



Repaso de programación en C

Programación en C

- ◆ C es un lenguaje *imperativo* (= le decimos paso a paso lo que tiene que hacer)
- ◆ Maneja
 - ❖ Variables de distintos *tipos*
 - ✓ En general C es MUY Estricto con los tipos: no deja mezclar
 - ✓ ... salvo con punteros
 - ❖ *Constantes*
 - ❖ *Funciones* = bloque de código al que se le puede pasar varios parámetros y devuelve resultados
 - ✓ Nos facilita reutilizar código: escribo algo que se comporta de forma levemente distinta según los argumentos de entrada
 - ❖ *Estructuras de control*: if, bucles, ...
- ◆ Escribimos uno o varios ficheros de texto, y con unas herramientas (compilador y enlazador) obtenemos UN fichero ejecutable

Ejemplo

◆ Fichero: holaMundo.c

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

int main (void)
{
    printf("Hello World\n");
    exit(0);
}
```

- ◆ (En Linux) se compila usando gcc
gcc -o holaMundo holaMundo.c
- ◆ Se ejecuta con
./holaMundo

```
it001:~/tmp> gcc -o holaMundo holaMundo.c
it001:~/tmp> ./holaMundo
Hello World
it001:~/tmp> █
```



Entendiendo el ejemplo

- ◆ Todo programa en C tiene una función llamada `main`, que es donde comienza la ejecución
- ◆ Puede ser de la siguiente forma

```
int main (void) { /* código */... },
```

Información que
retornará la función

Información recibida por la función

- ◆ Opcionalmente, el programa puede tener otras funciones que son llamadas desde el interior de `main`
 - ❖ Si no son llamadas desde `main`, no se ejecutan

Entendiendo el ejemplo

- ◆ Hay cosas que C NO sabe hacer (ni nosotros tampoco):
 - ❖ Acceder a ficheros, escribir cosas en la pantalla, leer del teclado...
 - ❖ ¡C sólo sabe hacer operaciones matemáticas sencillas y mover/copiar datos de un lado a otro!
- ◆ Para solucionarlo utilizamos **librerías del sistema**:
 - ❖ Funciones que han escrito otros (ej., los que han hecho el Sistema Operativo, los que han hecho el compilador...)
 - ❖ Las funciones están declaradas en ficheros ***.h** (*ficheros de cabeceras*)
 - ✓ Hay que incorporar las declaraciones de las funciones en nuestro fichero

```
#include <stdio.h>
#include <stdlib.h>
```
 - ❖ `#include` sustituye la línea en la que está por el fichero completo
 - ❖ Gracias a
 - ✓ `stdio.h`, el compilador sabe que hay una función llamada `printf()`, que sirve para escribir cosas por la pantalla
 - ✓ `stdlib.h`, conoce la función `exit()` que sirve para terminar y devolver un valor a la línea de comando (típicamente -1 si ha habido un error, 0 en caso contrario)
 - ✓ El `#include` sólo indica la declaración de la función (= el nombre, el valor que devuelve, y los argumentos que toma)
 - ❖ Más ficheros que declaran funciones en `/usr/include`
 - ❖ El compilador sabe encontrar el código que **realmente** hace el trabajo de `printf` y `exit`
 - ✓ Está en una *librería* llamada `glibc`, que el compilador conoce sin necesidad de que le tengamos que decir nada



Tipos de datos y operadores

◆ Tipos de datos básicos

- ❖ **Tipos:** char, int, float, double
 - ✓ char es un byte.
 - ✓ Recordar que los números representan caracteres (según ASCII)
- ❖ **Calificadores:** short (16 bit), long (64 bit), unsigned, signed, const
- ❖ **Ejemplo de declaración de variables**
`unsigned int variable1, variable2;`
- ❖ **Declaro y asigno valor inicial**
`int indice = 0;`
- ❖ **Asignación**
`variable1 = 2; variable2 = variable1 + 9;`

◆ **Aritmética:** +, -, *, /, %

- ❖ Sólo con números, y sólo entre números del mismo tipo
- ❖ Otro tipo de aritmética con funciones; ejemplo
`log (3.4556) /*logaritmo decimal, utilizando función
definida en #include math */
variable1++ ; /* es igual que
variable1 = variable1 + 1 */`



Tipos de datos y operadores

◆ Constantes

```
#define constante1 0x1234
```

```
#define texto "Cadena de caracteres"
```

◆ Relaciones y lógica: <, >, <=, >=, ==, !=, &&, ||

(1 && 0) es igual a 0

Ojo: ASIGNACIÓN es =, COMPARACION es ==

Cuidado con asignar cuando se quiere comparar!

if (a = 3) ... ¡Siempre verdadero!

Aritmética entera

◆ ¡Cuidado con orden de las operaciones!

```
int resultado;
```

```
resultado = 30*400/1000;
```

```
    /*resultado es 12, lo que queríamos*/
```

```
resultado = 30/1000*400
```

```
    /* resultado es 0, no deseado */
```

◆ Dividir siempre **al final**

- ❖ Con *aritmética flotante*, el orden de las operaciones es menos relevante, pero en *aritmética entera* sí lo es

Arrays

- ◆ **Un array es una serie de N elementos del mismo tipo**

```
int x[4]; /* existen cuatro variables de tipo entero, referibles como x[0],  
x[1], x[2] y x[3] */  
    ¡NO existe x[4]!  
char texto[10];
```

- ◆ **Los arrays son muy útiles para utilizar con estructuras de control ‘repetitivas’ (bucles...) sobre datos del mismo tipo**

- ◆ **Strings son cadenas de caracteres (arrays de char) especiales: siempre terminadas en un 0 (byte con valor 0 en uno de los char)**
 - ❖ El valor (numérico) 0 se escribe en una variable de tipo ‘carácter’ como ‘\0’

```
texto[0]='h'; texto[1]='o', texto[2]='l',  
texto[3]='a'; texto[4]='\0'; /* 'backslash' cero */
```

¡El espacio necesario para almacenar una palabra de 4 caracteres ('hola') son CINCO chars!

```
printf trabaja con strings  
printf ("%s", texto); /* imprime 'hola' */
```



Condiciones

/* defino una función que compara un número con 10.
La función indica si es mayor/menor/igual devolviendo
'1' si es igual, '0' en caso contrario */

```
int comparaCon10 (int a) {  
    if (a < 10) {  
        printf("a es menor que 10\n");  
        return 0; /* no es igual que 10 */  
    }  
    else if (a == 10) {  
        printf("a es igual a 10\n");  
        return 1;  
    }  
    else {  
        printf("a es mayor que 10\n");  
        return 0;  
    }  
}
```



==

Bucles

```
while (n != 10) {  
    ... /* haz cosas */  
}
```

```
do {  
    ... /* haz cosas */  
} while (m < 20);
```

Bucles

```
for (i = 0; i < MAXVAL; i++) {  
    ... /* haz cosas */  
}
```

Es una abreviatura para

```
i = 0;  
while (i < MAXVAL) {  
    ... /* haz cosas */  
    i++;}
```



Funciones

- ◆ Los argumentos de las funciones SIEMPRE se pasan en C *por valor*
= al entrar en la función se **copia** el valor, la función **NO modifica** la variable original

```
int i=10;  
int resultado;  
  
main () {...  
    resultado = elevaASiMismo(i);  
    printf ("%d elevado a si mismo es %d", i, resultado);  
    /* i sigue valiendo 10? O vale 100? */  
}
```

Distintos!

```
int elevaASiMismo (int n) {  
    int i, resultado=1;  
    for (i = 0; i < n; i++) { resultado = resultado * n;}  
    n = resultado;  
    return n;  
}
```

Funciones

```
int i=10;  
int resultado;
```

```
main () {...  
  
    resultado = elevaASiMismo(i);  
    printf ("%d elevado a si mismo  
           es %d", i, resultado);  
}
```

```
int elevaASiMismo (int n) {  
    int i, resultado = 1;  
    for (i = 0; i < n; i++) {  
        resultado = resultado * n;}  
    n = resultado;  
    return n;  
}
```

Contenido	Nombre variable
10	<i>global_i</i>
?????	<i>global_resultado</i>



Funciones

```
int i=10;  
int resultado;
```

```
main () {...
```

```
    resultado = elevaASiMismo(i);  
    printf ("%d elevado a si mismo  
           es %d", i, resultado);
```

```
}
```

Copia al entrar en la función

```
int elevaASiMismo (int n) {  
    int i, resultado = 1;  
    for (i = 0; i < n; i++) {  
        resultado = resultado * n;}  
    n = resultado;  
    return n;  
}
```

Contenido	Nombre variable
10	<i>global_i</i>
?????	<i>global_resultado</i>
10	<i>elevaASiMismo_n</i>
0, 1, ...	<i>elevaASiMismo_i</i>
1, 10, ...	<i>elevaASiMismo_resultado</i>



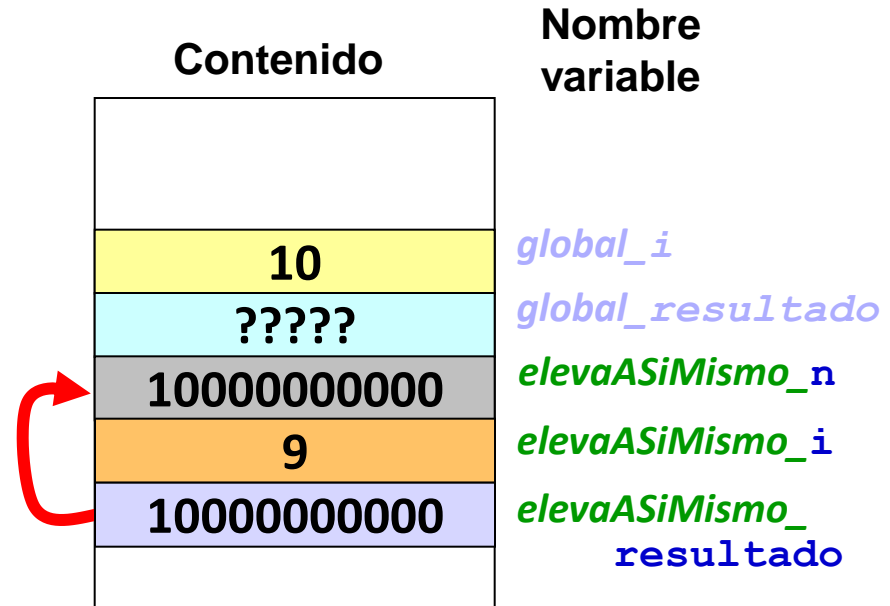
Funciones

```
int i=10;
int resultado;

main () {...

    resultado = elevaASiMismo(i);
    printf ("%d elevado a si mismo
            es %d", i, resultado);
}

int elevaASiMismo (int n) {
    int i, resultado = 1;
    for (i = 0; i < n; i++) {
        resultado = resultado * n;}
    n = resultado;
    return n;
}
```



Funciones

```
int i=10;  
int resultado;
```

```
main () {...
```

```
    resultado = elevaASiMismo(i);  
    printf ("%d elevado a si mismo  
            es %d", i, resultado);
```

```
}
```

return copia valor

```
int elevaASiMismo (int n) {  
    int i, resultado = 1;  
    for (i = 0; i < n; i++) {  
        resultado = resultado * n;}  
    n = resultado;  
    return n;  
}
```

Contenido	Nombre variable
10	<i>global_i</i>
10000000000	<i>global_resultado</i>
100000000000	<i>elevaASiMismo_n</i>
9	<i>elevaASiMismo_i</i>
100	<i>elevaASiMismo_resultado</i>



Funciones

- ◆ **A veces quiero que la función PUEDA CAMBIAR el valor de la variable que utilizo al llamarla**
 - ❖ Porque quiero que la función devuelva varios resultados (y no sólo uno, como permiten las funciones de C en general)
 - ❖ Porque quiero que devuelva algún resultado complejo, y las funciones solo pueden retornar tipos sencillos (`int`, `float`...)
- ◆ **Para esto utilizo PUNTEROS**
 - ❖ Los punteros también se utilizan para más cosas (estructuras de datos complejas como *listas enlazadas*, etc.), ...
 - ✓ En nuestro proyecto sólo los utilizaremos para pasar/retornar valores al llamar a funciones

Punteros

◆ Para cualquier tipo T, podemos formar un ‘puntero a (tipo T)’

- ❖ El valor del puntero es la primera dirección de la zona de memoria que ocupa la variable de tipo T
- ❖ Ejemplos **de declaraciones** de punteros:

```
int *i;  
char *x;
```

◆ Operadores de punteros:

* puntero

obtiene el **valor** de un puntero (retorna el valor al que apunta el puntero)

& variable

obtiene la **dirección** de una variable (retorna la primera dirección de memoria de la posición en la que está almacenada una variable – la variable no tiene por qué ser un puntero, y en general no lo es)



Punteros

Para los ejemplos, supongo que valores int y punteros ocupan todos 4 bytes. Esto no tiene por qué ser así.

```
int res;
```

```
int *y;
```

```
int x;
```

```
x = 4;
```

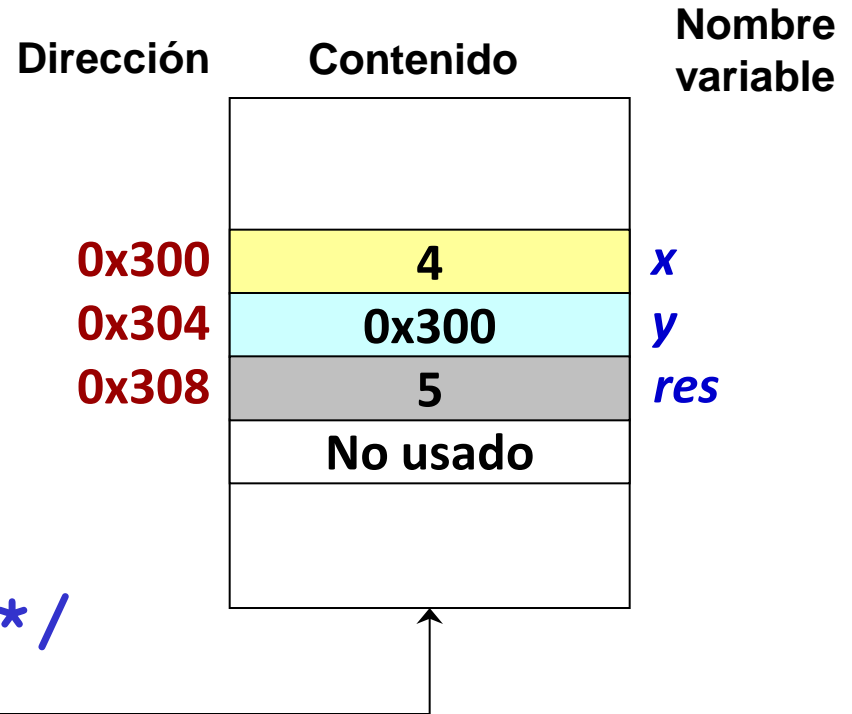
```
y = &x;
```

```
res = *y+1; /* da 5*/
```

● Estado en este punto

```
x++;
```

```
res = *y+1 /* da 6 */
```



Punteros y Arrays

- ◆ Un identificador de un array es equivalente a un puntero que apunta al primer elemento del array

```
int array[5];  
int *ptr;  
ptr = &array[0] es equivalente a ptr = array;
```

- ◆ Si una función se define como

```
int func (int *ptr_dato);  
Entonces también vale llamarla con func(array);
```

- ◆ Aritmética de punteros y arrays

- ❖ array[2] es igual que *(ptr + 2),
 - ✓ El compilador asume que (ptr+2) significa apuntar a 2 objetos del tipo de lo apuntado (int) más allá de ptr



void *

- ◆ **void *** es un tipo de datos especial que no admite
 - ❖ el operador *
 - ❖ ni la aritmética de punterosSíplemente, apunta a un sitio
- ◆ **Puedo aplicar aritmética de punteros si 'fuerzo' un tipo al puntero**
 - ❖ A esto se le llama hacer un *cast*

```
void * punt_void;  
int * numero;  
punt_void = 0x...; /* apunto a una dirección */  
numero = (int *) punt_void + 2;  
/* apunta a un espacio 'dos enteros después' del espacio  
inicial punt_void */
```



¡Cuidado con los punteros!

◆ Valores apuntados inesperados

- ❖ En C, al definir una variables lo que se hace es **reservar una zona de memoria para la variable**
 - ✓ Pero NO se borra el contenido de la variable
 - Puede haber valores previos (distintos de 0)

◆ Punteros a zonas erráticas

- ❖ Cuando definimos un puntero, el puntero puede tener un valor
 - ✓ Puede apuntar a una zona de memoria reservada para otra cosa, o
 - ✓ A una zona de memoria que está prohibida para la aplicación
 - Si accedemos en Linux, vemos error '**SEGMENTATION FAULT**' (o 'Violación de segmento')
 - ✓ Entonces, es mala práctica hacer

```
int *puntero;
*puntero = 3; /* escribo un 3... ¡no sé en dónde! */
```
- ❖ Precaución: Valor especial de un puntero: **NULL (ó 0)**
 - ✓ Indica que el puntero todavía no está apuntado a nada
 - Útil porque... Puedo preguntar al puntero si está apuntando válidamente a algún sitio o no

```
if (puntero == NULL) ...
```
 - ✓ Si se referencia (*), da error '**Segmentation fault**'
 - Al menos, está claro cuál es el problema
 - ✓ (pero cuando declaras un puntero... NO se inicializa como **NULL** – hay que hacerlo explícitamente, si se desea)

Punteros

- ◆ Hay que apuntar el puntero a una zona de memoria que esté asociada de forma única con el uso que queremos dar al puntero

- ◆ Dos formas

- ❖ Declaro variable (creo el espacio) y apunto puntero a variable

```
int a;  
int *ptrInt;  
ptrInt = &a;  
*ptrInt = 7; /* lo escribe en el espacio de la variable 'a' */
```

- ❖ Crear espacio de memoria dinámica, utilizando llamada al SO:

```
malloc (void *malloc(size_t size) );
```

- ✓ Ejemplo

```
char *ptrChar;  
ptrChar = (char *) malloc (10); /* crea espacio para guardar 10  
caracteres */
```

- ✓ La memoria dinámica hay que eliminarla antes de terminar el programa (no se elimina automáticamente)

```
free (ptrChar) ;
```

- ✓ La memoria dinámica permite definir **en tiempo de ejecución** espacio de memoria (~ variables) para ser usado por el programa

- ✓ (char *) es un **cast** – una conversión de tipos explicitada por el programador

- “compilador, hazme caso porque sé lo que estoy haciendo, supón que el void * que devuelve malloc en realidad es un char *”

- Si no hago el *cast*, el compilador genera un *warning*
- NO queremos *warnings*



Tamaños de variables

- ◆ A veces necesitamos conocer el tamaño de un tipo o una variable:
operador `sizeof()`

```
tamano = sizeof(int);
```

- ❖ ¡Puede variar según el equipo! (4, 8...)

```
tamano = sizeof (struct xxx);
```

- ❖ Devuelve el tamaño ocupado por una estructura `struct xxx` que habremos definido antes

```
char buffer_1[100];
```

```
tamano = sizeof(buffer_1);
```

- ❖ Devuelve 100

```
char * buffer_2;
```

```
buffer_2 = malloc(100);
```

```
tamano = sizeof (buffer_2);
```

- ❖ Devuelve...4 bytes, ¡el tamaño del puntero!
- ❖ `sizeof` no funciona aquí como esperábamos, porque es un operador que se aplica en *tiempo de compilación* (y el tamaño de `buffer_2` se define en *tiempo de ejecución*)
 - ✓ ¡Cuidado en su código con esto!



Copiando memoria

```
#include <string.h>
```

```
void *memcpy(void *dest, const void *orig, size_t n);
```

- ◆ Copia n bytes contando a partir de posición de memoria apuntada por `orig` a posición de memoria apuntada por `dest`
- ◆ Devuelve un puntero a `dest`
- ◆ El modificador `const` de `const void * orig` indica que la función NO cambia nada en las posiciones de memoria apuntadas por `orig` (el compilador asegura esto)

```
char *strcpy(char *dest, const char *orig);
```

- ◆ Espera un string: copia hasta que encuentra un `'\0'`
- ◆ ¡Bastante distinto a `memcpy`!



Estructuras

- ◆ Forma de definir variables que tienen diferentes partes (diferentes componentes)

```
struct servicio {int direccion, short int puerto};  
struct servicio web; /* variable de tipo struct servicio */  
  
web.direccion = 0xA3758B80; /* 163.177.139.128 en un  
entero de 32 bits, en formato hexadecimal */  
web.puerto = 80;
```

- ◆ Y con punteros

```
struct servicio *ptrServicio;  
ptrServicio = &web;  
if ( (ptrServicio ->puerto) == 80) {ptrServicio->puerto  
= 8080;} /*cambia a https */
```

- ◆ Nota (ptrServicio -> puerto) es una abreviatura C de ((*ptrServicio).puerto)
- ◆ Las funciones que manipulan estructuras declaran como parámetros punteros a la estructura

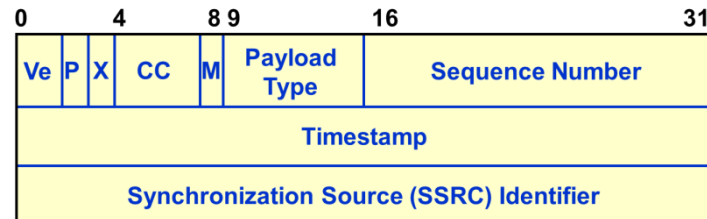


Estructuras

- ◆ C permite ‘tocar’ los bits

```
typedef struct {
    unsigned int version:2;    /* protocol version */
    unsigned int p:1;          /* padding flag */
    unsigned int x:1;          /* header extension flag */
    unsigned int cc:4;         /* CSRC count */
    unsigned int m:1;          /* marker bit */
    unsigned int pt:7;         /* payload type */
    unsigned int seq:16;       /* sequence number */
    u_int32 ts;                /* timestamp */
    u_int32 ssrc;              /* synchronization source */
} rtp_hdr_t;
```

```
rtp_hdr_t miCabecera; /* si no hubiera utilizado typedef, tendria
    que haber declarado esta variable como
    struct rtp_hdr_t miCabecera; */
miCabecera.version = 2;
miCabecera.pt = 8; /* payload type 8 */
```



Pasando parámetros por referencia

```
struct tiempo {  
    int segundos,  
    int miliseg };
```

```
main () { ...  
    struct tiempo tiempoAhora;  
    tomaTiempo (&tiempoAhora);  
}
```

```
void tomaTiempo (struct tiempo  
    *ptrTiempo)  
{  
    (*ptrTiempo).segundos =  
        /* accedo a hw...*/;  
    (*ptrTiempo).miliseg = /*...*/  
}
```

Direc	Contenido	Nombre variable
0x300	???	<i>main_tiempoAhora</i>
0x304	???	
0x308		



Pasando parámetros por referencia

```
struct tiempo {  
    int segundos,  
    int miliseg };  
  
main () { ...  
    struct tiempo tiempoAhora;  
    tomaTiempo (&tiempoAhora);  
}  
  
void tomaTiempo (struct tiempo  
    *ptrTiempo)  
{  
    (*ptrTiempo).segundos =  
        /* accedo a hw...*/;  
    (*ptrTiempo).miliseg = /*...*/  
}
```

Direc	Contenido	Nombre variable
0x300	???	<i>main_tiempoAhora</i>
0x304	???	
0x308	0x300	<i>tomaTiempo_ptrTiempo</i>



Pasando parámetros por referencia

```
struct tiempo {  
    int segundos,  
    int miliseg };
```

```
main () { ...  
    struct tiempo tiempoAhora;  
    tomaTiempo (&tiempoAhora);  
}
```

```
void tomaTiempo (struct tiempo  
    *ptrTiempo)  
{  
    (*ptrTiempo).segundos =  
        /* accedo a hw...*/;  
    (*ptrTiempo).miliseg = /*...*/  
}
```

Direc	Contenido	Nombre variable
<i>(*ptrTiempo).segundos</i>	2078	<i>main_tiempoAhora</i>
<i>(*ptrTiempo).miliseg</i>	???	
	0x300	<i>tomaTiempo_ptrTiempo</i>



Pasando parámetros por referencia

```
struct tiempo {  
    int segundos,  
    int miliseg };
```

```
main () { ...  
    struct tiempo tiempoAhora;  
    tomaTiempo (&tiempoAhora);  
}
```

```
void tomaTiempo (struct tiempo  
    *ptrTiempo)  
{  
    (*ptrTiempo).segundos =  
        /* accedo a hw...*/;  
    (*ptrTiempo).miliseg = /*...*/  
}
```

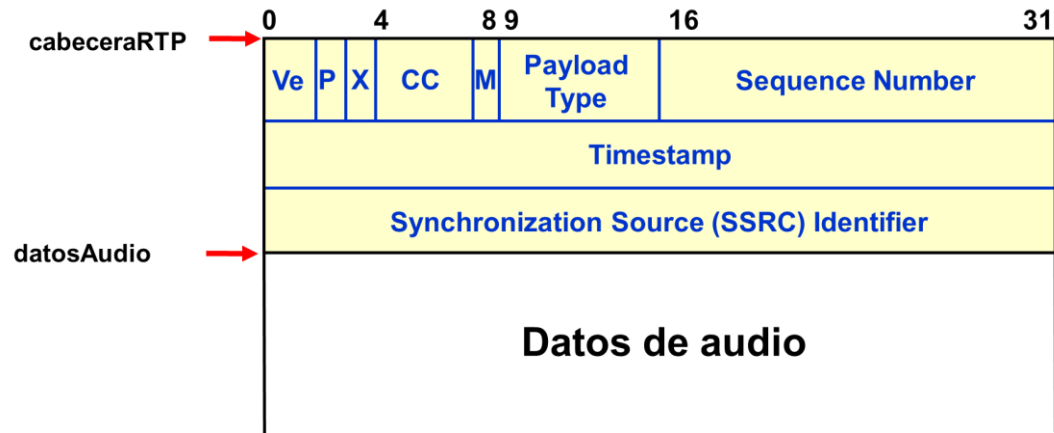
Direc	Contenido	Nombre variable
	2078 389	<i>main_tiempoAhora</i>
	0x300	<i>tomaTiempo_ ptrTiempo</i>



Ejemplo de aritmética de punteros

- ◆ Supongamos que queremos construir un paquete RTP para ser enviado: necesito una zona de memoria consecutiva en la que
 - ❖ Los 12 primeros bytes sean la cabecera RTP
 - ❖ Los 128 (por ejemplo) siguientes bytes sean datos de audio

```
void * paquete;  
rtp_hdr_t *cabeceraRTP;  
char * datosAudio;
```



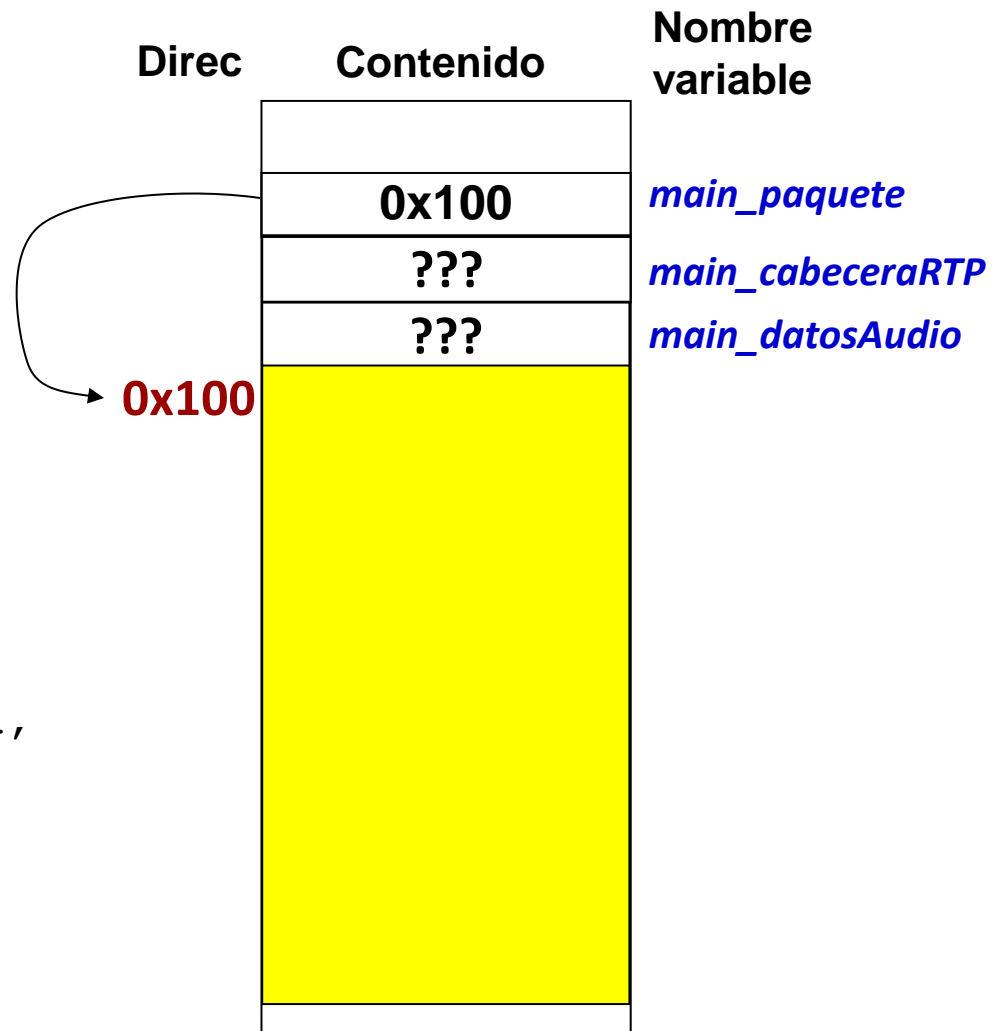
```
paquete = malloc (140);  
cabeceraRTP = (rtp_hdr_t *) paquete;  
(*cabeceraRTP).version = 2;  
/* relleno cabecera... */  
datosAudio = ((char *) paquete) + sizeof (rtp_hdr_t);  
/* copio datos o llamo a read(..., datosAudio,...)
```

NO
`sizeof(cabeceraRTP)`



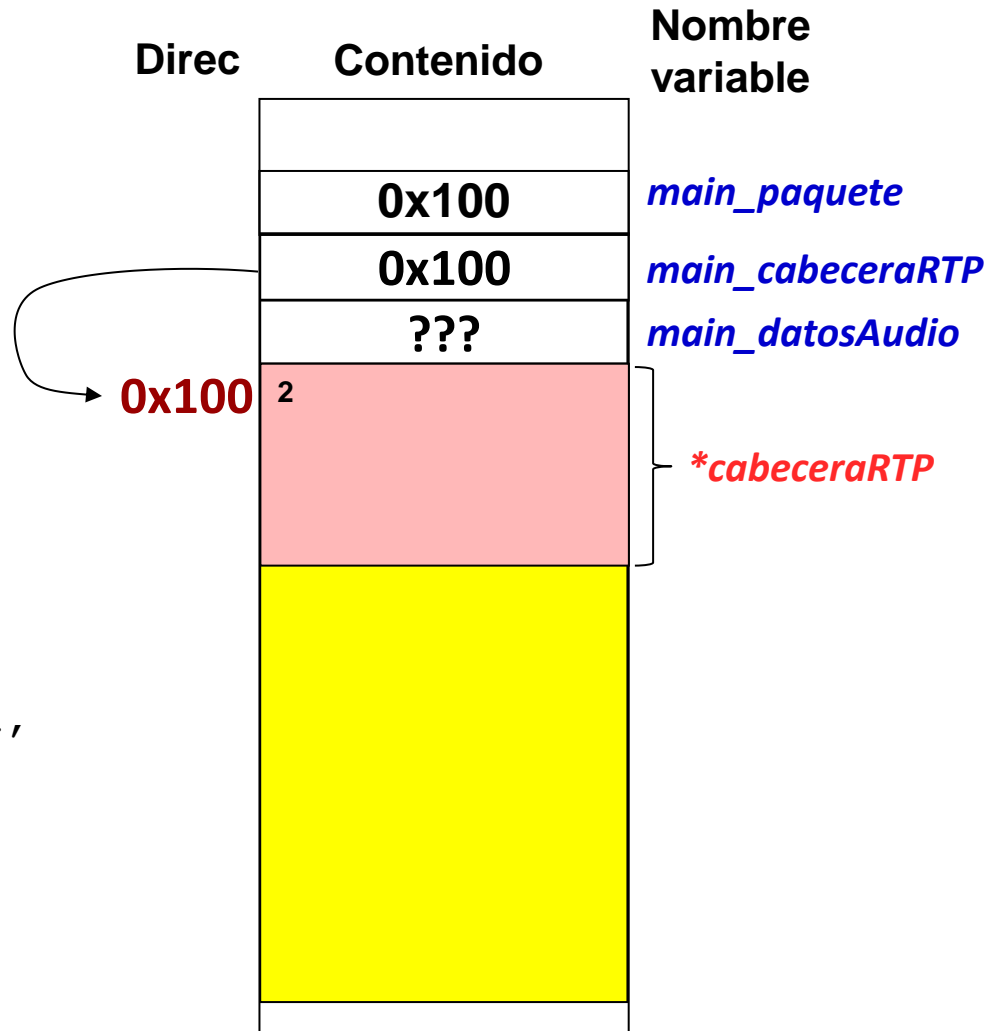
Ejemplo de aritmética de punteros

```
void * paquete;  
rtp_hdr_t *cabeceraRTP;  
char * datosAudio;  
  
paquete = malloc (140);  
cabeceraRTP = (rtp_hdr_t *)  
paquete;  
(*cabeceraRTP).version = 2;  
/* relleno cabecera... */  
datosAudio = ((char *) paquete)  
+ sizeof (rtp_hdr_t);  
/* copio datos o llamo a read(...,  
datosAudio,...)
```



Ejemplo de aritmética de punteros

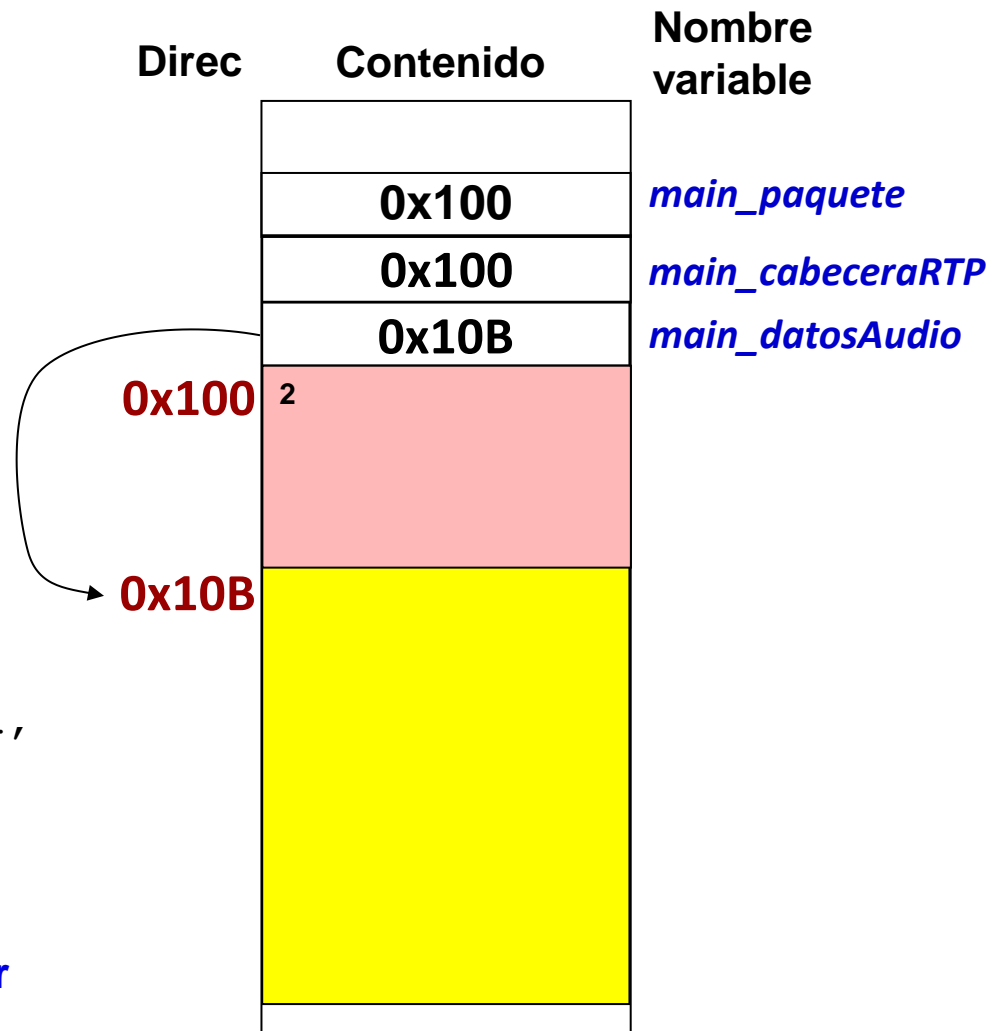
```
void * paquete;  
rtp_hdr_t *cabeceraRTP;  
char * datosAudio;  
  
paquete = malloc (140);  
cabeceraRTP = (rtp_hdr_t *)  
paquete;  
(*cabeceraRTP).version = 2;  
/* relleno cabecera... */  
datosAudio = ((char *) paquete)  
+ sizeof (rtp_hdr_t);  
/* copio datos o llamo a read(...,  
datosAudio,...)
```



Ejemplo de aritmética de punteros

```
void * paquete;  
rtp_hdr_t *cabeceraRTP;  
char * datosAudio;  
  
paquete = malloc (140);  
cabeceraRTP = (rtp_hdr_t *)  
paquete;  
(*cabeceraRTP).version = 2;  
/* relleno cabecera... */  
datosAudio = ((char *) paquete)  
+ sizeof (rtp_hdr_t);  
/* copio datos o llamo a read(...,  
datosAudio,...)
```

*char * es como decir
'puntero a byte'*



printf

- ◆ Permite combinar 'texto fijo' con variables
- ◆ Indicación de sustituir por una variable
 - ❖ `%d` - variable de tipo entero, imprimir en formato decimal
 - ❖ `%s` - variable de tipo string (`char *`, terminado por un caracter `'\0'`)

✓ Más formatos disponibles (ver man 3 printf)

```
printf ("Volumen de reproducción: %d.  
Numero de bits por muestra: %d", vol,  
bits);
```

- ◆ Numero variable de argumentos, pero
 - ❖ Tiene que haber tantas `%_` como variables
 - ❖ El tipo indicado en `%_` tiene que corresponderse con el de la variable 'que le toca'
 - ❖ Si hay variables `'char *'`, tienen que ser cadenas de caracteres, es decir, terminar en `'\0'`



Generando un ejecutable

◆ Utilizaremos la herramienta `gcc` (GNU C Compiler)

❖ En realidad, son varias herramientas:

- ✓ **Compilador:** coge 1 fichero `.c` y lo convierte en 1 fichero `.o` que está en formato 'código objeto'
 - También ejecuta un preprocesador: sustituye `#include...`
- ✓ **Enlazador:** coge uno o varios ficheros `.o` y los junta en 1 ejecutable (posiblemente, juntando código de otras funciones del sistema – `printf`, `exit`, `open...`)
 - El ejecutable no suele llevar extensión
 - Por motivos de seguridad, el ejecutable lo deberá invocar como `./nombreEjecutable`

◆ Al compilar, `gcc` tiene que entender todos los términos que aparezcan en el programa

- ❖ O son variables definidas, o funciones definidas dentro o en un `#include...`

◆ Al enlazar, `gcc` tiene que encontrar el código (`.o` o `.so`) de todas las funciones utilizadas

- ❖ Parte de este código (el de `printf`, `exit`, `open...`) está en 'librerías'
- ❖ P.ej, `printf` está en `glibc` (`/usr/lib.../libc-2.13.so`)
- ❖ Si está en `glibc`, no tengo que decir nada al compilador
 - ✓ Si está en otra librería, tengo que decirle el nombre de la librería
 - ✓ Ejemplo: `log` está en `libm`. Al compilador hay que decirle `-lm` (= "enlaza con `libm`")



Escribiendo código

◆ Puede escribir su código en varios ficheros

- ❖ Facilita que varias personas trabajen en el mismo programa
 - ✓ En su caso, yo doy varios ficheros hechos!
- ❖ Facilita reutilizar código
 - ✓ Se puede utilizar fácilmente algunas funciones en otros programas
- ❖ Permite estructurar mejor el código

◆ Cuando se utilizan varios ficheros

- ❖ En *.h se ponen la declaración de las funciones
 - ✓ Se pueden definir tipos, constantes... siempre que sean importantes para otros ficheros que vayan a utilizar estas funciones
- ❖ En *.c se pone la cabecera de la función (otra vez)... y el código que lo hace (de verdad)
- ❖ Si un fichero uno.c utiliza funciones de otro dos.c, uno.c tiene que tener un #include dos.h
- ❖ Puede compilar cada .c por separado

```
gcc -c fichero.c      -- genera un fichero.o
```
- ❖ Sólo puede haber 1 main entre todos los ficheros
- ❖ Cuando están todos los .o disponibles, se pueden enlazar en un ejecutable

```
gcc -o ejecutable fich1.o fich2.o fich3.o ...
```

Compilando

- ◆ **Usuarios avanzados utilizan make**
 - ❖ Pero NO merece la pena 'pegarse' con él

Alternativa

- ◆ **Cree un fichero compila que contiene una línea**
`gcc -o audioSimple audioSimple.c audioSimpleArgs.c
configureSndcard.c ... -Wall -lm`
 - ❖ Todos los .c que necesita
 - ❖ -Wall: chequeo más estricto
 - ❖ -lm: enlazar con librería matemática
 - ✓ En audioSimple no hace falta, pero puede hacerle falta; no hace daño
- ◆ **Ejecute**
`# chmod 755 compila`
- ◆ **Cada vez que quiera compilar,**
`# ./compila`

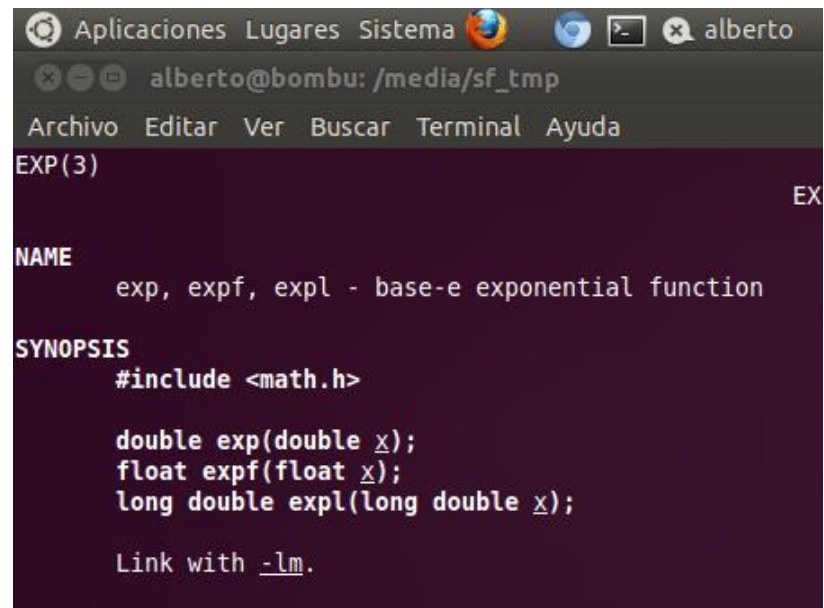


Si quiero utilizar una función 'estándar'...

- ◆ ¿cómo sé qué argumentos toma y que valores devuelve?
- ◆ ¿cómo sé qué fichero tengo que incluir?
- ◆ Ejecutar 'man funcion'

`man exp`

- ◆ También se detalla cómo se usa la función



```
alberto@bomбу: /media/sf_tmp
Archivo  Editar  Ver  Buscar  Terminal  Ayuda
EXP(3)

NAME
    exp, expf, expl - base-e exponential function

SYNOPSIS
    #include <math.h>

    double exp(double x);
    float expf(float x);
    long double expl(long double x);

    Link with -lm.
```

Sobre #include

- ◆ Si se quiere incluir una cabecera estándar, utilizar

```
#include <cabecera.h>
```

- ❖ El compilador sabe en qué directorios buscar estas cabeceras

- ◆ Si se quiere incluir una cabecera desarrollada por nosotros

```
#include "cabecera.h"
```

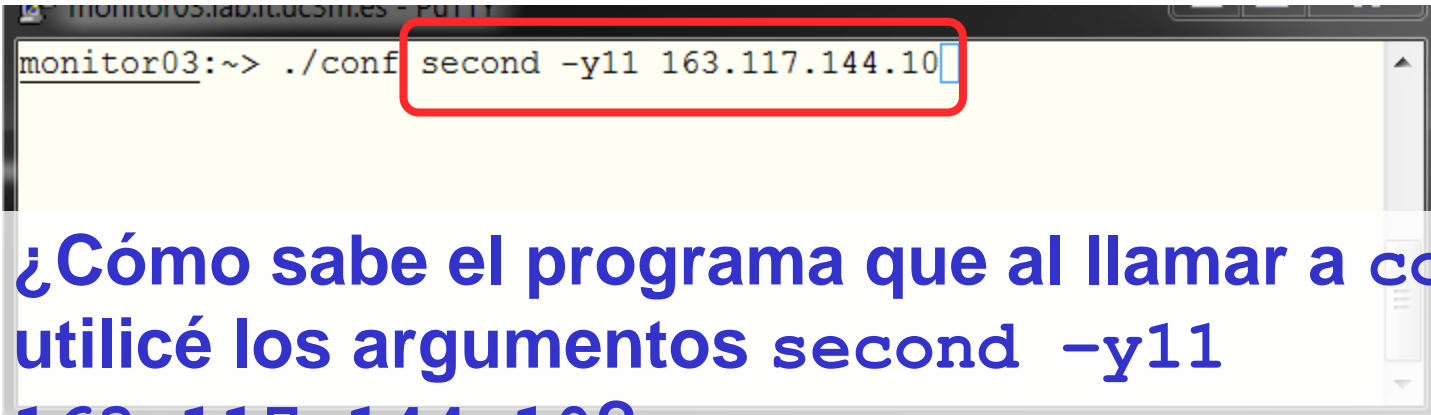
- ✓ Si está en el mismo directorio que el fichero que lo incluye

```
#include "/home/.../cabecera.h"
```

- ✓ Si está en otro directorio que el fichero que lo incluye



Capturando argumentos de la línea de comandos

A terminal window with a yellow background. The prompt is 'monitor03:~>'. The command being entered is './conf second -y11 163.117.144.10'. The command is highlighted with a red rectangular box.

```
monitor03:~> ./conf second -y11 163.117.144.10
```

- ◆ ¿Cómo sabe el programa que al llamar a `conf` utilicé los argumentos `second -y11 163.117.144.10`?
- ◆ C dice que en la función `main` se define como

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

 - ❖ ‘mágicamente’, cuando el código se ejecute, en la variable `argc` y `argv` está la información para recuperar los argumentos
 - ✓ No damos valor a estas variables, sino que el valor sale de ‘la ejecución’

Capturando argumentos de la línea de comandos

◆ ESTO NO LO TENEIS QUE HACER VOSOTROS

- ❖ Ya lo damos hecho (pero...)

◆ La función main está declarada como

```
int main (int argc, char *argv[]) { ... }
```



- ❖ Tipo de argv se 'lee' aplicando la regla de 'apretar el tornillo'

- ✓ Declaraciones complejas se pueden analizar con herramienta cdecl (o en web <http://www.lemode.net/c/cdecl/>)

```
$ cdecl explain char *argv[]  
declare argv as array of pointer to char
```

- ❖ aunque main también se puede utilizar como

```
int main (void)
```

- ✓ C es muy estricto con las declaraciones... ¡salvo con las de main!

◆ El intérprete de la línea de comandos deja en

- ❖ argc el número de argumentos del comando
- ❖ argv un puntero a un array de punteros a strings
- ❖ argv[0] es el nombre de programa, con que argc es siempre al menos 1.

Capturando argumentos de la línea de comandos

```
./audiosimple play -v100 music
```

```
argc = 4,  
argv = <direcc 0>
```

argv:
[0] <dir 1>
[1] <dir 2>
[2] <dir 3>
[3] <dir 4>
[4] NULL

'a' 'u' 'd' 'i' 'o' 's' 'i' 'm' 'p' 'l' 'e' '\0'

'p' 'l' 'a' 'y' '\0'

'-' 'v' '1' '0' '0' '\0'

'm' 'u' 's' 'i' 'c' '\0'



◆ Ejercicio: compilar práctica 0

- ❖ Analizar el código de los ficheros `audioSimpleArgs.h` y `audioSimpleArgs.c`