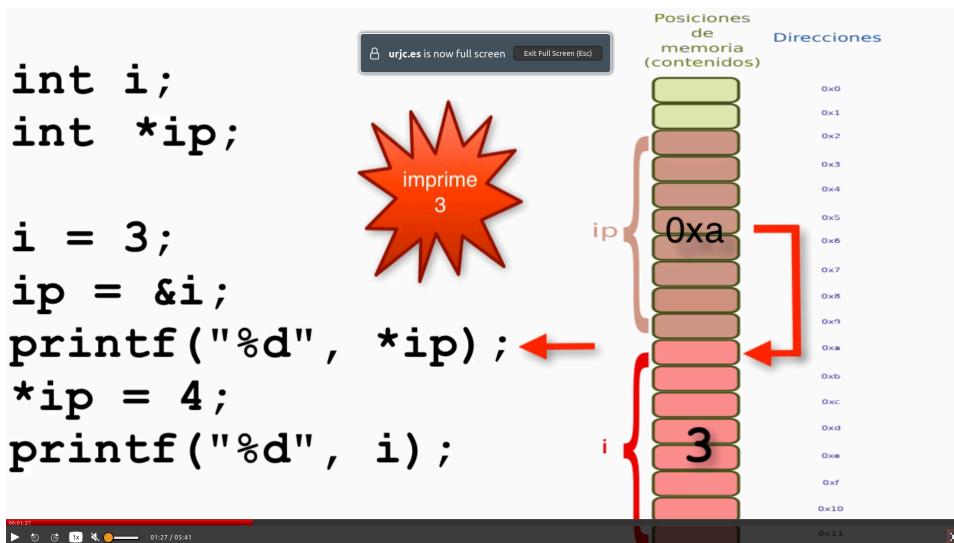
ip = &i (&i = obtiene la dirección de i y lo mete dentro del puntero ip. Ahora el contenido de la posición de memoria representada por la variable ip va a ser la dirección de memoria donde se encuentra i.)



Imprime a donde apunta ip, se suele llamar el contenido donde apunta el puntero. Aunque lo que es, es el contenido de la posición de memoria cuya dirección contiene el puntero.

El contenido del puntero se consigue con el operador asterisco (*).

*ip atraviesa el puntero para obtener el valor de la dirección a la que apunta.



Ejemplo:

int i;
int *ip;
int *jp;

i = 15;
ip = &i; ←
jp = ip;
*ip = 9;
printf("%d", *jp);

Posiciones de memoria (contenidos) Direcciones
0x0
0x1
0x2
0x3
0x4
0x5
0x6
0x7
0x8
0x9
0xa
0xb
0xc
0xd
0xe
0xf
0x10
0x11
0x12
0x13
0x14
0x15
0x16
0x17
0x18
0x19
0x1a
0x1b
0x1c
0x1d
0x1e

int i;
int *ip;
int *jp;

i = 15;
ip = &i;
jp = ip; ←
*ip = 9;
printf("%d", *jp);

Posiciones de memoria (contenidos) Direcciones
0x0
0x1
0x2
0x3
0x4
0x5
0x6
0x7
0x8
0x9
0xa
0xb
0xc
0xd
0xe
0xf
0x10
0x11
0x12
0x13
0x14
0x15
0x16
0x17
0x18
0x19
0x1a
0x1b
0x1c
0x1d
0x1e

int i;
int *ip;
int *jp;

i = 15;
ip = &i;
jp = ip;
*ip = 9; ←
printf("%d", *jp);

Posiciones de memoria (contenidos) Direcciones
0x0
0x1
0x2
0x3
0x4
0x5
0x6
0x7
0x8
0x9
0xa
0xb
0xc
0xd
0xe
0xf
0x10
0x11
0x12
0x13
0x14
0x15
0x16
0x17
0x18
0x19
0x1a
0x1b
0x1c
0x1d
0x1e

Imprimiría 9

& sirve para obtener la dirección de una variable

Aritmética de punteros

- Hace que C sea un lenguaje de bajo nivel
- sumar algo a un puntero es hacer a qué siguiente número apuntaría
- Los punteros se pueden sumar, restar, etc. P.e. para conseguir el tamaño de un buffer.
- Las operaciones se hacen en múltiplos del tamaño en bytes del tipo de datos al que apunta el puntero.

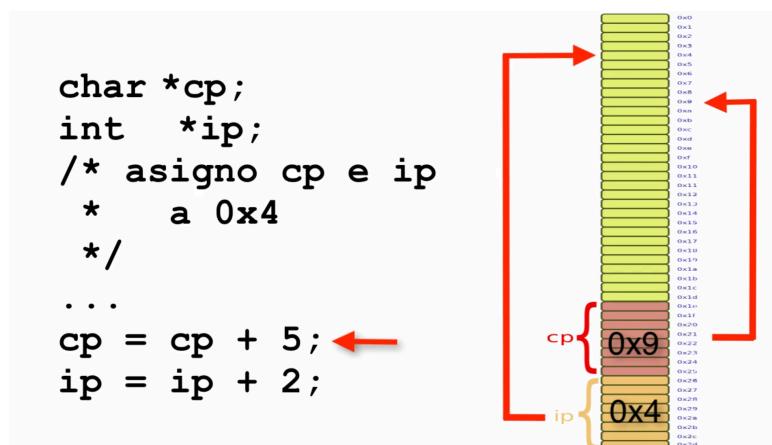
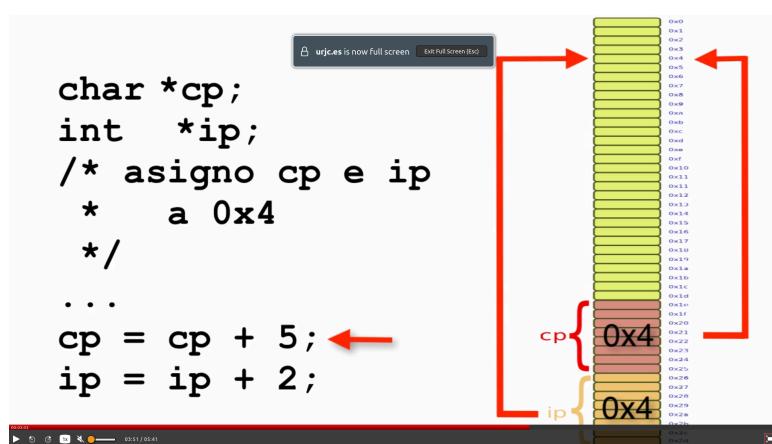
```
char *cptr; /* los char ocupan 1 byte */
int *iptr; /* los int ocupan 4 bytes */
```

...

```
cptr = cptr + 4; /* la dirección de memoria + 4 posiciones */
iptr = iptr + 4; /* la dirección de memoria + (4*4) posiciones */
```

Mirar diapositivas.

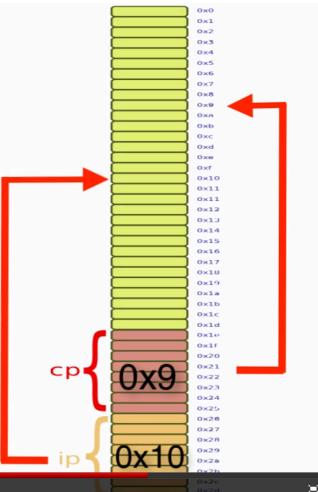
Explicación video:



```

char *cp;
int *ip;
/* asigno cp e ip
 * a 0x4
 */
...
cp = cp + 5;
ip = ip + 2;

```



```

char *p, *q;
int n;

p - q      → int
p + n      → char *
p - n

```

$p - q$ devuelve un entero que se trata de la diferencia de las 2 direcciones de memoria

$p + q$ no tendría sentido

```

char *p;
p = &j;
printf("%p\n", p);

```

A veces es interesante imprimir la dirección contenida en un puntero, se hace con el `%p` e imprime la dirección en hexadecimal.

Los Arrays en C

- El índice para N elementos va de 0 a N-1.
- No se comprueban los límites.
- No son más que azúcar sintáctico para los punteros.
- int lista[N]; → “reserva la memoria necesaria para tener N objetos de tipo int y guarda la dirección en la variable lista”. Lista es un puntero constante al primer byte.
p = lista es igual que p = &lista
lista[1] = 2 es azúcar sintáctico en realidad es *(a+1) = 2
no se puede hacer lista = lista + 2 ya que lista es una constante.
- lista[NUM] = 3; → “escribe el entero 3 en la posición de memoria (NUM * tamaño de int) a partir del puntero lista”.
- El operador sizeof sobre un array devuelve el tamaño de la memoria reservada (NO el número de elementos!).
- Inicialización de arrays:

```
int lista1[5] = { 1, 2, 3, 4, 5 }; /* damos el tamaño e
                                         inicializamos           */
int lista2[] = { 1, 2, 3, 4, 5 };   /* tamaño == numero elementos
                                         en la inicialización   */
int lista3[5] = { 1, 2, 3 };        /* da igual si sobra huecos */
int lista2[4] = { 1, 2, 3, 4, 5 }; /* error!!! ATENCION!!! esto compila
```

Los corchetes ponerlo solo cuando se inicialicen

int lista[]; NO HACERLO INDEPENDIENTEMENTE DE SI ES LOCAL O GLOBAL

int
int lista[]; (pero darle dimensión)
f(int *l){} y luego llamarlo como f(lista)
Si es global, todos los valores se inicializan a 0.
C tiene un comportamiento indefinido = cuidado/buena suerte

int a,b;
a = sizeof b;
a = sizeof (int); da el tamaño de un entero

int lista[3]

a = sizeof(lista); Devuelve en la arquitectura ficticia un valor de 12.

```
f(int*l) {
    sizeof l ( es el valor del tamaño del puntero que artificialmente es 4.
}
```

Explicación del video:

Un ejemplo de un programa erróneo en c.

Badfunc intenta devolver un puntero a un valor de la pila, el problema es que como ya hemos visto, los valores de la pila viven en el registro de activación de esa llamada a la función que se reutilizará esa zona de memoria (la cima de la pila), cuando se llame a la siguiente función. Devolver un puntero a esas posiciones es un ERROR DEL PROGRAMADOR EN C.

```
$ cat main.c
#include <stdlib.h>
#include <stdio.h>

int *
badfunc(int b)
{
    int z;
    z = b;
    return &z;
}

int main(int argc, char *argv[])
{
    int *ip;

    ip = badfunc(2);
    printf("val is %d", *ip);
    exit(0);
}
$
```

Otro ejemplo erróneo:

Usar aritmética de punteros y salirse de las posiciones de memoria en las que se hay algo.

```
$ cat over.c
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int *ip;
    int i;

    ip = &i;
    ip += 5;
    printf("%d\n", *ip);
    exit(0);
}
$
```

ip apunta a la variable i y luego le suma 5.

```
[*] $ ./over
-845149215
```

Otros operadores:

a[c]

f -> y

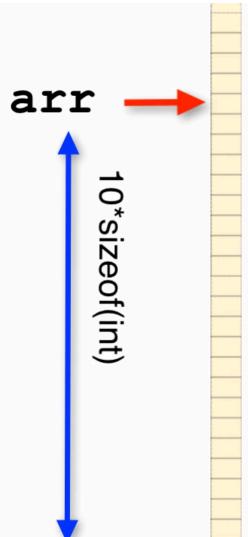
Son azúcar sintáctico, el compilador cuando los ve los reescribe en términos de operadores que ya hemos visto.

a[c] —> *(a + c)

f -> y —> (*f).y

Ejemplos:

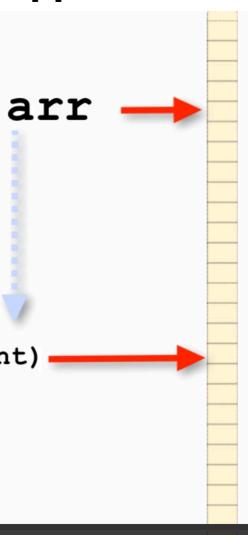
```
int arr[10];  
arr[3] = 4;
```



arr es un puntero constante, por eso podemos hacer arr[3] = 4

```
int arr[10];  
arr[3] = 4;
```

arr+3*sizeof(int)



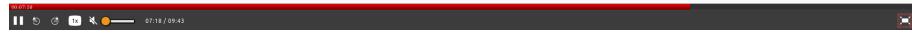
```
int arr[10];  
int *ip;
```

ip = arr;  Equivalentes
ip = &arr;
n = sizeof(arr); //10*sizeof(int)

Son lo mismo porque la dirección de arr se lo interpreta como la dirección del array y la dirección del array es la posición de su primer elemento y arr a su vez es un puntero constante a esa posición.

!!CUIDADO: Si ahora a una función que recibe un puntero a int, le paso arr y hago sizeof arr de ese argumento, el compilador solo sabe de ese argumento que va a ser un puntero a int, no necesariamente un array. Con lo cual, el tamaño del puntero va a ser el tamaño de la variable que es el puntero y va a depender de la arquitectura y no va a ser el tamaño del array.

```
int  
indsz(int *p)  
{  
    return sizeof(p);  
}  
  
void dif(void)  
{  
    int nb[5];  
    printf("%d %d", sizeof(nb), indsز(nb));  
}
```

 sizeof(p) va a ser el tamaño de un puntero en esa arquitectura, que puede ser 8 o lo que sea.
Sizeof(nb) sí que nos dará el tamaño de nb completo

Pasando direcciones de memoria como argumento

```
#include <stdlib.h>  
#include <stdio.h>  
  
void  
dameletra(char *c)  
{  
    *c = 'A';  
}  
  
int  
main(int argc, char *argv[])  
{  
    char c = 'b';  
  
    dameletra(&c);  
    printf("c es %c\n", c);  
    exit(EXIT_SUCCESS);  
}
```

Forma de hacer a mano el paso por referencia

Si quitamos & y *, el valor no se modificaría.

Cadenas de caracteres (string)

Son arrays de caracteres acabados en un carácter '\0' (el carácter nulo).

- Si no se acaba en un nulo, no es una string.
- Inicializar una string:

```
char str[] = "hola"; /* inicializando una string */  
char str2[] = {'h', 'o', 'l', 'a', '\0'}; /* equivalente a lo anterior */
```

char *p;
p = "hola"; (llevan implícito \0) Región memoria inmutable.

***MIRAR DIAPOSITIVA

si *p = 'c' error

Funciones para manejo de cadenas (ver prototipos en las páginas de manual):

- **snprintf**: similar a printf, pero imprime en una cadena. Escribe como mucho el número de bytes que se especifica en el segundo argumento, contando el carácter nulo. Devuelve el número de caracteres escritos en la cadena.

```
char cad[255];  
snprintf(255,cad, "hola %d", 3)  
imprime hola en cad
```

- **strlen**: devuelve el tamaño de una cadena, sin contar el carácter nulo.

```
strlen("hola") devuelve 4
```

- **strcat**: concatena dos cadenas, la segunda al final de la primera, dejando el resultado en la primera. Devuelve un puntero a la cadena resultante. La primera cadena tiene que tener espacio suficiente como para que quepa la concatenación.

- Cómo no comparar las strings:

```
char *p;  
char *s;  
p = "hola"  
s = "hola"
```

```
if( p == s) { print(bla) }
```

Cómo ordenar:

```
if (strcmp(p,s) == 0)
    printf("bla")
```

strcmp ordena lexicográficamente: orden de diccionario.

Explicación video:

Cualquier puntero a char que apunte a una dirección de memoria en la que haya caracteres acabados en un carácter NULL o '\0' es una string en C.

Ejemplo:

*str = "hola"; ("hola" reserva memoria en el segmento de datos en una zona de solo lectura. Dentro de esa memoria hay 5 caracteres: 'h', 'o', 'l', 'a', '\0'. Str apunta a esa dirección de memoria a la primera posición; en realidad apunta a h)

char strx[3] = "ah" (inicialización del array y es de tamaño 3 porque tendrá 'a', 'h', '\0')

Funciones:

- **strlen**: devuelve la longitud de una string
No tiene en cuenta el '\0'
strnlen es una versión modificada de strlen para darle un tamaño máximo. Esto es útil si no sabemos exactamente de dónde viene la string.
- **strcmp o strncmp**: compara 2 strings lexicográficamente (orden de diccionario)
Devuelve 0 si son iguales. (no se puede usar como booleano)
- **strcpy**: copia una string a otra. El puntero a char de destino tiene que tener memoria suficiente para almacenar los caracteres de origen y su '\0'.

Argumentos de main

- **argc**: variable entera que indica el número de argumentos que se le han pasado a main.
- **argv**: array de strings con cada uno de los argumentos. El primer argumento se corresponde con el nombre del programa que se invoca. Desde UNIX V7, siempre va terminado con un NULL.
Apunta a un sitio en el que hay punteros a char (direcciones de chars)

Registros

```

struct Coordenada{
    int x;
    int y;
};

struct Coordenada c = {13, 33}; /* inicialización */

```

- El tamaño que ocupa en memoria no tiene porqué coincidir con la suma de los tipos de datos que contiene la estructura.
- Sólo se pueden hacer 3 cosas con ellas: copiarlas/asignarlas (esto incluye pasárselas como parámetro o retornarla), obtener su dirección (&), y acceder a sus campos.

```

struct Coordenada{
    int x;
    int y;
};

typedef struct Coordenada Coordenada; /* definición de tipo Coordenada */
Coordenada c = {13,31};           /* declaración e inicialización */

```

- Se suele definir un tipo de datos nuevo con `typedef` para usarlas de forma más cómoda.
- Si tenemos un puntero a una estructura, el operador `->` sirve para acceder a sus campos:

$$p->x \equiv (*p).x$$

Explicación del video:

Declaración del tipo de datos

```

struct point {
    int x;
    int y;
};

```

Uso del tipo de datos

```

struct point p;
p.x = 3;
p.y = 8;

```

```

$ cat main.c
#include <stdlib.h>
#include <stdio.h>

struct point {
    int x;
    int y;
};

int
main(int argc, char *argv[])
{
    struct point p;
    p.x = 3;
    p.y = 5;
    printf("p.x vale %d, p.y vale %d\n", p.x, p.y);
    exit(0);
}
$ 

```



```
p.x vale 3, p.y vale 5
$
```

Otro ejemplo:

```
struct point {
    int x;
    int y;
};

void
reg(struct point p)
{
    p.x = 3;
    p.y = 5;
    printf("reg: p.x vale %d, p.y vale %d\n", p.x, p.y);
}

int
main(int argc, char *argv[])
{
    struct point p;

    reg(p);
    printf("reg: p.x vale %d, p.y vale %d\n", p.x, p.y);
    exit(0);
}
$
```

```
$ cc -g -c -Wall -Wshadow mainp.c
$ cc -o mainp mainp.o
$ ./mainp
reg: p.x vale 3, p.y vale 5
reg: p.x vale 0, p.y vale 0
$
```

Recordemos que los argumentos se pasan siempre por copia en C y reg tiene su propia copia del registro dentro de la pila y modifica esa copia.

Cuando volvemos a main tenemos la copia original en la pila de la variable local de main que está sin inicializar (casualmente está a 0 aunque podría tener cualquier otro valor)

En C uno puede definir tipos de datos compuestos por otro tipo de datos compuesto.

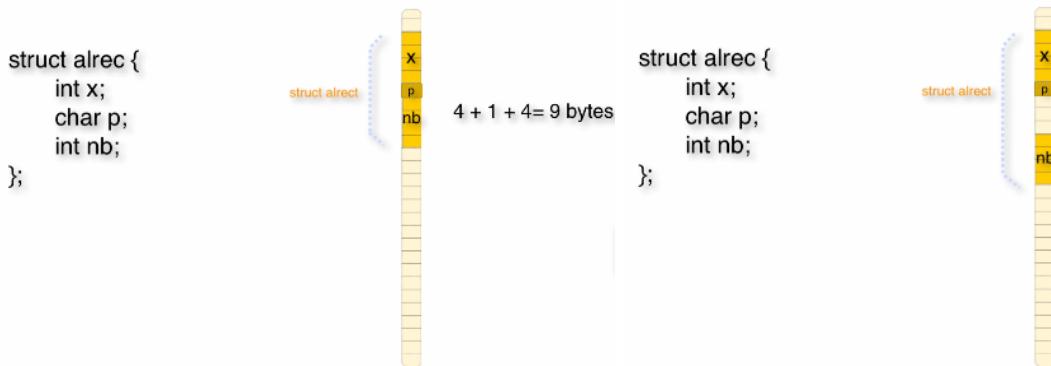
Ejemplo:

```
struct point {
    int x;
    int y;
};

struct rect {
    struct point upleft;
    struct point dnright;
};

struct rect r;
r.upleft.x = 5;
```

Cómo se distribuye en memoria los bytes de un record/struct, depende de cada compilado. Es probable que ocupe más de lo pensado debido al **alineamiento** (las restricciones en algunas arquitecturas, los campos empiezan en algún múltiplo de 2)



Sizeof nos dirá lo que ocupa realmente un record/struct.

Es posible en c que aparezcan en memoria de forma distinta a lo que aparece en la declaración.

En c se puede redefinir el nombre de un tipo de datos con otro nombre, establece un alias para el mismo tipo de datos. No es muy común usarlo salvo para algunos casos.

Ejemplo:

```

struct Point {
    int x;
    int y;
};

typedef struct Point Point;

```

Cuando vea Point lo sustituirá como struct Point.

```

Point p;
p.x = 3;
p.y = 8;

```

Memoria dinámica

Free y malloc (leer la página de manual):

- `malloc`: sirve para pedir memoria en tiempo de ejecución. La memoria devuelta se localiza en el heap.
- La memoria reservada con `malloc` puede tener cualquier contenido.
- Si no hay memoria en el sistema, devuelve NULL.
- `free`: sirve para liberar la memoria devuelta anteriormente por `malloc`. No se puede liberar memoria que no se ha solicitado con `malloc`.

- Hay que liberar la memoria cuando ya no nos hace falta.

Programas con varios ficheros fuente

- Las variables globales que declara un fichero externo tienen que definirse como `extern`.
- Una variable tiene que estar declarada en un fichero fuente.
- Una función o variable global declarada como `static` no es visible desde otros ficheros. Si no se especifica, si son visibles.
- Las variables, tipos de datos, constantes, etc. compartidas por los ficheros fuente de un programa deberían estar en un fichero de cabeceras.
- Cada fichero fuente debe incluir los ficheros de cabeceras que necesite. No es buena idea incluir ficheros de cabecera en otros ficheros de cabecera.
- Para incluir un fichero de cabeceras que no está en los directorios del sistema (`/usr/include,...`):
`#include "rutadelfichero"`
 (si no lo encuentra, lo busca entre los directorios del sistema)
- No se deben incluir dos veces un mismo fichero de cabecera.

gdb

El comando `gdb` es un depurador que nos permite:

- Inspeccionar un programa (p. ej. desensamblar).
- Inspeccionar un proceso (p. ej. ver los valores de la memoria).
- Inspeccionar un core: es la foto de un proceso en un fichero.
- En la mayoría de las ocasiones, **no es la forma más eficiente de depurar un programa.**

1. Arrancamos gdb con el ejecutable:
`gdb ejecutable`
2. Ejecutamos dentro de gdb:
`run argumento1 argumento2 ...`
3. ... fallo en ejecución ...
4. `bt` #vuelca la pila
5. `frame 3` #selecciono el registro de activación que deseo inspeccionar
6. `info locals` #veo el valor de las variables locales
7. `info args` #veo el valor de los argumentos
8. `what is z` #veo el tipo de la variable z

Meter punto de ruptura y ejecutar paso a paso:

1. Arrancamos gdb con el ejecutable:
`gdb ejecutable`
2. `break f1` #mete punto de ruptura en la función f1
3. Ejecutamos dentro de gdb:
`run argumento1 argumento2 ...`
4. se para en f1, ahora podemos inspeccionar como en el ejemplo anterior.
5. `stepi` #ejecuta una instrucción
6. ...
7. `continue` # sigue ejecutando normalmente.

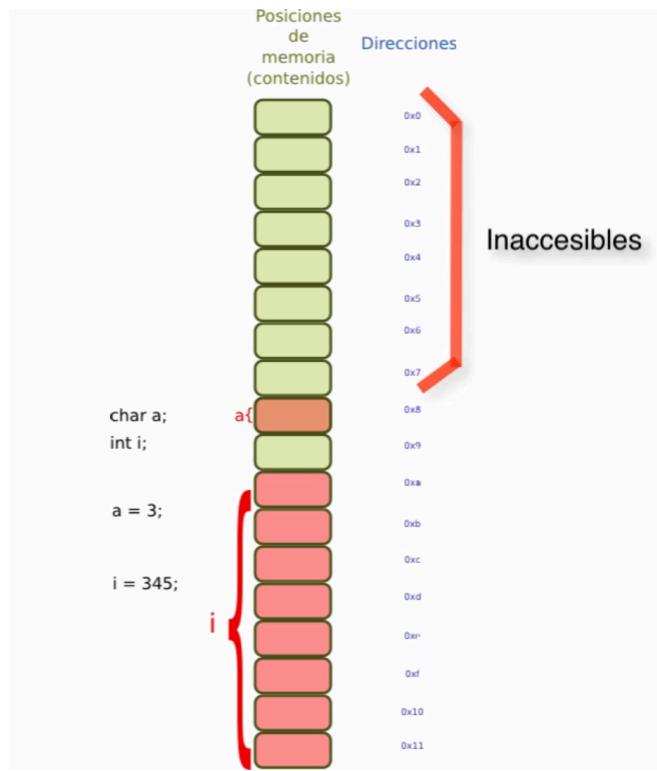
Análisis dinámico: Valgrind

- Ayuda para encontrar errores y problemas automáticamente
- Hace análisis dinámico del código en ejecución (instrumentado o sin instrumentar)
- Leaks, corrupción de memoria, patrones incorrectos de uso de los recursos
- <http://valgrind.org>
- Para asegurarte de que no hay leaks y ver dónde están:
`valgrind --leak-check=full programa`

```
$ valgrind ./a.out lespaul sg telecaster stratocaster
==6491== Memcheck, a memory error detector
==6491== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6491== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==6491== Command: ./a.out lespaul sg telecaster stratocaster
==6491==
[1] lespaul
[2] sg
[3] telecaster
[4] stratocaster
The largest argument is: stratocaster
==6491==
==6491== HEAP SUMMARY:
==6491==     in use at exit: 19 bytes in 2 blocks
==6491==   total heap usage: 4 allocs, 2 frees, 1,056 bytes allocated
==6491==
==6491== LEAK SUMMARY:
==6491==     definitely lost: 19 bytes in 2 blocks
==6491==     indirectly lost: 0 bytes in 0 blocks
==6491==     possibly lost: 0 bytes in 0 blocks
==6491==     still reachable: 0 bytes in 0 blocks
==6491==           suppressed: 0 bytes in 0 blocks
==6491== Rerun with --leak-check=full to see details of leaked memory
==6491==
==6491== For lists of detected and suppressed errors, rerun with: -s
==6491== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
$
```

Memoria

La memoria es una sucesión de bytes numerados (son las direcciones). Los bytes en sí mismos son las posiciones de memoria o el contenido de la memoria.



char a; (declaramos un carácter, 1byte, reservamos una de esas posiciones de memoria y damos nombre)

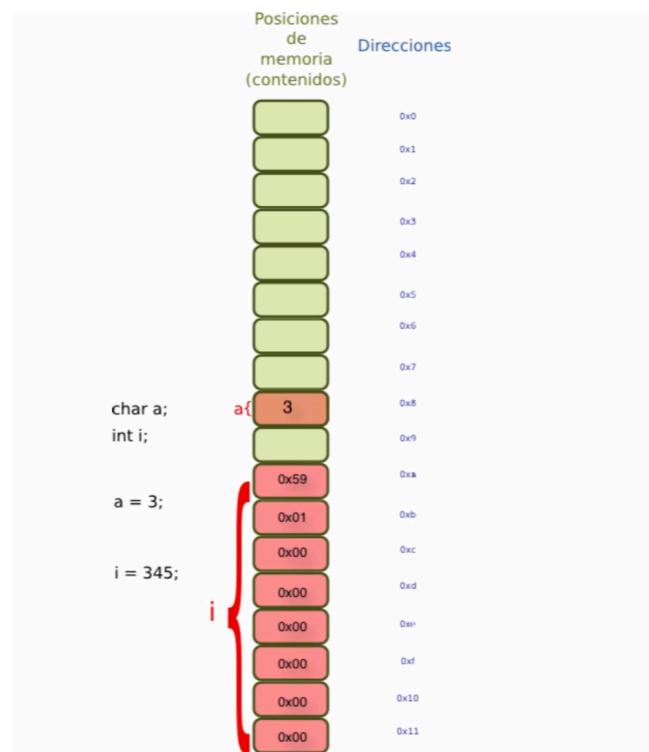
int i (vale 8 bytes, comprobarlos con sizeof)

a = 3 (asignamos 3 a a, metemos en el contenido del byte que representa a de dir 0x8, un 3)

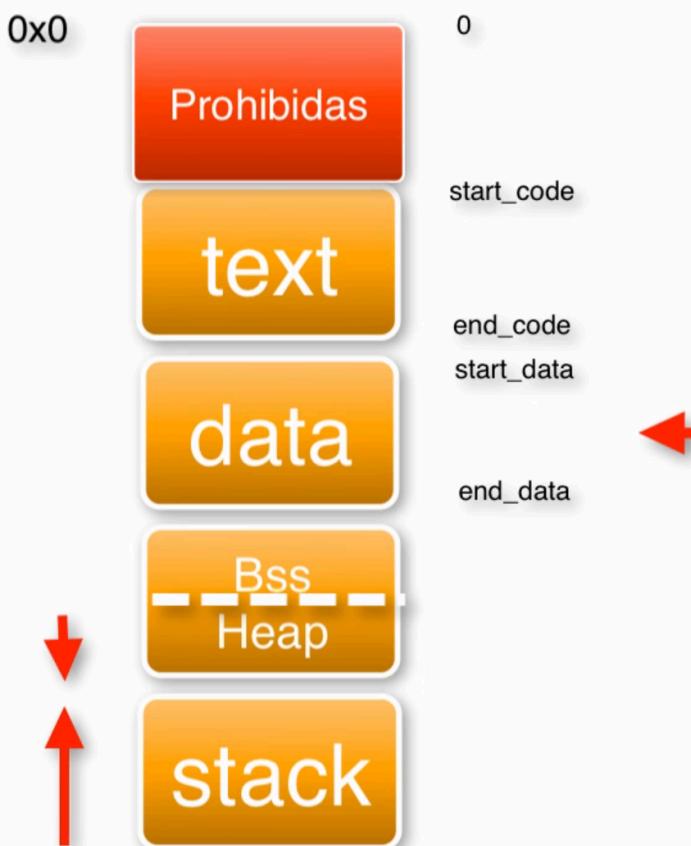
i = 345 (lo que va a hacer es meter cada una de las partes que compone 345, se divide en bytes, dentro del byte correspondiente).

¿Cómo se hace esto?

Depende con lo de endianes y pasamos el decimal a hexadecimal



La memoria se divide en regiones o segmentos:



text: suele contener el **código máquina**.

data: contiene las **variables globales inicializadas**.

bss: las variables globales sin inicializar.

heap: se puede reservar memoria dinámica para malloc/free.

stack: se guarda las direcciones de retorno, argumentos y registros de activación.

Los segmentos pueden crecer o decrecer.

Puede haber más de un segmento repetido dependiendo del sistema operativo que nos encontremos.

Funciones

```
$ cat main.c
#include <stdlib.h>
#include <stdio.h>

int scale(int a, int b)
{
    return (a - b) * 5;
}

int main(int argc, char *argv[])
{
    int x, y;

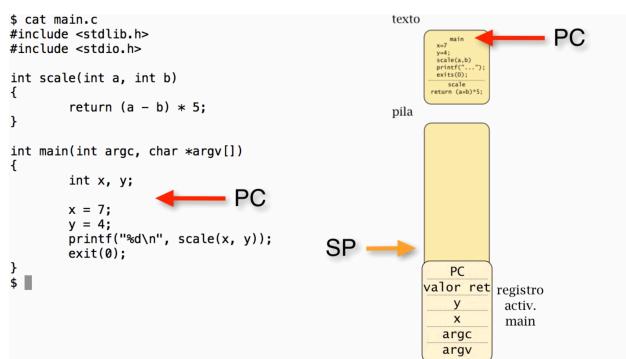
    x = 7;
    y = 4;
    printf("%d\n", scale(x, y));
    exit(0);
}
$ cc -g -c -Wall -Wshadow main.c
$ cc -o main main.o
$ ./main
15
```

Demostración de qué hace este programa paso a paso con la memoria:

SP: puntero de pila apuntando encima de la pila

PC: El contador de programa apunta en el segmento de texto a la instrucción que se está ejecutando en ese momento.

1º El programa llamará a main, va a meter en la pila el registro de activación, el valor de retorno , los argumentos y las variables locales.



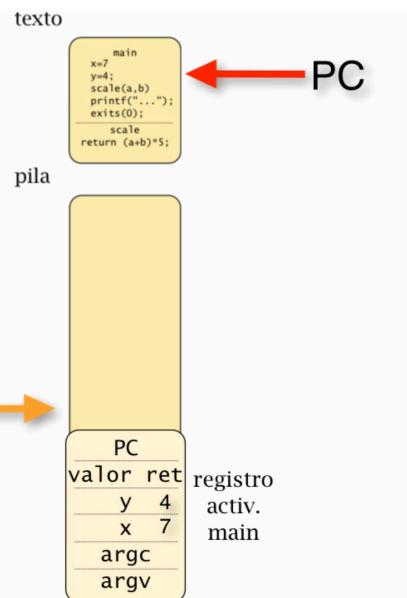
2º las asignaciones 7 y 4 a las variables x e y.

```
$ cat main.c
#include <stdlib.h>
#include <stdio.h>

int scale(int a, int b)
{
    return (a - b) * 5;
}

int main(int argc, char *argv[])
{
    int x, y;

    x = 7;
    y = 4;
    printf("%d\n", scale(x, y));
    exit(0);
}
```



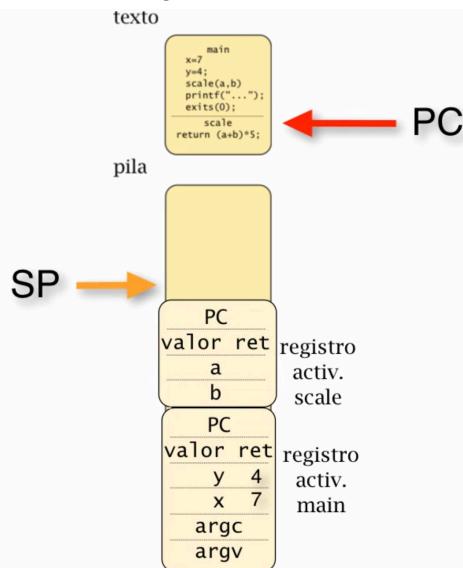
3º Los parámetros se pasan por copia a la pila al registro de activación de scale.

```
$ cat main.c
#include <stdlib.h>
#include <stdio.h>

int scale(int a, int b) ← PC
{
    return (a - b) * 5;
}

int main(int argc, char *argv[])
{
    int x, y;

    x = 7;
    y = 4;
    printf("%d\n", scale(x, y));
    exit(0);
}
```



4º Se calcula el valor de retorno que va en la pila

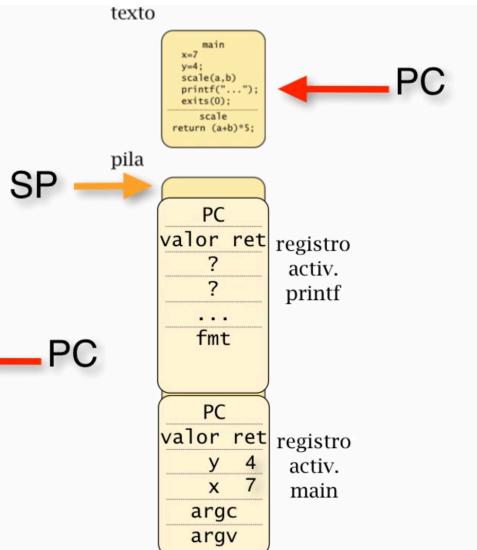
5º El programa vuelve a llamar a otra función que es printf

```
$ cat main.c
#include <stdlib.h>
#include <stdio.h>

int scale(int a, int b)
{
    return (a - b) * 5;
}

int main(int argc, char *argv[])
{
    int x, y;

    x = 7;
    y = 4;
    printf("%d\n", scale(x, y));
    exit(0);
}
```



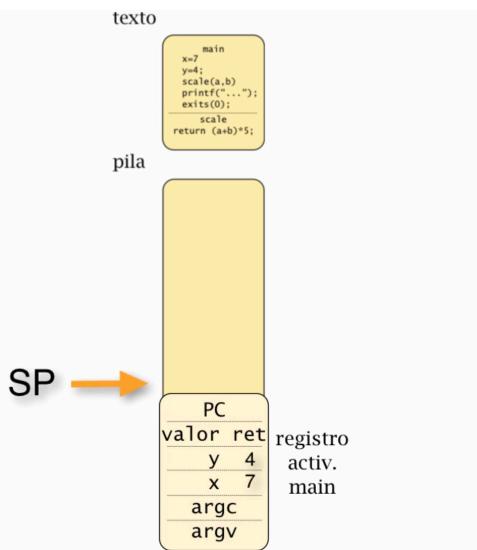
6º y último, se saca de la pila

```
$ cat main.c
#include <stdlib.h>
#include <stdio.h>

int scale(int a, int b)
{
    return (a - b) * 5;
}

int main(int argc, char *argv[])
{
    int x, y;

    x = 7;
    y = 4;
    printf("%d\n", scale(x, y));
    exit(0);
}
```



Cada arquitectura tiene su propio funcionamiento..

Para optimización, se usará registros.