



INTRODUCCIÓN AL LENGUAJE C

ARCOS . INF . UC3M . ES

FÉLIX GARCÍA CARBALLEIRA,
ALEJANDRO CALDERÓN MATEOS

ADVERTENCIA

- Este material es un simple guión de la clase: no son los apuntes de la asignatura.
- El conocimiento exclusivo de este material no garantiza que el/la estudiante pueda alcanzar los objetivos de la asignatura.
- Se recomienda que el/la estudiante utilice los materiales complementarios propuestos.

Bibliografía

- **Problemas resueltos de programación en C**
F. García, J. Carretero, A. Calderón, J. Fernández, J. M. Pérez.
Thomson, 2003 (ISBN: 84-9732-102-2)
- **El lenguaje de programación C. Diseño e implementación de programas**

J. Carretero, F. García, J. Fernández, A. Calderón
Prentice Hall, 2001

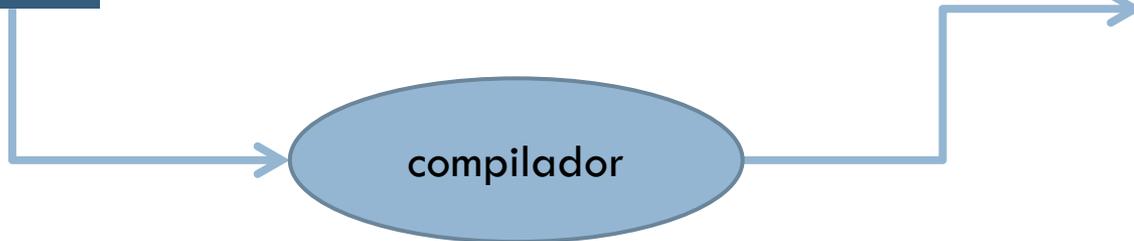
Contenidos

- **Introducción al lenguaje C**
 - **Preprocesador**
 - **Comentarios**
 - **Tipos de datos básicos, variables y constantes**
 - **Asignación y conversión de tipos (casting)**
 - **Definición de tipos**
 - **Tipos compuestos: array y struct**
 - **Sentencias de control**
 - **Funciones**
 - **Punteros**

Hola mundo

```
/* Inclusión de archivos */  
#include <stdio.h>  
  
/* Función principal */  
int main (int argc, char **argv)  
{  
    /* Impresión por pantalla y salida del programa*/  
    printf("Hola mundo\n");  
    return 0;  
}
```

hola.c



Hola mundo

```
/* Inclusión de archivos */  
#include <stdio.h>  
  
/* Función principal */  
int main (int argc, char **argv)  
{  
    /* Impresión por pantalla y salida del programa*/  
    printf("Hola mundo\n");  
    return 0;  
}
```

hola.c



```
gcc hola.c -Wall -g -o hola
```



./hola

Hola mundo

7



Félix García Carballera,
Alejandro Calderón Mateos

```
/* Inclusión de archivos */  
#include <stdio.h>  
  
/* Función principal */  
int main (int argc, char **argv)  
{  
    /* Impresión por pantalla y salida del programa*/  
    printf("Hola mundo\n");  
    return 0;  
}
```

hola.c



./hola

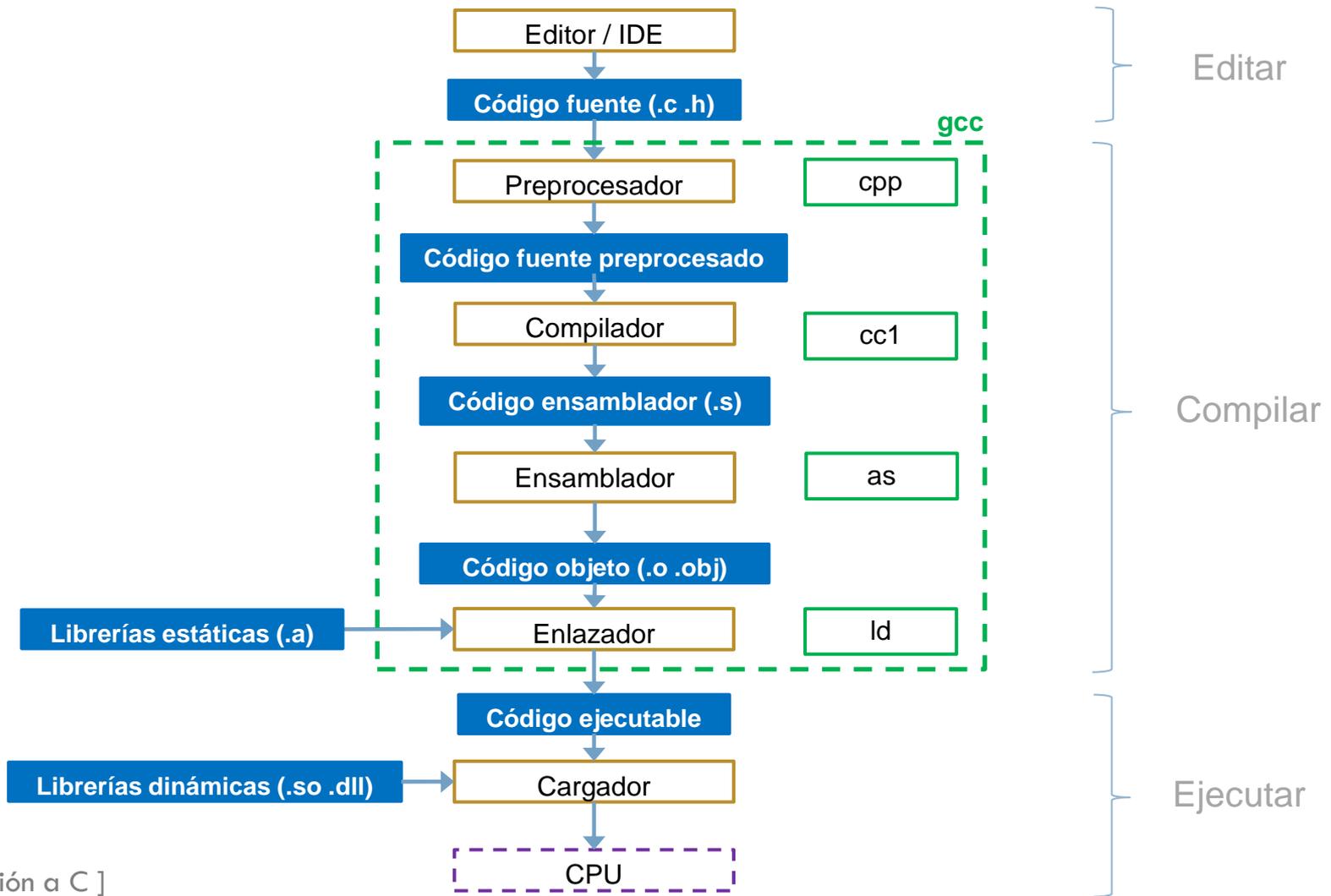
Modelo de compilación de C

¿Qué pasa al ejecutar `gcc -g hola.c -o hola`?

8



Félix García Carballera,
Alejandro Calderón Mateos



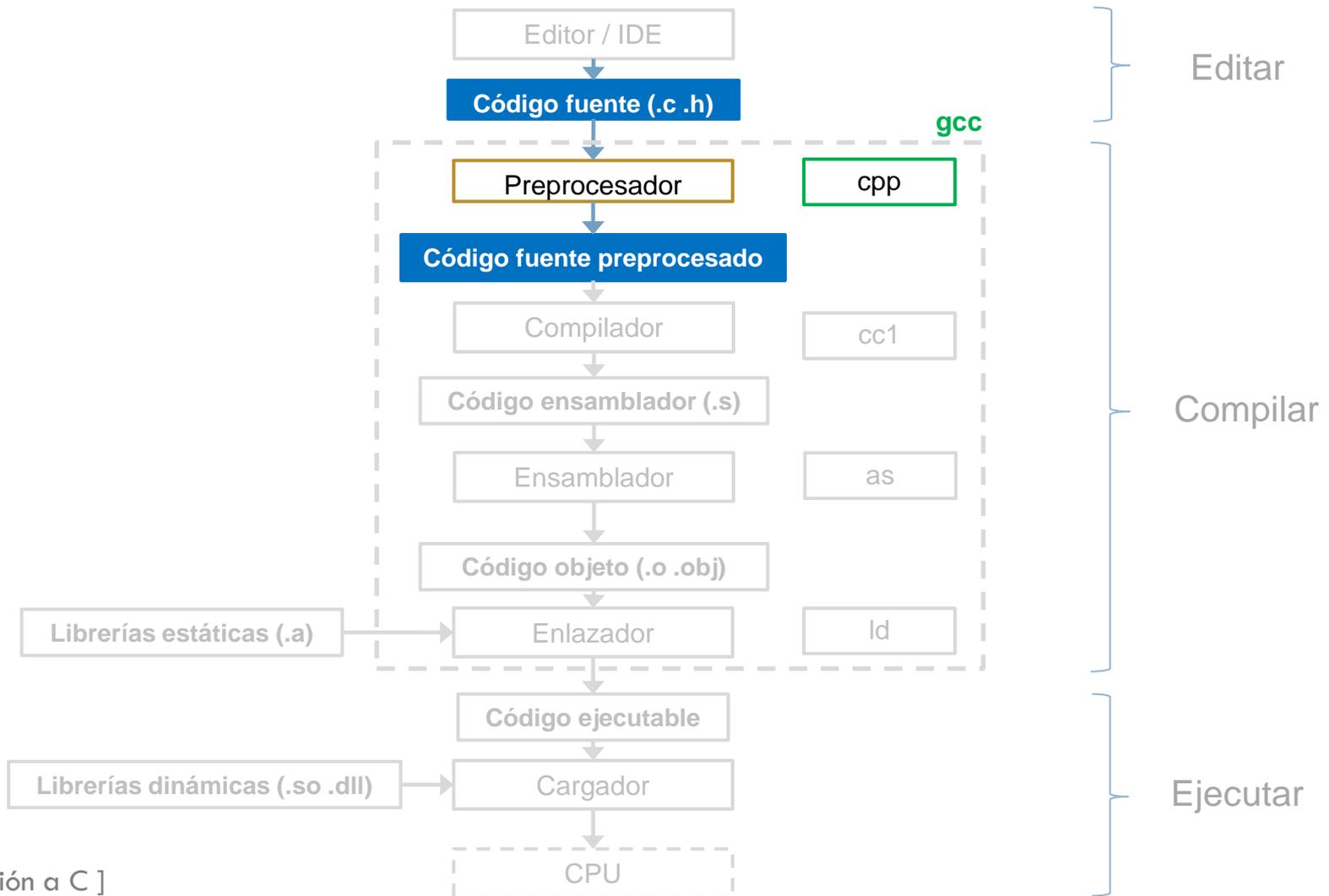
Modelo de compilación de C

¿Qué pasa al ejecutar `gcc -g hola.c -o hola`?

9



Félix García Carballera,
Alejandro Calderón Mateos



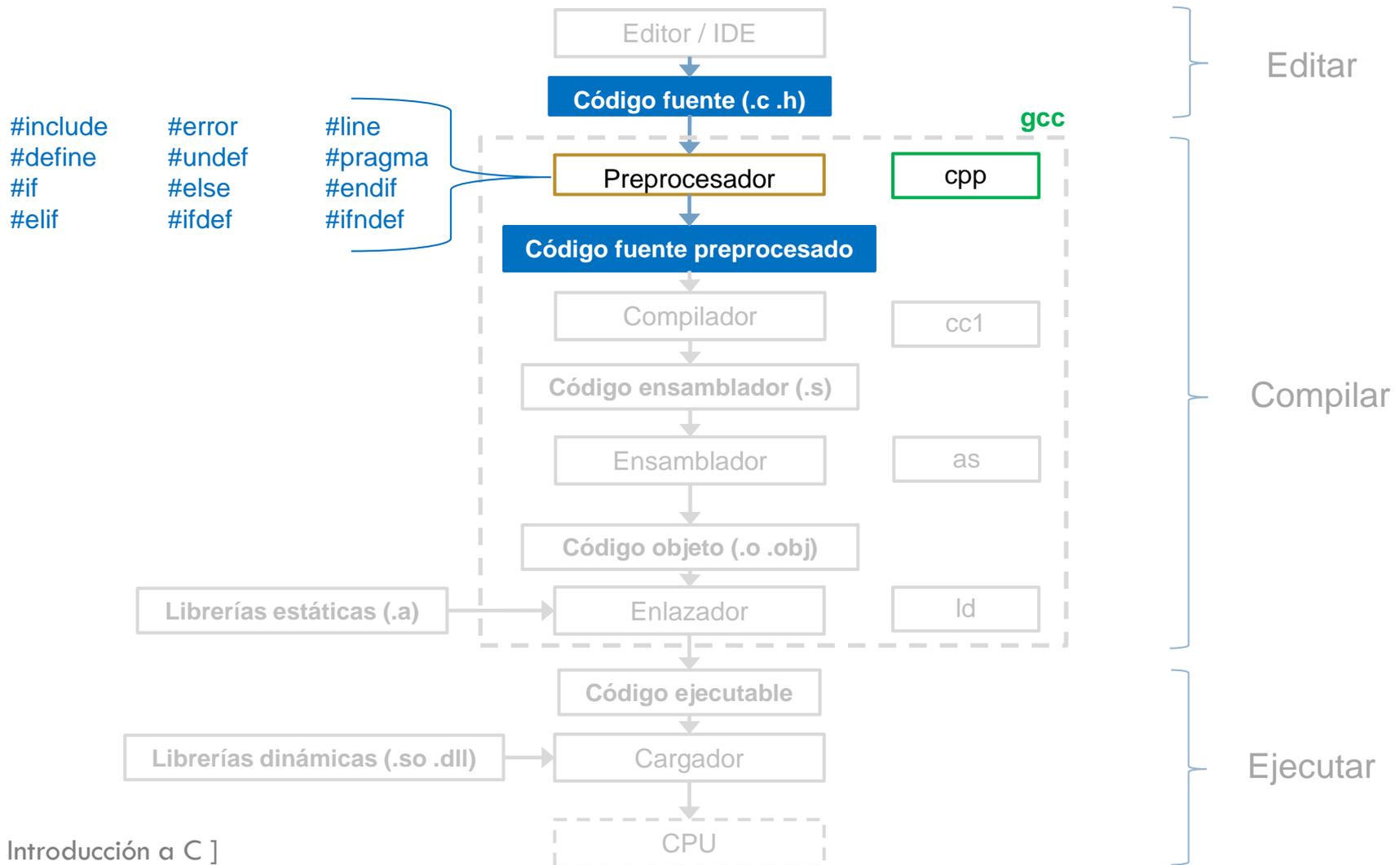
Modelo de compilación de C

¿Qué pasa al ejecutar `gcc -g hola.c -o hola`?

10



Félix García Carballera,
Alejandro Calderón Mateos



(1 / 5) #include (inclusión)



Félix García Carballera,
Alejandro Calderón Mateos

.c / .h

Preprocesador

.i

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

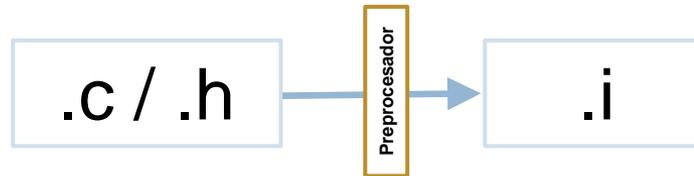
```
struct __sFile
{
    int unused;
};

typedef struct __sFILE FILE;
...
```

(2/5) #define (constantes)



Félix García Carballeira,
Alejandro Calderón Mateos



```
#define PI_PLUS_ONE (3.14 + 1)
#define PI_PLUS_TWO 3.14 + 2

x = PI_PLUS_ONE * 5;
x = PI_PLUS_TWO * 5;
```

```
x = (3.14 + 1) * 5;
x = 3.14 + 2 * 5;
```

(3/5) #if/#endif (guardas)



Félix García Carballera,
Alejandro Calderón Mateos

.c / .h

Preprocesador

.i

```
#ifndef NULL
#define NULL (void *)0
#endif
```

```
#define NULL (void *)0
```

(4/5) #define (macros)



.c / .h

Preprocesador

.i

```
#define ADD(x,y) (x+y)
```

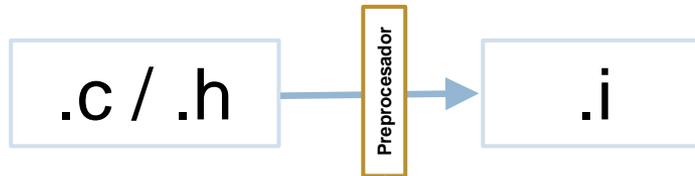
```
A = ADD(3, 4) ;
```

```
A = (3 + 4) ;
```

(5/5) #error (error preprocesando)



Félix García Carballera,
Alejandro Calderón Mateos



```
#ifndef NULL
#error "Undefined NULL value"
#endif
```

<Si NULL no está definido el
compilador muestra error>

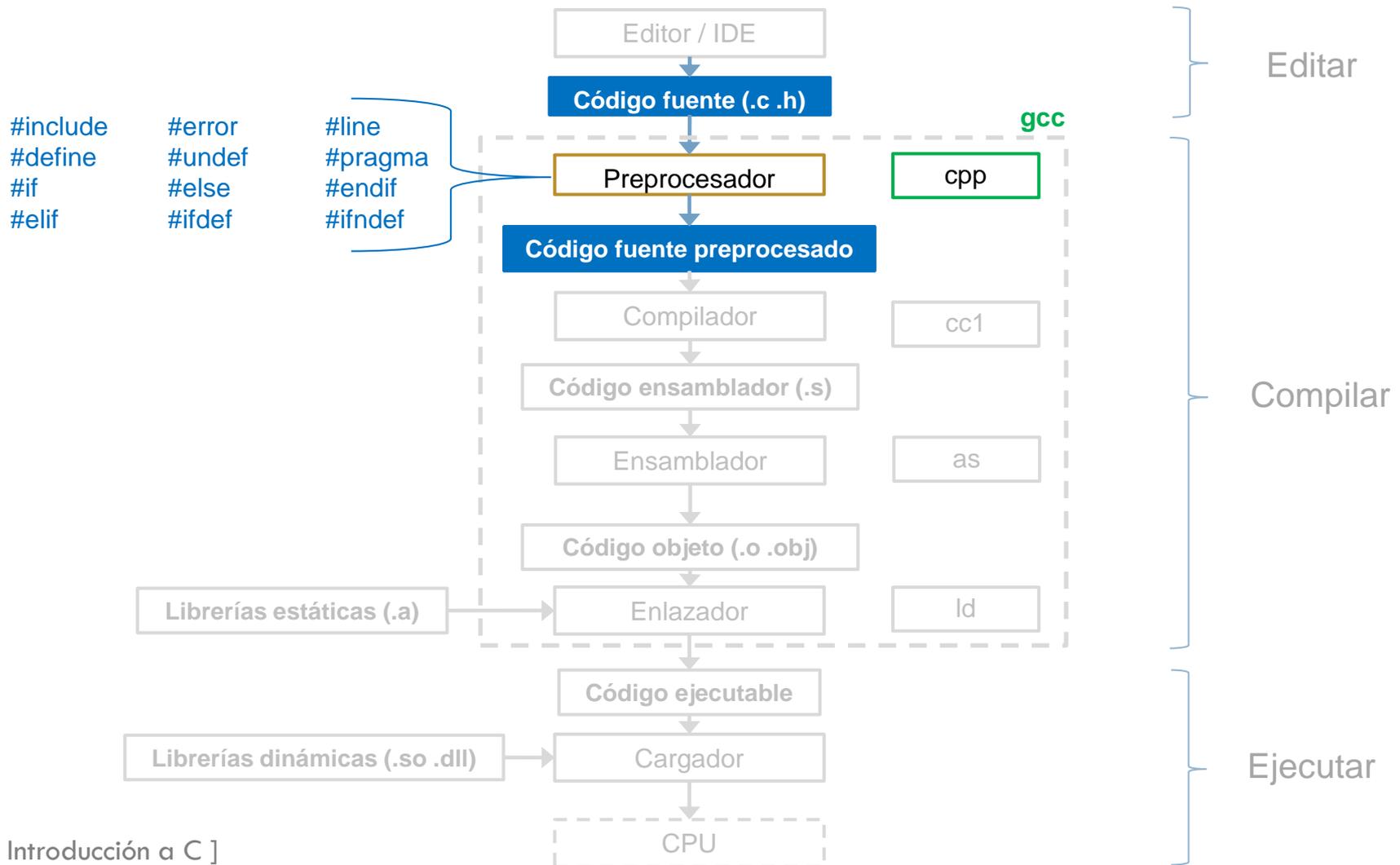
Modelo de compilación de C

¿Qué pasa al ejecutar `gcc -g hola.c -o hola`?

16



Félix García Carballera,
Alejandro Calderón Mateos



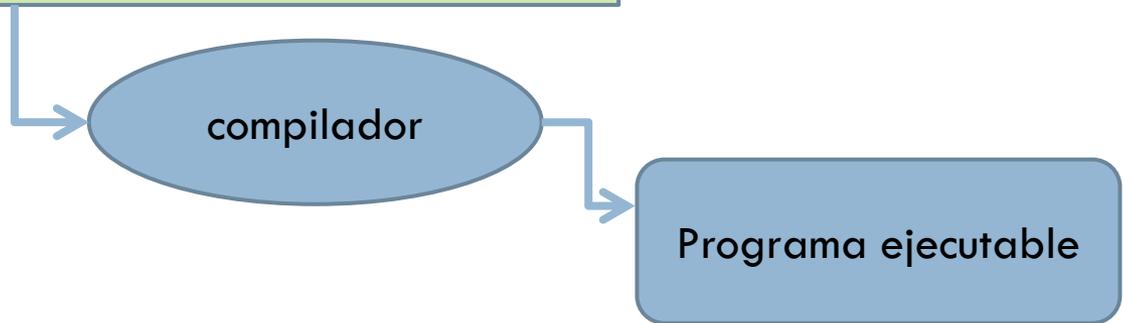
Contenidos

- **Introducción al lenguaje C**
 - **Preprocesador**
 - **Comentarios**
 - **Tipos de datos básicos, variables y constantes**
 - **Asignación y conversión de tipos (casting)**
 - **Tipos compuestos: array y struct**
 - **Definición de tipos**
 - **Sentencias de control**
 - **Funciones**
 - **Punteros**

Hola mundo

hola.c

```
/* Inclusión de archivos */  
#include <stdio.h>  
  
/* Función principal */  
int main (int argc, char **argv)  
{  
    /* Impresión por pantalla y salida del programa*/  
    printf("Hola mundo\n");  
    return 0;  
}
```



Comentarios

- **Comentario de línea**
 - `//` ignora hasta el final de línea
 - Los comentarios del tipo `//` no son válidos en ANSI C.

- **Comentario de bloque**
 - Cualquier secuencia entre `/*` y `*/`.
 - No se pueden anidar comentarios: `/* /* ... */ */`

Comentarios

- Comentar ayuda a entender el código:
 - Ejemplo: `int funcion1 (char *parámetro) ;`
 - ¿Es un char por referencia, una cadena, ...?

- Comentar permite añadir un poco de arte:
 - Generador de “ascii-art”

Comentarios

- Comentar ayuda a entender el código:
 - Ejemplo: `int funcion1 (char *parámetro) ;`
 - ¿Es un char por referencia, una cadena, ...?

- Comentar permite añadir un poco de arte:
 - Generador de “ascii-art”

Doxygen

Generate documentation from source code

Doxygen is the de facto standard tool for generating documentation from annotated C++ sources, but it also supports other popular programming languages such as C, Objective-C, C#, PHP, Java, Python, IDL (Corba, Microsoft, and UNO/OpenOffice flavors), Fortran, VHDL and to some extent D.

Doxygen can help you in three ways:

1. It can generate an on-line documentation browser (in HTML) and/or an off-line reference manual (in \LaTeX) from a set of documented source files. There is also support for generating output in RTF (MS-Word), PostScript, hyperlinked PDF, compressed HTML, and Unix man pages. The documentation is extracted directly from the sources, which makes it much easier to keep the documentation consistent with the source code.
2. You can [configure](#) doxygen to extract the code structure from undocumented source files. This is very useful to quickly find your way in large source distributions. Doxygen can also visualize the relations between the various elements by means of include dependency graphs, inheritance diagrams, and collaboration diagrams, which are all generated automatically.
3. You can also use doxygen for creating normal documentation (as I did for the doxygen user manual and web-site).

Doxygen is developed under Mac OS X and Linux, but is set-up to be highly portable. As a result, it runs on most other Unix flavors as well. Furthermore, executables for Windows are available.

Adobe Creative Cloud for Teams
starting at \$33.99 per month.
ADS VIA CARBON

Ejemplo de comentario para doxygen

```
/**  
 * a normal member taking two arguments and returning an integer value.  
 * @param a an integer argument.  
 * @param s a constant character pointer.  
 * @see testMeToo()  
 * @see publicVar()  
 * @return The test results  
 */  
int testMe ( int a, const char *s ) ;
```

Ejemplo de uso de doxygen

1. `cd <directorio raíz del proyecto>`
2. `doxygen -g <config-file>`
3. `gedit <config-file>`
 - ▣ `PROJECT_NAME='proyecto'`
 - ▣ `INPUT=./src ./include`
 - ▣ `HTML_OUTPUT=html/`
 - ▣ `GENERATE_HTML=YES`
4. `doxygen <config-file>`

Comentarios

- Comentar ayuda a entender el código:
 - Ejemplo: `int funcion1 (char *parámetro) ;`
 - ¿Es un char por referencia, una cadena, ...?

- Comentar permite añadir un poco de arte:
 - Generador de “ascii-art”

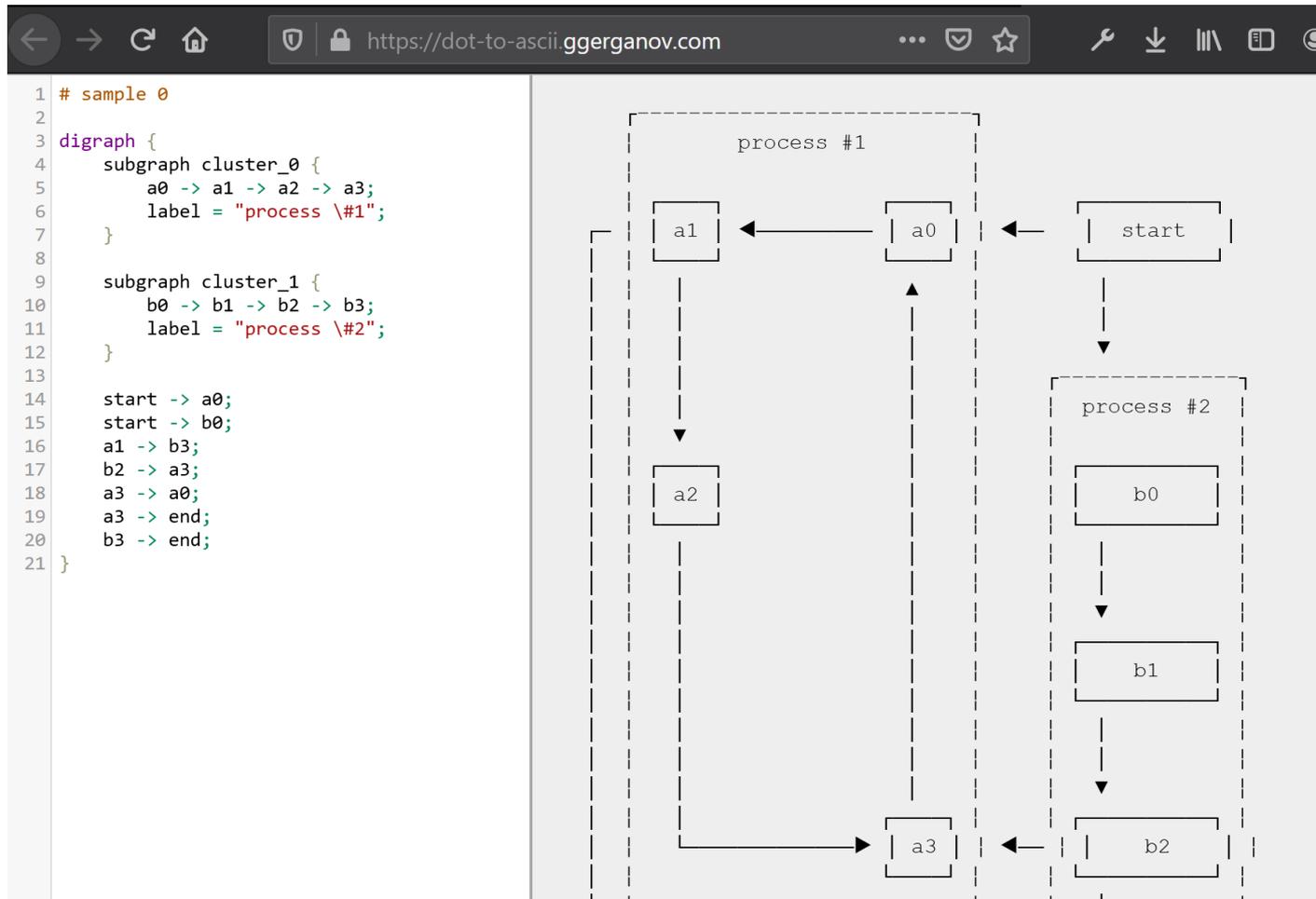
ASCII-ART

<https://dot-to-ascii.ggerganov.com/>

26



Félix García Carballera,
Alejandro Calderón Mateos



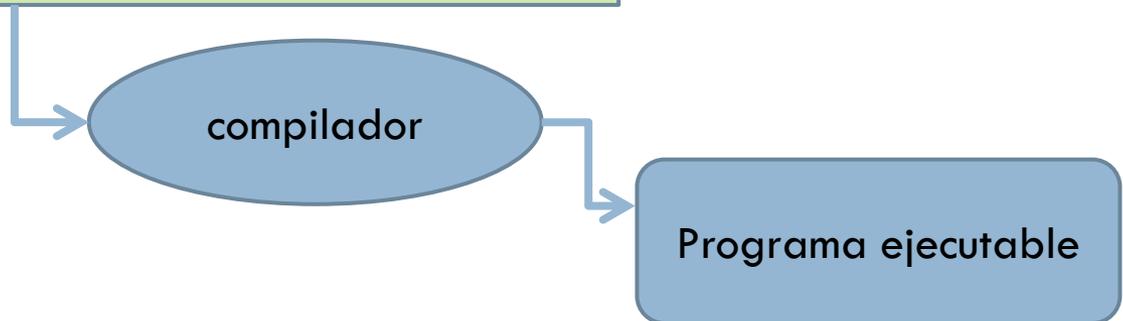
Contenidos

- **Introducción al lenguaje C**
 - **Preprocesador**
 - **Comentarios**
 - **Bibliotecas**
 - **Tipos de datos básicos, variables y constantes**
 - **Asignación y conversión de tipos (casting)**
 - **Tipos compuestos: array y struct**
 - **Definición de tipos**
 - **Sentencias de control**
 - **Funciones**
 - **Punteros**

Hola mundo

hola.c

```
/* Inclusión de archivos */  
#include <stdio.h>  
  
/* Función principal */  
int main ( int argc, char **argv )  
{  
    /* Impresión por pantalla y salida del programa*/  
    printf("Hola mundo\n");  
    return 0;  
}
```



Modelo de compilación de C

¿Qué pasa al ejecutar `gcc -g hola.c -o hola`?

29

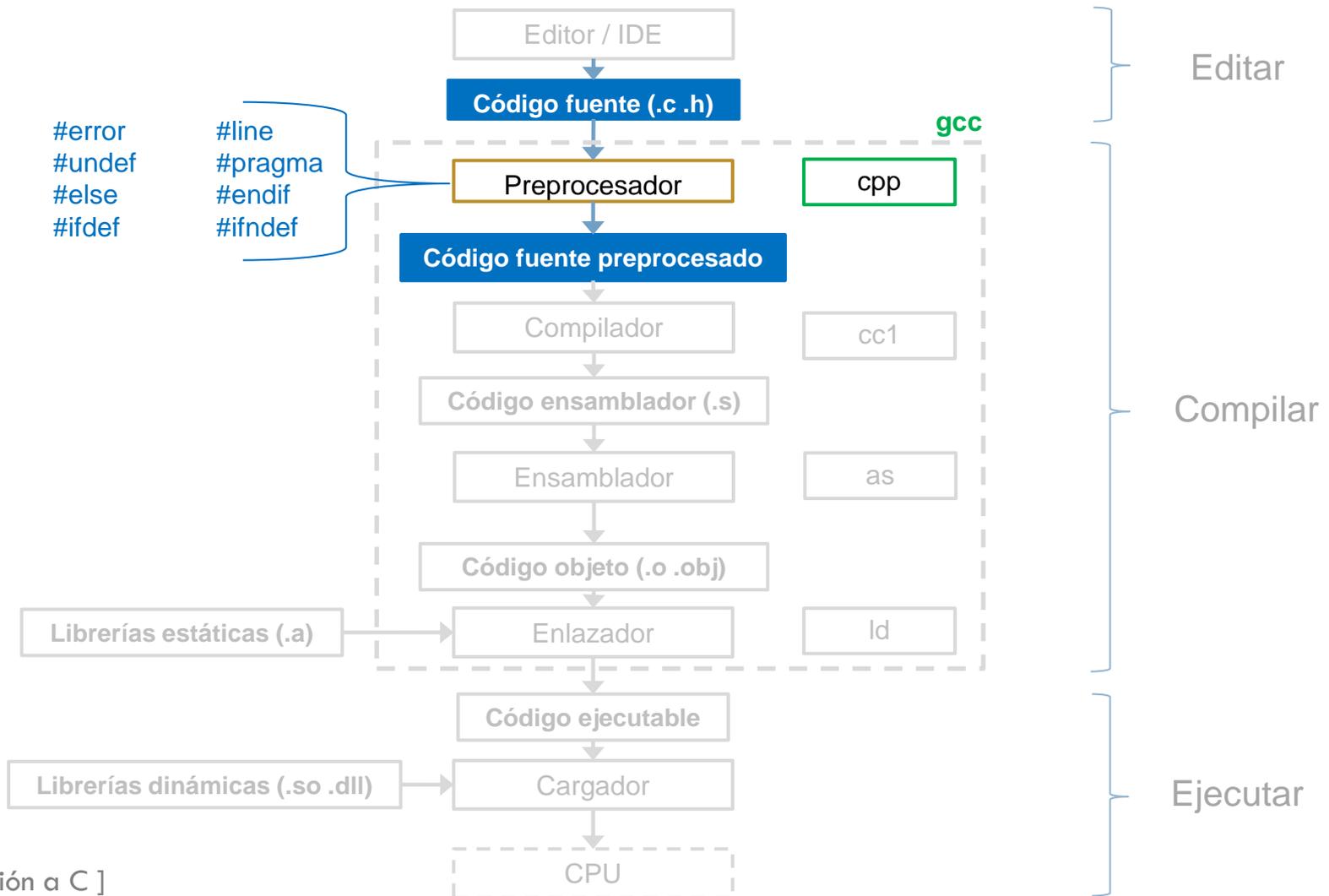


Félix García Carballera,
Alejandro Calderón Mateos

`#include`
`#define`
`#if`
`#elif`

`#error`
`#undef`
`#else`
`#ifdef`

`#line`
`#pragma`
`#endif`
`#ifndef`



Utilización de bibliotecas

- En **C** La directiva `#include` de preprocesador copia el contenido de un archivo un el punto de inclusión.
 - `#include <archivo.h>` → Inclusión de biblioteca del sistema.
 - `#include "archivo.h"` → Inclusión de biblioteca del usuario.

Ejemplo de biblioteca de usuario

declaraciones

mi.h

```
extern int g1 ;  
int f1( int p1, char p2 ) ;
```

definiciones

mi.c

```
int g1 = 10 ;  
int f1( int p1, char p2 )  
{  
    return p1+(int)p2 ;  
}
```

main.c

```
#include "mi.h"  
#include <stdio.h>  
  
int main ( int argc,  
           char *argv[] )  
{  
    int r ;  
  
    r=f1(5,'0') ;  
    printf("r=%d\n",r) ;  
    return 0 ;  
}
```

Ejemplo de biblioteca de usuario

declaraciones

mi.h

```
extern int g1 ;  
int f1( int p1, char p2 );
```

definiciones

mi.c

```
int g1 = 10 ;  
int f1( int p1, char p2 )  
{  
    return p1+(int)p2 ;  
}
```

main.c

```
extern int g1 ;  
int f1( int p1, char p2 );  
#include <stdio.h>  
  
int main ( int argc,  
           char *argv[] )  
{  
    int r ;  
  
    r=f1(5,'0') ;  
    printf("r=%d\n",r) ;  
    return 0 ;  
}
```

Utilización de bibliotecas

- En C La directiva `#include` de preprocesador copia el contenido de un archivo un el punto de inclusión.
 - `#include <archivo.h>` → Inclusión de biblioteca del sistema.
 - `#include "archivo.h"` → Inclusión de biblioteca del usuario.

- En **C** **NO** existe nativamente el concepto en Python de *module* ni hay clausula *import*.
 - ▣ Hay que incluir en el programa la declaración de los tipos de datos y funciones que se usan.
 - ▣ Las bibliotecas estándar se enlazan por defecto. Las bibliotecas propias se le indican al compilador.

Ejemplo de biblioteca de usuario

declaraciones

```
mi.h
#ifndef _MI_H_
#define _MI_H_
extern int g1 ;
int f1( int p1, char p2 );
#endif
```

definiciones

```
mi.c
#include "mi.h"

int g1 = 10 ;
int f1( int p1, char p2 )
{
    return p1+(int)p2 ;
}
```

main.c

```
#include "mi.h"
#include <stdio.h>

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

    r=f1(5,'0') ;
    printf("r=%d\n",r) ;
    return 0 ;
}
```

Ejemplo de biblioteca de usuario

declaraciones

```
mi.h
#ifndef _MI_H_
#define _MI_H_
extern int g1 ;
int f1( int p1, char p2 );
#endif
```

definiciones

```
mi.c
#include "mi.h"

int g1 = 10 ;
int f1( int p1, char p2 )
{
    return p1+(int)p2 ;
}
```

```
gcc -Wall -g -c mi.c -o mi.o
```

1

```
main.c
#include "mi.h"
#include <stdio.h>

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

    r=f1(5,'0') ;
    printf("r=%d\n",r) ;
    return 0 ;
}
```

```
gcc -Wall -g -c main.c -o main.o
```

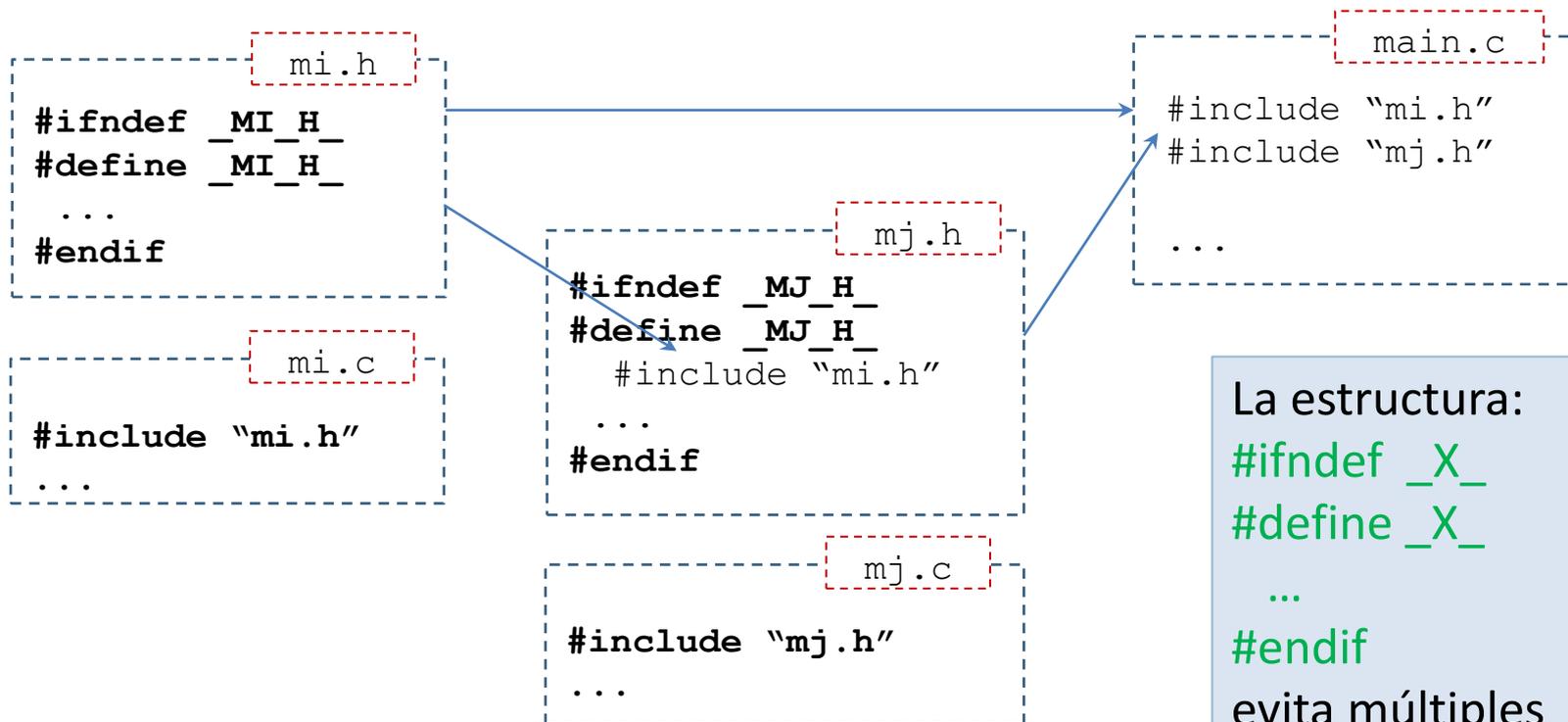
2

```
gcc -o main main.o mi.o
```

3

dependencias

Ejemplo de guardas



La estructura:

```
#ifndef _X_
```

```
#define _X_
```

```
...
```

```
#endif
```

evita múltiples
inclusiones
no necesarias

Ejemplo de biblioteca de usuario

declaraciones

```
mi.h
#ifndef _MI_H_
#define _MI_H_
extern int g1 ;
int f1( int p1, char p2 );
#endif
```

definiciones

```
mi.c
#include "mi.h"

int g1 = 10 ;
int f1( int p1, char p2 )
{
    return p1+(int)p2 ;
}
```

```
main.c
#include "mi.h"
#include <stdio.h>

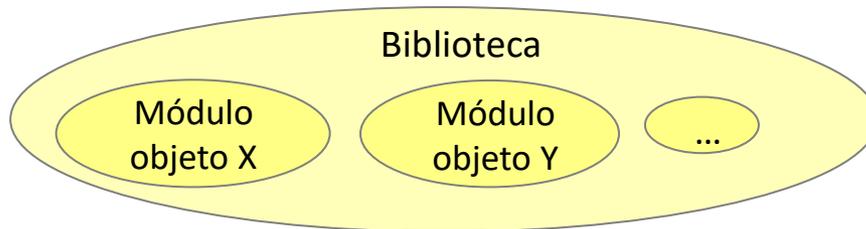
int main ( int argc,
           char *argv[] )
{
    int r ;

    r=f1(5,'0') ;
    printf("r=%d\n",r) ;
    return 0 ;
}
```

```
gcc -Wall -g -c mi.c -o mi.o
gcc -Wall -g -o main.o -c main.c
gcc -o main main.o mi.o
```

Biblioteca estática vs dinámica

- Biblioteca
 - ▣ Colección de módulos objetos relacionados



- Biblioteca **estática**
 - ▣ Carga y montaje en tiempo de compilación
- Biblioteca **dinámica**
 - ▣ Carga y montaje en tiempo de ejecución
 - ▣ Se indica al montar qué biblioteca usar, carga y montaje posterior

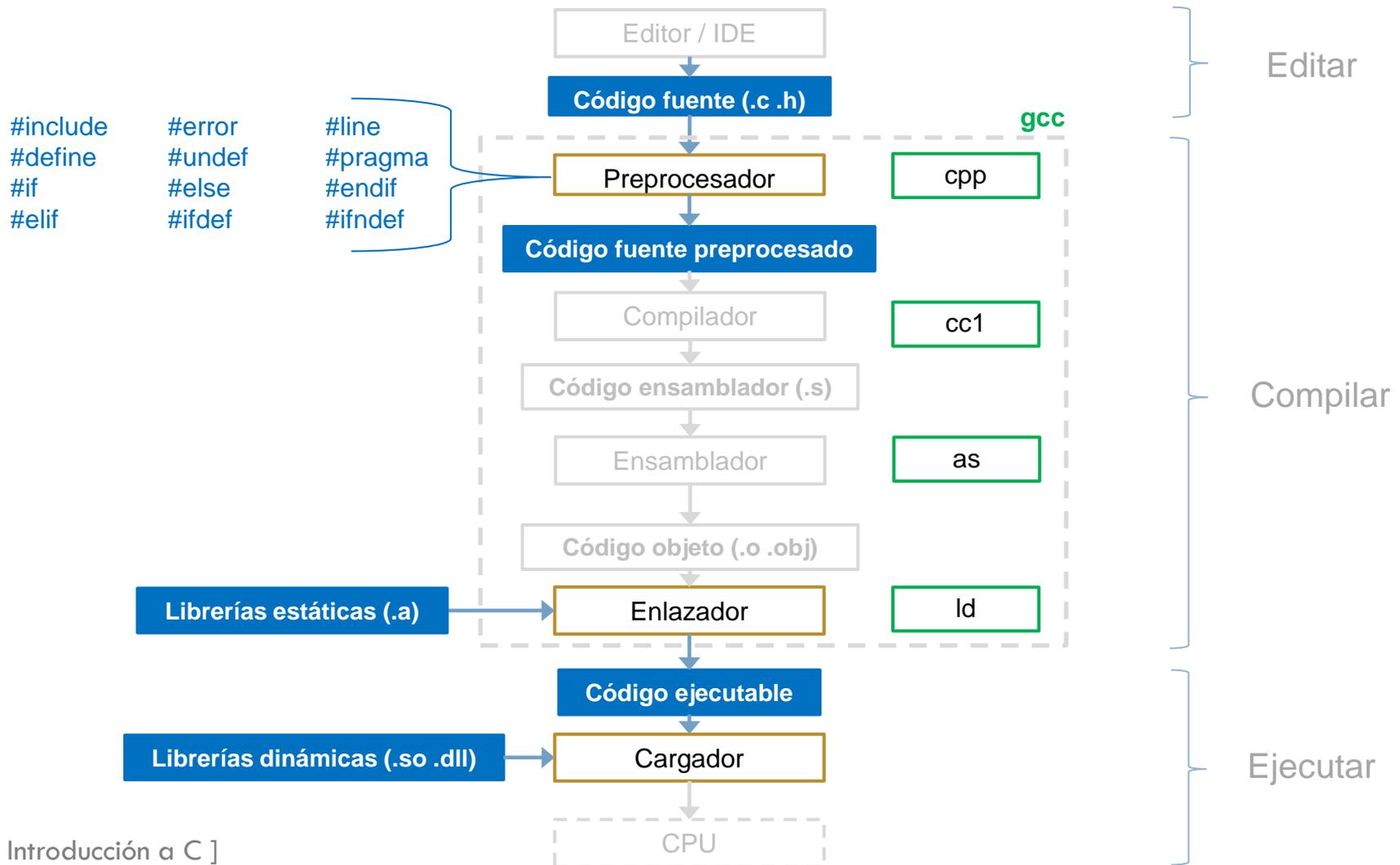
Modelo de compilación de C

¿Qué pasa al ejecutar `gcc -g hola.c -o hola`?

39



Félix García Carballera,
Alejandro Calderón Mateos



Ejemplo de biblioteca estática y dinámica

a.c

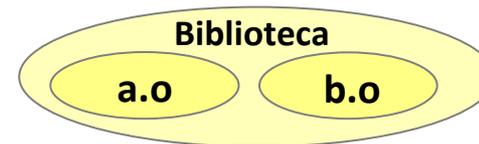
```
extern void decir ( char * str ) ;  
  
void decir_hola( void )  
{  
    decir("Hola mundo...\n") ;  
}
```

b.c

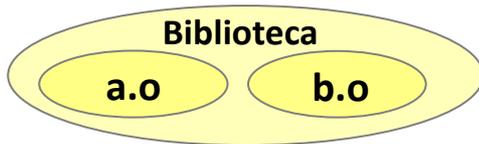
```
#include <stdio.h>  
  
void decir ( char * str )  
{  
    printf("%s",str) ;  
}
```

main.c

```
extern void decir_hola( void ) ;  
  
int main (int argc, char *argv[])  
{  
    decir_hola() ;  
    return 0 ;  
}
```



Ejemplo de biblioteca estática



□ Biblioteca **estática**

```
gcc -Wall -g -o a.o -c a.c  
gcc -Wall -g -o b.o -c b.c  
ar rcs libestatica.a a.o b.o
```

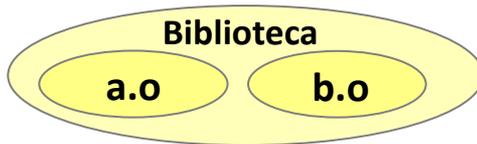
- Carga y montaje en tiempo de compilación

□ Biblioteca **dinámica**

```
gcc -Wall -g -o main.exe main.c -lestatica -L./  
./main.exe
```

- Carga y montaje en tiempo de ejecución
- Se indica al montar qué biblioteca usar, carga y montaje posterior

Ejemplo de biblioteca dinámica



□ Biblioteca **estática**

- Carga y montaje en tiempo de ejecución

```
gcc -Wall -g -fPIC -o a.o -c a.c  
gcc -Wall -g -fPIC -o b.o -c b.c  
gcc -shared -Wl,-soname,libdinamica.so \  
-o libdinamica.so.1.0 a.o b.o  
ln -s libdinamica.so.1.0 libdinamica.so
```

□ Biblioteca **dinámica**

- Carga y montaje en tiempo de ejecución
- Se indica al momento de ejecutar el programa

```
gcc -Wall -g -o main.exe main.c -ldinamica -L./  
env LD_LIBRARY_PATH=$LD_LIBRARY_PATH:./main.exe
```

Contenidos

- **Introducción al lenguaje C**
 - **Preprocesador**
 - **Comentarios**
 - **Bibliotecas**
 - **Tipos de datos básicos, variables y constantes**
 - **Asignación y conversión de tipos (casting)**
 - **Definición de tipos**
 - **Tipos compuestos: array y struct**
 - **Sentencias de control**
 - **Funciones**
 - **Punteros**

Tipos de datos básicos

Tipo		Con signo	Sin signo
Carácter	ASCII	<code>char c = 'a' ;</code> <code>signed char c = 'a' ;</code>	<code>char c = 'a' ;</code> <code>unsigned char c = 'a' ;</code>
Número Entero	corto (mitad)	<code>s = 1 ;</code> <code>short int s = 1 ;</code> <code>signed short s = 1 ;</code>	<code>unsigned short int s=1 ;</code> <code>unsigned short s=1 ;</code>
	por defecto	<code>int e = 2 ;</code> <code>signed int e = 2 ;</code>	<code>unsigned e = 2 ;</code> <code>unsigned int e = 2 ;</code>
	largo (doble)	<code>long e = 2 ;</code> <code>signed long e = 2 ;</code>	<code>unsigned long e = 2 ;</code>
Número Decimal	simple	<code>float f = 1.2 ;</code>	
	doble	<code>double f = 1.2 ;</code>	

Tipos de datos básicos

Tipo	C90	C99 (se añade a C90)
Enumerado	<pre>enum color { rojo, verde, azul } ; enum color c = azul;</pre>	
Booleano	<pre>int c = 0 ; // false c = 1 ; // true (valor !=0)</pre>	<pre>#include <stdbool.h> bool a=true, b=false;</pre>

Tipos enumerados

- Tipos definidos por el usuario con una enumeración finita de valores.

```
enum color { rojo, verde, azul } ;  
           //    0     1     2    //
```

```
enum color c ;  
c = azul; // c = 2;
```

- Detalles:
 - ▣ Tratamiento similar a los enteros.
 - ▣ No se pueden imprimir (se imprime el entero asociado).

Definición de tipos

- Se puede definir un alias de un tipo con typedef

```
typedef float medida;  
medida x = 1.0;
```

- “Si primero define una variable y luego se pone al inicio typedef entonces la variable se transforma en el nombre del nuevo tipo”

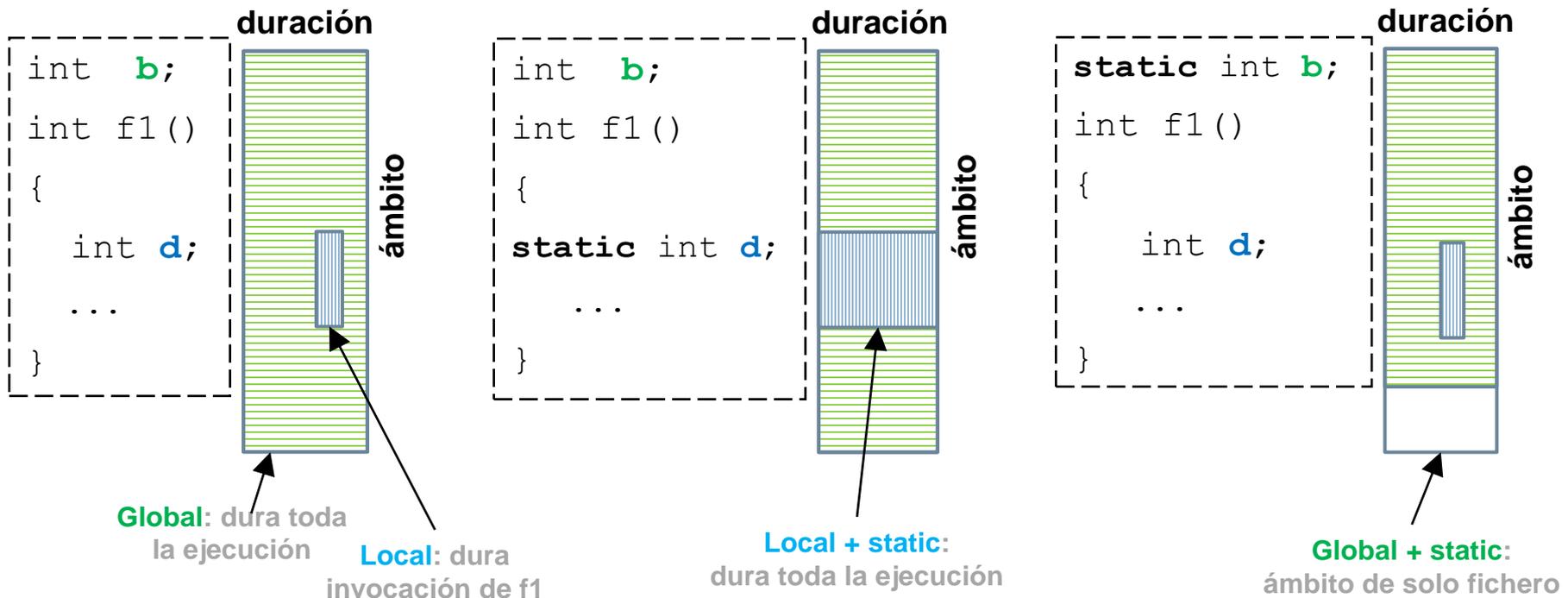
```
float medida ; // medida -> variable  
typedef float medida ; // medida -> tipo
```

Tipo boolean con typedef/enum

Tipo	C90	C99
Enumerado	<pre>enum color { rojo, verde, azul } ; color c = azul;</pre>	
Booleano	<pre>int c = 0 ; // false c = 1 ; // true (valor !=0) typedef int bool; enum { false, true } ; typedef int bool; #define true 1 #define false 0</pre>	<pre>#include <stdbool.h> bool a=true, b=false;</pre>

Variable: ámbito y duración

- **Ámbito:** zona de código donde se puede hacer referencia.
- **Duración:** espacio de tiempo que la variable persiste.



Variable: declaración y definición

- Declaración: asocia un tipo de datos a una o más variables.
 - ▣ **extern** tipo_de_datos var1, var2, ..., varN;
- Definición: declara y reserva espacio de memoria.
 - ▣ tipo_de_datos var1, var2, ..., varN;
- Deben declararse todas las variables antes de su uso.
Cada variable debe definirse solo una vez.

declaraciones

```
#ifndef _MI_H_
#define _MI_H_
extern int g1 ;
int f1( int p1, char p2 );
#endif
```

mi.h

definiciones

```
#include "mi.h"

int g1 = 10 ;
int f1( int p1, char p2 ) {
    return p1+(int)p2 ;
}
```

mi.c

Constantes

- Dos alternativas:
 - (**#define**) Utilizando el preprocesador:
 - `#define A1 4;`
 - Reemplaza texto en el archivo fuente antes de compilar.
 - (**const**) Declarando una variable como constante:
 - `const int A2 = 4;`
 - Permite comprobación de tipos.
- La 2ª opción es preferible, pero existe mucho código ya escrito que sigue usando la 1ª.

Constantes

```
#define PI_1    3.1416

const float PI_2 = 3.1416;

int main()
{
    float radio = 20.0 ;
    print("%e\n", 2*PI_1*radio) ;
    print("%e\n", 2*PI_2*radio) ;

    ...
    return 0;
}
```

Operadores de asignación

□ Asignación:

```
identificador = expresión
```

- El **operador de asignación** `=` y el de **igualdad** `==` son **distintos**

□ Asignaciones múltiples:

```
int i = j = 5 ;
```

- Las asignaciones se efectúan de derecha a izquierda. (5 se asigna a j y luego el valor de j se asigna a i)

Conversión de tipos o *casting*

casting implícito

- En C un **operador** se puede aplicar a dos variables o expresiones distintas.
- Ejemplo:
$$5 * 3.14 ;$$
- Conversión implícita:
 - ▣ Los operandos que difieren en tipo pueden sufrir una **conversión de tipo o casting**.
 - ▣ **Norma general:** El operando de menor precisión toma el tipo del operando de mayor precisión.

Conversión de tipos o *casting*

casting explícito

- Se puede convertir una expresión a otro tipo:

`(tipo datos) expresión`

- Ejemplos:

```
int a = (int) (5.5) % 4 ;
```

```
int b = (int) (5.5 % 4) ;
```

```
printf("%d\n", (int) 'a') ;
```

Reglas de asignación

- Si en una sentencia de asignación los dos operandos son de tipos distintos, entonces el valor del operando de la derecha será automáticamente convertido al tipo del operando de la izquierda.

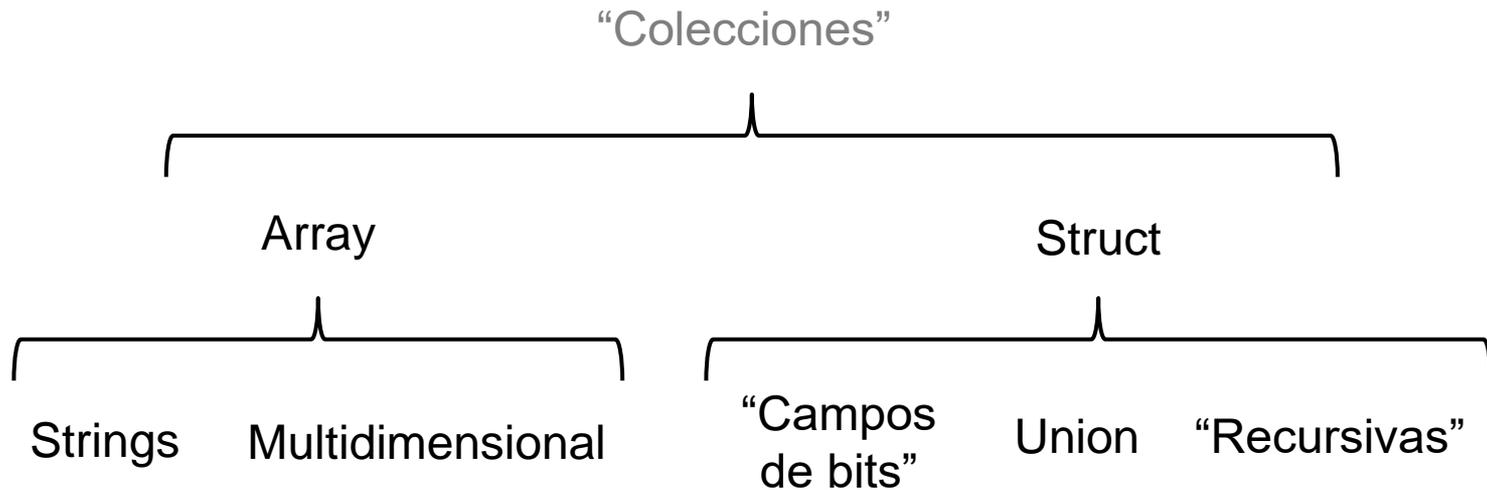
double d = 5.1 ;

- Además:
 - ▣ Entero = decimal \Rightarrow se puede truncar.
 - ▣ Simple = doble \Rightarrow puede redondearse.
 - ▣ Entero corto = entero \Rightarrow puede alterarse el valor entero resultante.
- Es **importante** en C usar correctamente la conversión de tipos.

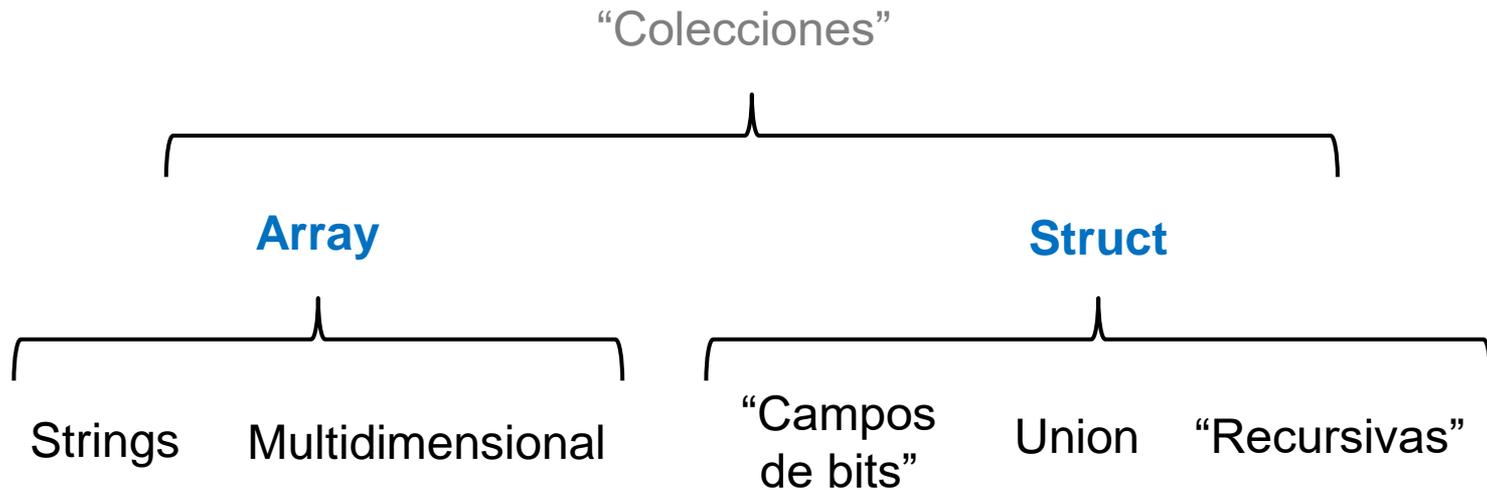
Contenidos

- **Introducción al lenguaje C**
 - Preprocesador
 - Comentarios
 - Bibliotecas
 - Tipos de datos básicos, variables y constantes
 - Asignación y conversión de tipos (casting)
 - Definición de tipos
 - **Tipos compuestos: array y struct**
 - Sentencias de control
 - Funciones
 - Punteros

Array, struct, union, ...



Array, struct, union, ...



Array vs Struct

	Array	Struct
Colección de...	Elementos del mismo tipo	Elementos de igual o distinto tipo
Definición	<pre>int arr1[10];</pre>	<pre>struct punto { char x ; int y ; }; struct punto p1 ;</pre>
Acceso	<pre>arr1[0] = arr1[9] = ...</pre>	<pre>p1.x = p1.y = ...</pre>

typedef

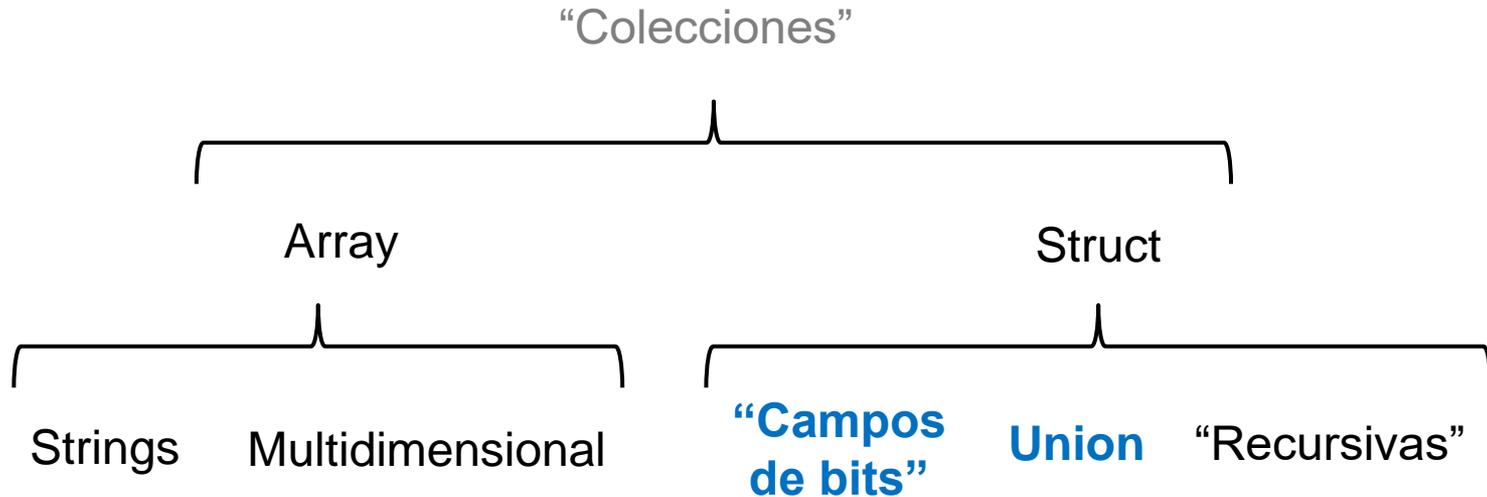
61

- Para struct y union el uso de typedef permite ahorrar tener que poner struct/union como parte del nombre del tipo.

```
struct punto {  
    char x ;  
    int y ;  
};  
  
// sin typedef -> "struct punto"  
struct punto var1 = { 1, 2 } ;  
  
typedef struct punto punto_t ;  
  
// con typedef  
punto_t var2 = { 1, 2 } ;  
...
```

eira,
ateos

Array, struct, union, ...



Campos de bits

63

- En los campos de una estructura (struct) que son números enteros se puede indicar el número de bits.

```
struct float32 {
    int signo:      1 ;
    int exponente:  8 ;
    int mantisa:   23 ;
};

struct float32  f1 ;

// dar valores dentro del rango
f1.signo      = 0 ;
f1.exponente = 0x0003 ;
f1.mantisa    = 0x12345 ;

// "overflow"
f1.signo      = 3 ;
f1.exponente = 257 ;
...
```

eira,
ateos

Uniones

64

- Una union se diferencia de un struct en que los campos están solapados en memoria.
- El tamaño de la unión es el tamaño del campo de mayor tamaño.

```
struct float32 {
    int signo:      1 ; 
    int exponente:  8 ; 
    int mantisa:   23 ; 
};

union vistas {
    struct float32 partes ;  
    float          valor ; 
};

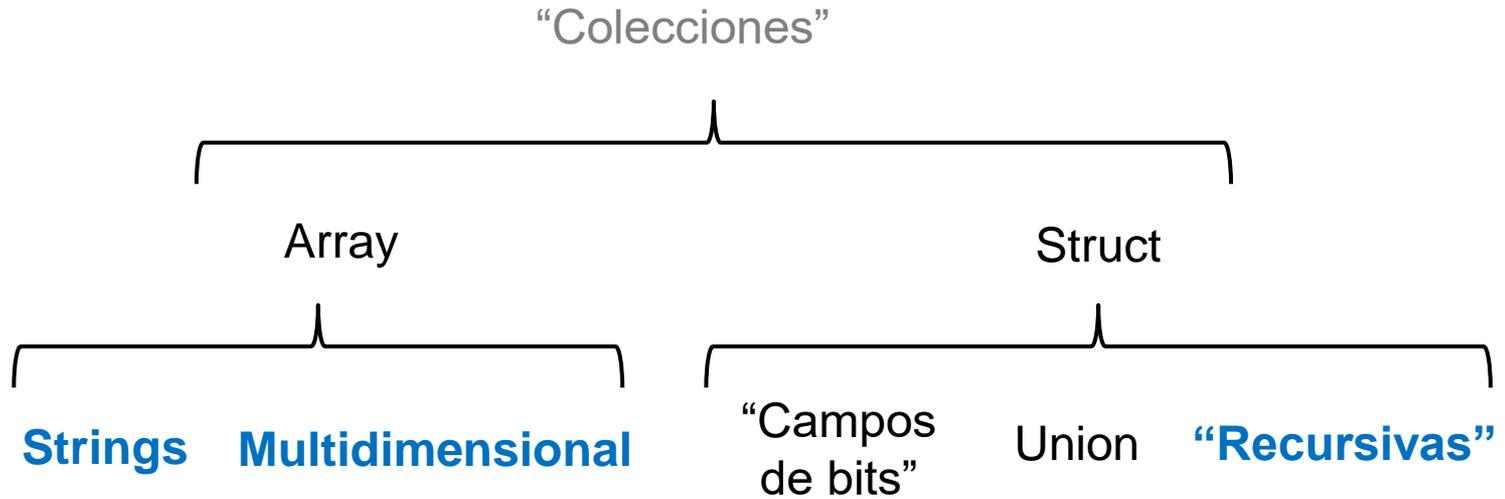
union vistas u1 ;

u1.valor = 3.14 ;

printf(“%d\n”, u1.partes.signo) ;
printf(“%d \n”, u1.partes.exponente) ;
printf(“%d \n”, u1.partes.mantisa) ;
```

eira,
ateos

Array, struct, union, ...



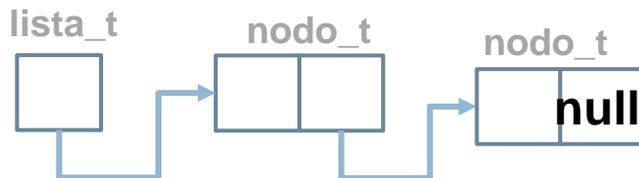
{array, struct} x {tipo, array, struct}

	Array	Struct
Tipo básico	<pre>char s1[12] = "hola" ; char s2[12] = {'h','o','l','a','\0'} ; s1[1] = 'o' ;</pre>	<pre>struct punto { int x ; int y ; } ; struct punto p1 ;</pre>
Array	<pre>int mat1[3][4] = { {0, 1, 3, 5}, {1, 2, 3, 4}, {2, 4, 8, 9} } ; mat1[1][2] = 2 ;</pre>	<pre>struct ficha { char nombre[20]; int id ; } ; struct ficha f1 = { "n1", 2 } ;</pre>
Struct	<pre>struct ficha fichas1[100] ; fichas1[10].id = 0 ;</pre>	<pre>struct registro { struct ficha fichas[10]; int en_uso ; } r1 ;</pre>

{struct} x {struct}: recursivas

67

- **No** se puede definir un campo de una estructura de tipo estructura (definición recursiva directa).
- Pero **si** el campo puede ser un puntero a la propia estructura.



```
struct nodo {  
    struct nodo nodo;  
    int valor ;  
};
```

```
struct nodo {  
    struct nodo *siguiente;  
    int valor ;  
};
```

```
typedef struct nodo  nodo_t ;  
typedef struct nodo * lista_t ;
```

```
lista_t lista1 = NULL ;  
lista1 = malloc(sizeof(nodo_t)) ;  
lista1->valor = 1 ;  
lista1->siguiente = NULL;  
...
```

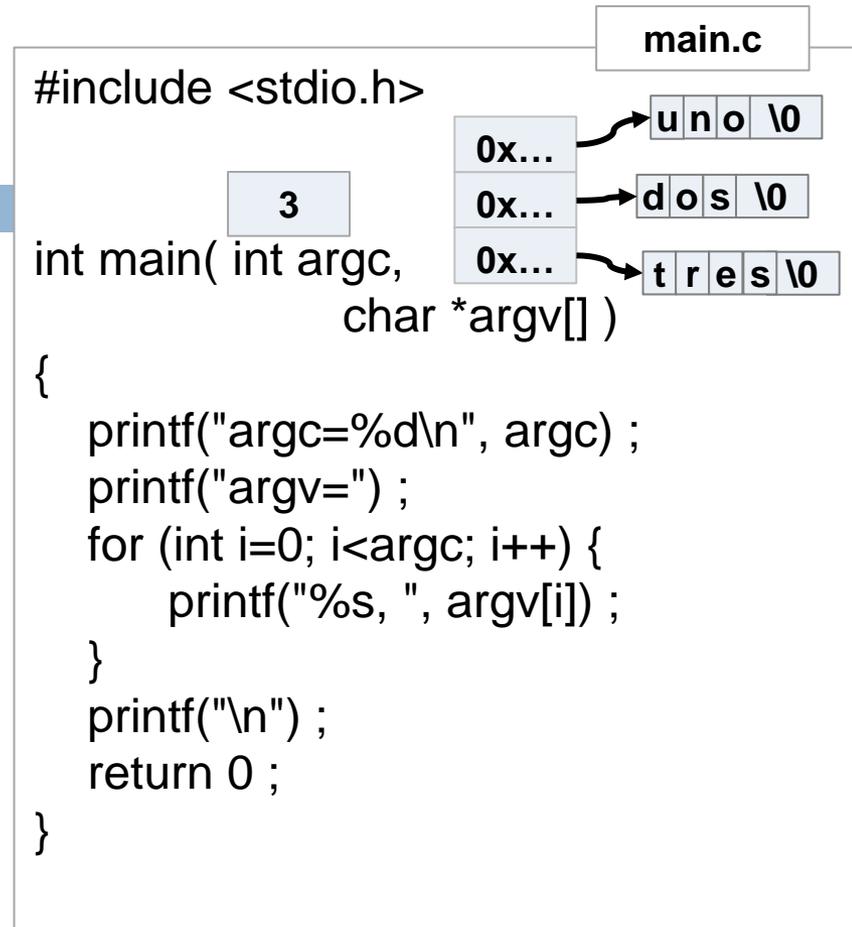
{array} x {tipo}: char [] => String

	Cadena de caracteres (String)	
Tipo básico	<pre>char s1[12] = { 'h', 'o', 'l', 'a', '\0' }; char s2[12] = "hola"; char s3[] = "hola"; char *s4 = "hola";</pre>	
Operaciones típicas	Longitud	<pre>int s1_length = strlen(s1);</pre>
	Comparar	<pre>if (strcmp(s1, s2) == 0) { printf("Iguales"); }</pre>
	Concatenar	<pre>strcat(s2, s1); // s2 = "holahola";</pre>
	Leer	<pre>scanf("%s", s2);</pre>
	Imprimir	<pre>printf("%s", s2);</pre>
	Copiar	<pre>strcpy(s2, "prueba"); // s2 = "prueba"</pre>
	String a entero	<pre>int v1 = atoi("123");</pre>

char [] => String

69

- Los argumentos pasados al programa llegan a la función main a través de `argc` y `argv`.
- ▣ **argc** es el número de argumentos.
- ▣ **argv** es un vector de cadenas de caracteres.



```
gcc -g -Wall main.c -o main
./main uno dos tres
argc=3
argv=uno, dos, tres,
```

Contenidos

- **Introducción al lenguaje C**
 - **Preprocesador**
 - **Comentarios**
 - **Bibliotecas**
 - **Tipos de datos básicos, variables y constantes**
 - **Asignación y conversión de tipos (casting)**
 - **Tipos compuestos: array y struct**
 - **Definición de tipos**
 - **Sentencias de control**
 - **Funciones**
 - **Punteros**

Sentencias de control

- Condicionales
 - Alternativas
 - if
 - if-else
 - switch
 - Iterativas
 - for
 - while
 - do-while
- Incondicionales
 - continue
 - break
 - goto

Sentencias de control

□ Condicionales

□ Alternativas

- if
- if-else
- switch

□ Iterativas

- for
- while
- do-while

□ Incondicionales

- continue
- break
- goto

Sentencias control: alternativas

73

<https://www.learn-c.org/>

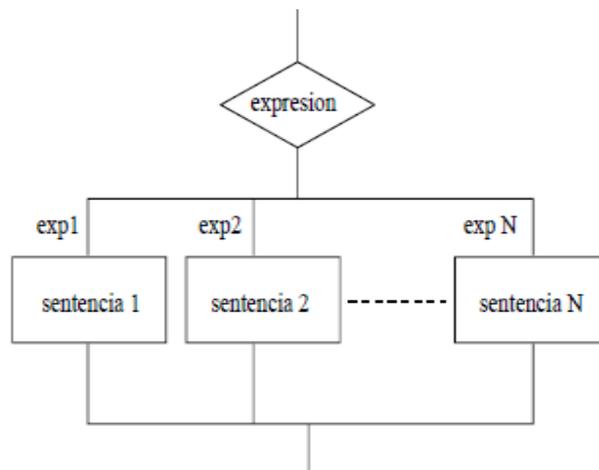


Félix García Carballeira,
Alejandro Calderón Mateos

Característica	Python	C
Alternativa (if)	<pre>if b > a: print("b > a")</pre>	<pre>if (b > a) { printf("b > a"); }</pre>
Alternativa con exclusión (if - else)	<pre>if b > a: print("b > a") else # elif print("b <= a")</pre>	<pre>if (b > a) { printf("b > a"); } else { printf("b <= a"); }</pre>
Alternativa múltiple (switch)		<pre>switch (x) { case 3: // ... break ; default: // ... }</pre>

Estructuras de control (alternativas)

- Estructuras de evaluación de condición: **if** y **switch**
 - ▣ Las condiciones son expresiones de tipo entero.
 - ▣ **switch** evitaría **if** encadenados:



```
switch (expresión)
{
    case exp1:
        sentencia1;
        ...
        break;

    case expX:
    case expY:
        sentenciaX1;
        ...
        break;

    default:
        sentenciaD1;
        ...
        break;
}
```

Estructuras de control (alternativas)

- Estructuras de evaluación de condición: **if** y **switch**
 - ▣ Mejor ($0 == \text{variable}$) que ($\text{variable} == 0$): cuidado **if** ($x=1$)...
 - ▣ Un **if** que precisa *scroll* de pantalla es difícil de leer

```
division_entera ( int par1, int part2 )
{
    int valor ;

    if (part2 != 0)
    {
        valor = par1 / par2 ;
    }
    else {
        valor = 0 ;
        printf("ERROR: par2 es cero\n") ;
    }

    return valor ;
}
```

```
division_entera ( int par1, int part2 )
{
    int valor = 0 ;

    // comprobar parámetros
    if (0 == part2) {
        printf("ERROR: par2 es cero\n") ;
        return valor ;
    }

    // caso correcto
    valor = par1 / par2 ;
    return valor ;
}
```

Sentencias de control

- Condicionales

 - Alternativas

 - if
 - if-else
 - switch

 - Iterativas

 - for
 - while
 - do-while

- Incondicionales

 - continue
 - break
 - goto

Sentencias control: iterativas

77

<https://www.learn-c.org/>



Félix García Carballera,
Alejandro Calderón Mateos

Característica	Python	C
for (0 ... N)	<pre>for x in range(6): print(x)</pre>	<pre>for (int x=0; x<6; x++) { printf("%d", x); }</pre>
while (0 ... N)	<pre>l = 1 while i < 6: print(i) i += 1 # break (end) vs # continue (skip 1)</pre>	<pre>int i = 1 ; while (i < 6) { printf("%d", i); i++; // break (end) vs // continue (skip 1) }</pre>
until (1 ... N)		<pre>int i = 1 ; do { // ... } while (i < 6) ;</pre>

Estructuras de control (iterativas)

- ▣ Estructuras de repetición: `while`, `do-while`, `for`
 - ▣ El índice `for` se debe declarar como otra variable más:

```
int main()
{
    int i;

    for (i=0; i<10; i++) {
        printf("hola\n");
    }
    return 0;
}
```

Sentencias de control

- Condicionales

- Alternativas

- if
 - if-else
 - switch

- Iterativas

- for
 - while
 - do-while

- Incondicionales

- continue
 - break
 - goto

Estructuras de control (continue)

- ▣ Estructuras de repetición: `while`, `do-while`, `for`
 - ▣ Uso de `continue` permite saltar ciertas iteraciones.

```
int main()
{
    int i;

    for (i=0;i<10;i++)
    {
        if ((i % 2) == 0)
            continue;
        printf("impar\n");
    }
    return 0;
}
```

Estructuras de control (break)

- ▣ Estructuras de repetición: **while**, **do-while**, **for**
 - ▣ **break** permite salir del bucle/switch cercano al break.

```
int main()
{
    int i, j;

    for (i=0; i<10; i++) {
        for (j=0; j<10; j++) {
            printf("diagonal\n");
            if (j > i)
                break;
        }
    }
    return 0;
}
```

```
switch (expresión)
{
    case exp1:
        sentencia1;
        ...
        break;

    case expX:
    case expY:
        sentenciaX1;
        ...
        break;

    default:
        sentenciaD1;
        ...
        break;
}
```

Estructuras de ~~des~~control (for goto it)

- ▣ Poco recomendable el uso de `goto`
- ▣ **No se puede usar en prácticas (salvo que el enunciado de forma explícita lo permita).**

```
int f1 ( char * p1)
{
    if (p1 == NULL)
        goto error;

    ... //

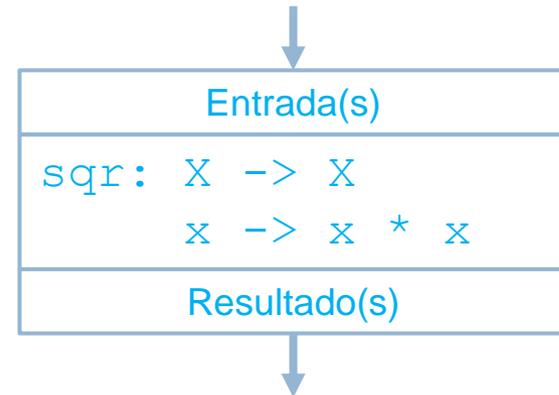
error:
    return -1;
}
```

Contenidos

- **Introducción al lenguaje C**
 - **Preprocesador**
 - **Comentarios**
 - **Bibliotecas**
 - **Tipos de datos básicos, variables y constantes**
 - **Asignación y conversión de tipos (casting)**
 - **Tipos compuestos: array y struct**
 - **Definición de tipos**
 - **Sentencias de control**
 - **Funciones**
 - **Punteros**

Inspiración de las funciones de C

- Inspiración de las funciones matemáticas:
 - Argumentos x
 - Resultado $f(x)$
 - Relación x y $f(x)$



Inspiración de las funciones de C

- Inspiración de las funciones matemáticas:
 - Argumentos x
 - Resultado $f(x)$
 - Relación x y $f(x)$

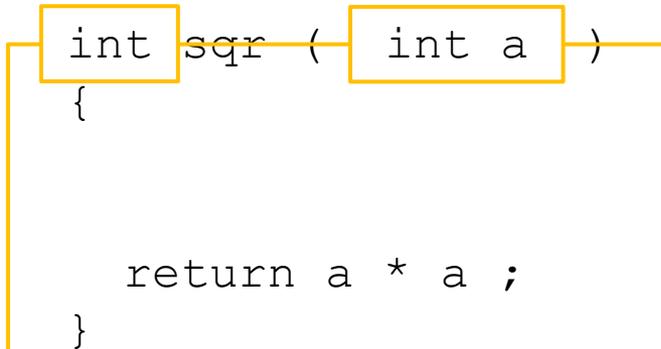
```
resultado(s)      argumento(s)
int sqr ( int a )
{
    return a * a ;
}
                cómputo
```

Inspiración de las funciones de C

- Inspiración de las funciones matemáticas:
 - Argumentos x
 - Resultado $f(x)$
 - Relación x y $f(x)$
- Función pura:
“Lo que pasa en una función se queda en una función”.

resultado(s) argumento(s)

```
int sqr ( int a )  
{  
  
    return a * a ;  
}
```



Inspiración de las funciones de C

- Inspiración de las funciones matemáticas:
 - Argumentos x
 - Resultado $f(x)$
 - Relación x y $f(x)$
- Función pura
- Efecto colateral

```
int sqr ( int a )  
{  
    printf("%d", a);  
    return a * a ;  
}
```

resultado(s)
fuera del
ámbito de
la función

Que no son funciones

- En C solamente hay **funciones**, no hay clases ni métodos.
- Una función **NO**...:
 - No se puede definir una función dentro de otra.
 - En una función no hay tratamiento de excepciones.
 - No se pueden sobrecargar funciones (usar el mismo nombre de función con distintos argumentos).

Prototipo de una función: declaración y definición

- **Prototipo** de una función es su declaración.
 - **Ejemplo:**

```
float potencia (float x, int y); /* prototipo */
float potencia (float x, int y) /* definición */
{
    int i;
    float prod = 1;
    for (i = 0; i < y; i++)
        prod = prod * x;
    return(prod);
}
```

Prototipo de una función: declaración y definición

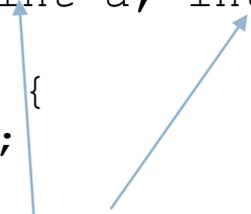
- Permite la comprobación de errores entre la llamada a una función y la definición de la función correspondiente.

```
#include <stdio.h>

int suma (int a, int b);      /* Prototipo */

int main() {
    int x, y;
    x=3;
    y = suma(x, 2);          /* Llamada a la función */
    printf("%d + %d = %d\n", x, 2, y);
    return 0;
}

int suma (int a, int b) {    /* Definición */
    return a+b;
}
```



Definición de una función

```
tipo nombre (tipo1 arg1, ..., tipoN argN)
{
    /* CUERPO DE LA FUNCIÓN */
}
```

- ▣ Una función devuelve un valor de tipo **tipo**:
 - Si se omite tipo se considera que devuelve un `int`.
 - Si no devuelve ningún tipo: `void`.
- ▣ Una función acepta un conjunto de argumentos:
 - Si se omite argumentos entonces equivale a `(int)`
 - Si no tiene argumentos: `void explicacion(void) { ... }`
- ▣ La última sentencia de una función es **return valor**;
 - Finaliza la ejecución y devuelve valor a la función que realizó la llamada.

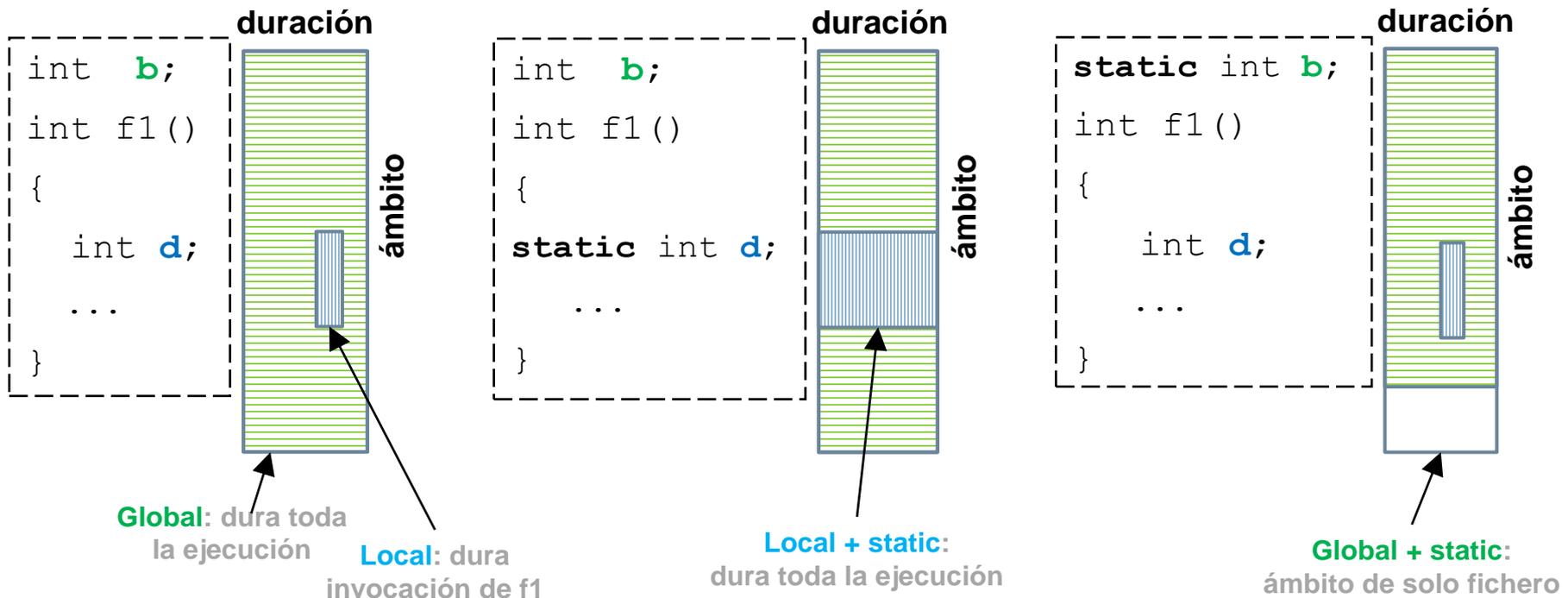
Variables locales y globales

- Las variables que se declaran dentro de una función, son **locales** a esa función.
 - ▣ Existen solamente durante la ejecución de la función.
 - ▣ Se almacenan en la pila.

- Las variables que se declaran fuera de las funciones, son **globales**.
 - ▣ Se puede acceder a ellas desde cualquier función.
 - ▣ Poco deseable, pero **necesario** a veces.

Variable: ámbito y duración

- **Ámbito:** zona de código donde se puede hacer referencia.
- **Duración:** espacio de tiempo que la variable persiste.



Paso de parámetros



```
void prueba2 ( int j, char c, float f, int pj )  
{  
    /* ... */  
}
```

Paso de parámetros

```
i = 10 ;  
float PI = 3.14 ;  
  
prueba2 ( i,      'a',      PI,      &i ) ;  
...
```

```
void prueba2 ( int j, char c, float f, int pj )  
{  
    /* ... */  
}
```

Paso de parámetros

```
i = 10 ;  
float PI = 3.14 ;  
  
prueba2 ( i,      'a',      PI,      &i ) ;  
...
```

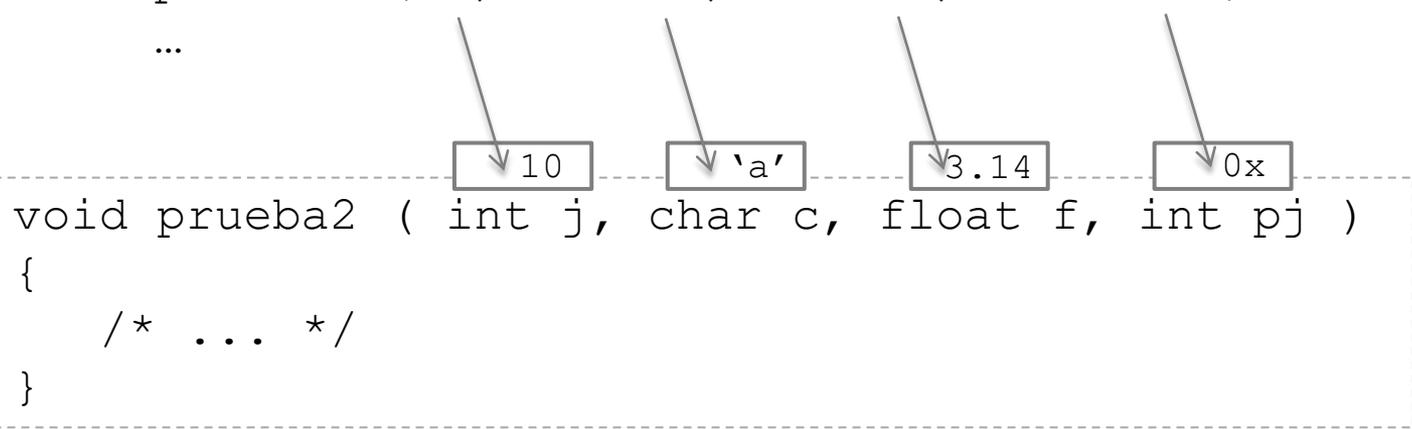
```
void prueba2 ( , , ,  )  
{  
    /* ... */  
}
```

1) Se crea en pila las variables formales

Paso de parámetros

```
i = 10 ;  
float PI = 3.14 ;
```

```
prueba2 ( i, 'a', PI, &i ) ;  
...
```



```
void prueba2 ( int j, char c, float f, int pj )  
{  
    /* ... */  
}
```

2) Se copia el valor de los parámetros reales

Paso de parámetros

Siempre se realiza una copia de los parámetros

```
i = 10 ;  
float PI = 3.14 ;
```

```
prueba2 ( i, 'a', PI, &i ) ;  
...
```



```
void prueba2 ( int j, char c, float f, int pj )  
{  
    /* ... */  
}
```

Paso de parámetros

Paso de parámetro por valor

```
#include <stdio.h>

int main (void)
{
    int i = 10;

    /* ... */
    inc(i) ;
    /* ... */
}
```

```
void inc ( int j )
{
    j = j + 1 ;
}
```

Paso de parámetros

Paso de parámetro por valor

```
#include <stdio.h>

int main (void)
{
    int i = 10;

    /* ... */
    inc(i) ;
    /* ... */
}
```

1) se copia
i en j

```
10
void inc ( int j )
{
    j = j + 1 ;
}
```

Paso de parámetros

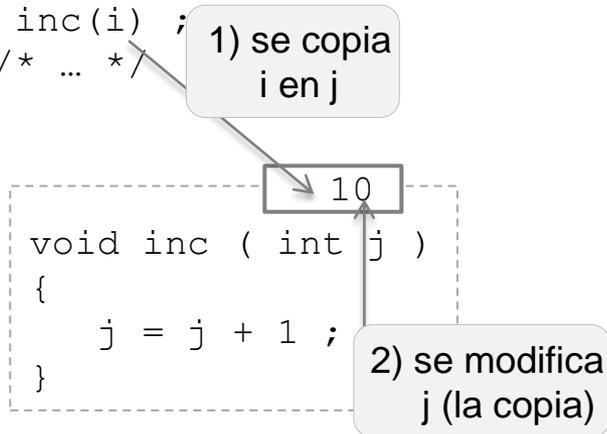


Paso de parámetro por valor

```
#include <stdio.h>

int main (void)
{
    int i = 10;

    /* ... */
    inc(i) ;
    /* ... */
}
```



Paso de parámetros

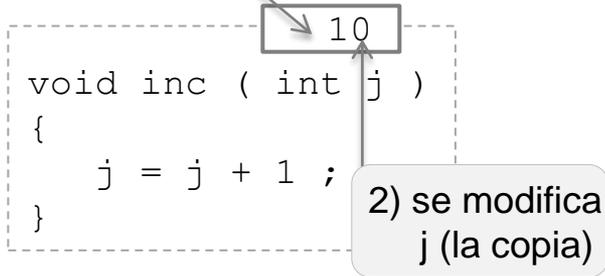
Paso de parámetro por valor

Paso de parámetro por referencia

```
#include <stdio.h>

int main (void)
{
    int i = 10;

    /* ... */
    inc(i) ;
    /* ... */
}
```



Paso de parámetros



Paso de parámetro por valor

```
#include <stdio.h>

int main (void)
{
    int i = 10;

    /* ... */
    inc(i) ;
    /* ... */
}
```

1) se copia
i en j

```
void inc ( int j )
{
    j = j + 1 ;
}
```

2) se modifica
j (la copia)

Paso de parámetro por referencia

```
#include <stdio.h>

int main (void)
{
    int i = 3;

    /* ... */
    inc(&i) ;
    /* ... */
}
```

```
void inc ( int *j )
{
    *j = *j + 1 ;
}
```

Paso de parámetros



Paso de parámetro por valor

```
#include <stdio.h>

int main (void)
{
    int i = 10;

    /* ... */
    inc(i);
    /* ... */
}
```

1) se copia i en j

```
void inc ( int j )
{
    j = j + 1 ;
}
```

2) se modifica j (la copia)

Paso de parámetro por referencia

```
#include <stdio.h>

int main (void)
{
    int i = 3;

    /* ... */
    inc(&i);
    /* ... */
}
```

1) se copia &i en j

```
void inc ( int *j )
{
    *j = *j + 1 ;
}
```

Paso de parámetros

para paso por referencia se usan punteros



Félix García Carballeira,
Alejandro Calderón Mateos

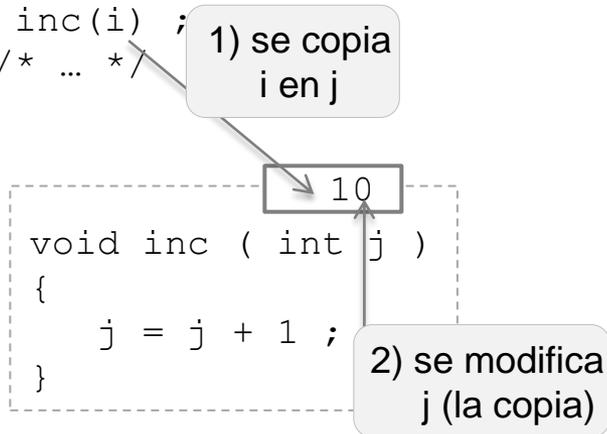
105

Paso de parámetro por valor

```
#include <stdio.h>

int main (void)
{
    int i = 10;

    /* ... */
    inc(i);
    /* ... */
}
```

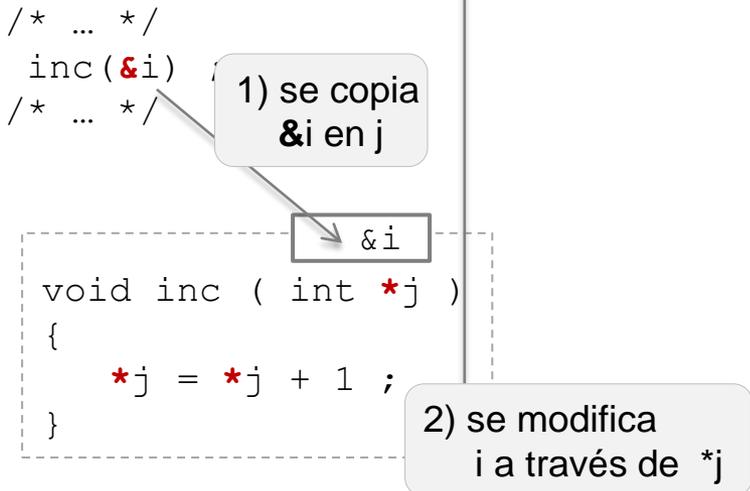


Paso de parámetro por referencia

```
#include <stdio.h>

int main (void)
{
    int i = 3;

    /* ... */
    inc(&i);
    /* ... */
}
```



Funciones (resumen)

- En C no hay clases ni métodos, solamente hay **funciones**.
- Una función:
 - Tiene que haberse declarado antes de usarse.
 - Acepta un **conjunto de parámetros [0,n]** y devuelve un **resultado [0,1]**.
 - El paso de parámetros es siempre por **valor** (se copian).
- Una función NO:
 - No se pueden sobrecargar los nombres de las funciones.
 - No se puede definir una función dentro de otra.
 - No hay tratamiento de excepciones.

Función como tipo de dato en C

Tipo	Array	Funciones
Concepto	Colección de elementos del mismo tipo.	Colección de sentencias (array de bytes de código).
Definición	<pre>int arr1 [10] = { 0x1, 10, 020 } ;</pre>	<pre>int fun1 (char ch1) { // sentencias }</pre>
Como es visto por el programador/a	<ul style="list-style-type: none">• arr1 es una variable de tipo array.• arr1 solo es la dirección del primer elemento del array.	<ul style="list-style-type: none">• fun1 es una constante de tipo función que acepta un char y devuelve un entero.• fun1 solo es la dirección del primer byte de código.

Función como tipo de dato en C

- Es posible que una variable sea de tipo puntero a función... y usar una función para darle valor:

```
int v1[10] ;  
char v2[10] ;
```

```
imprimir_entero ( void *v, int i ) {  
    printf("%d, ", ((int *)v)[i]) ;  
}
```

```
int (*pv1)[10] = v1;  
void (*fimpres) (void *v, int i) = imprimir_entero ;
```

```
int *pv1 [10] = v1; // array de 10 punteros a entero  
int (*pv1)[10] = v1; // puntero a array de 10 enteros
```

Función como tipo de dato en C

- Es posible que una variable sea de tipo puntero a función... y usar una función para darle valor:

```
int v1[10] ;  
char v2[10] ;
```

```
imprimir_entero ( void *v, int i ) {  
    printf("%d, ", ((int *)v)[i]) ;  
}
```

```
int (*pv1)[10] = v1;  
void (*fimpres) (void *v, int i) = imprimir_entero ;
```

```
void *fimpres (void *v, int i) // función... devuelve puntero a void  
void (*fimpres) (void *v, int i) // puntero función...
```

Función como tipo de dato en C

- Es posible que una función trabaje con variables que son funciones (metafunciones).

```
int v1[10] ;
char v2[10] ;

void imprimir_entero ( void *v, int i ) {
    printf("%d, ", ((int *)v)[i]) ;
}

int imprimir_vector( int neltos, void *vector,
                    void (*fimpre)(void *v, int i) )
{
    for (int k=0; k<neltos; k++) {
        (*fimpre)(vector, k) ;
    }
}

imprimir_vector(10, v1, imprimir_entero) ;
```

Función como tipo de dato en C

- Es posible que una variable sea de tipo puntero a función... y usar una función para darle valor:

```
int v1[10] ;
char v2[10] ;

void imprimir_entero ( void *v, int i ) {
    printf("%d, ", ((int *)v)[i]) ;
}

typedef int (*ptrarr_t)[10];
ptrarr_t pv1 = v1;

typedef void (*fprnt_t)(void *v, int i) ;
fprnt_t fimprel = imprimir_entero ;
```

Contenidos

- **Introducción al lenguaje C**
 - **Preprocesador**
 - **Comentarios**
 - **Tipos de datos básicos, variables y constantes**
 - **Asignación y conversión de tipos (casting)**
 - **Tipos compuestos: array y struct**
 - **Definición de tipos**
 - **Sentencias de control**
 - **Funciones**
 - **Punteros**

- **Introducción a punteros**
- Casos de uso típicos:
 - Iterador: `for (int *p=v; p<&(v[10]); p++) ...`
 - Memoria dinámica: `p = malloc(10); ...`
 - Paso de parámetros: `f1 (&a, &b) ;`
- Aritmética de punteros

Dirección, valor y tamaño



Félix García Carballera,
Alejandro Calderón Mateos

00	'a'
01	222
02	0x3F
03	'&'
...	...

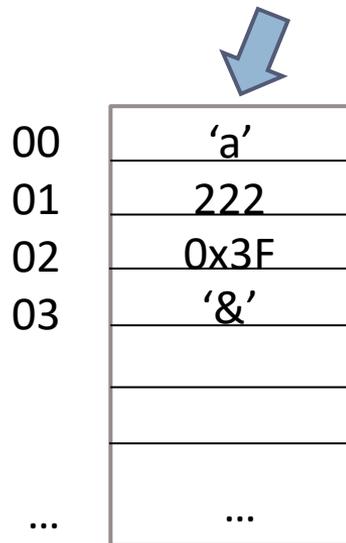
Cualquier variable de un tipo:
char chr1 = 'a' ;
Tiene asociada una terna (d, v, t)

Dirección, valor y tamaño



□ Valor

- Elemento guardado en memoria.

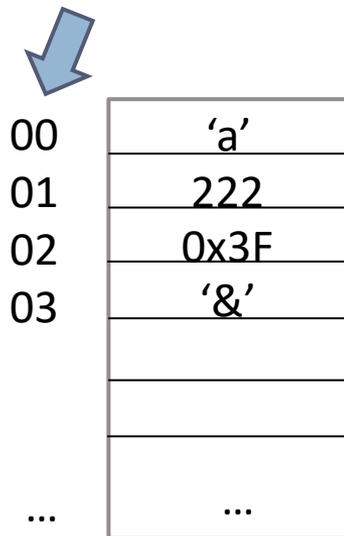


Cualquier variable de un tipo:

```
char chr1 = 'a' ;
```

Tiene asociada una terna (d, v, t)

Dirección, valor y tamaño

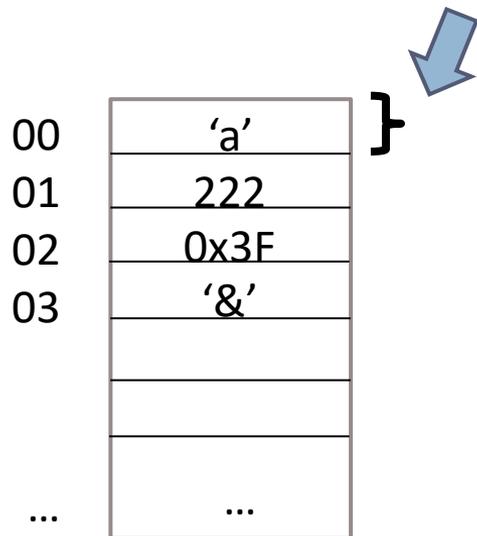


- Valor
 - Elemento guardado en memoria.

- Dirección
 - Posición de memoria.

Cualquier variable de un tipo:
`char chr1 = 'a' ;`
Tiene asociada una terna (d, v, t)

Dirección, valor y tamaño



□ Valor

- Elemento guardado en memoria.

□ Dirección

- Posición de memoria.

□ Tamaño

- Número de bytes necesarios para almacenar el valor.

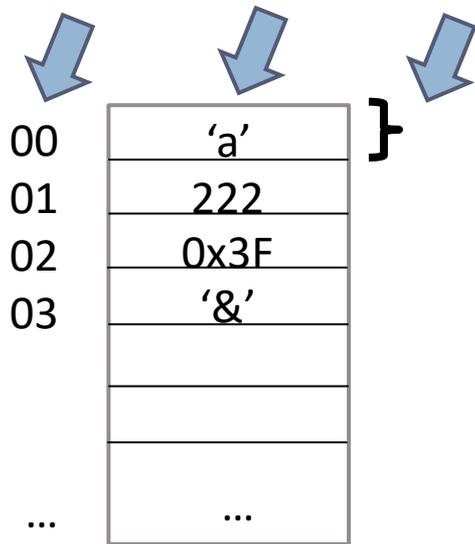
Cualquier variable de un tipo:

```
char chr1 = 'a' ;
```

Tiene asociada una terna (d, v, t)

Dirección, valor y tamaño

hablamos de dirección de memoria, todavía no de punteros



Cualquier variable de un tipo:
char chr1 = 'a';
Tiene asociada una terna (d, v, t)

- Valor
 - Elemento guardado en memoria a partir de una dirección, y que ocupa un cierto tamaño para ser almacenada.
- Dirección
 - Número que identifica la posición de memoria (celda) a partir de la cual se almacena el valor de un cierto tamaño.
- Tamaño <-> Tipo de datos
 - Número de bytes necesarios a partir de la dirección de comienzo para almacenar el valor de un tipo de datos.

Valor, dirección, tamaño: ejemplo 1

```
#include <stdio.h>

int main ( int argc,
           char *argv[] )
{
    int i = 3;

    printf("%d ", i);
    printf("%d ", &i);
    printf("%d", sizeof(i));

    return 0;
}
```

en definición: **int i** ;

Variable **i** de tipo entero

en uso: **<var>**

Valor de...

en uso: **&** **<var>**

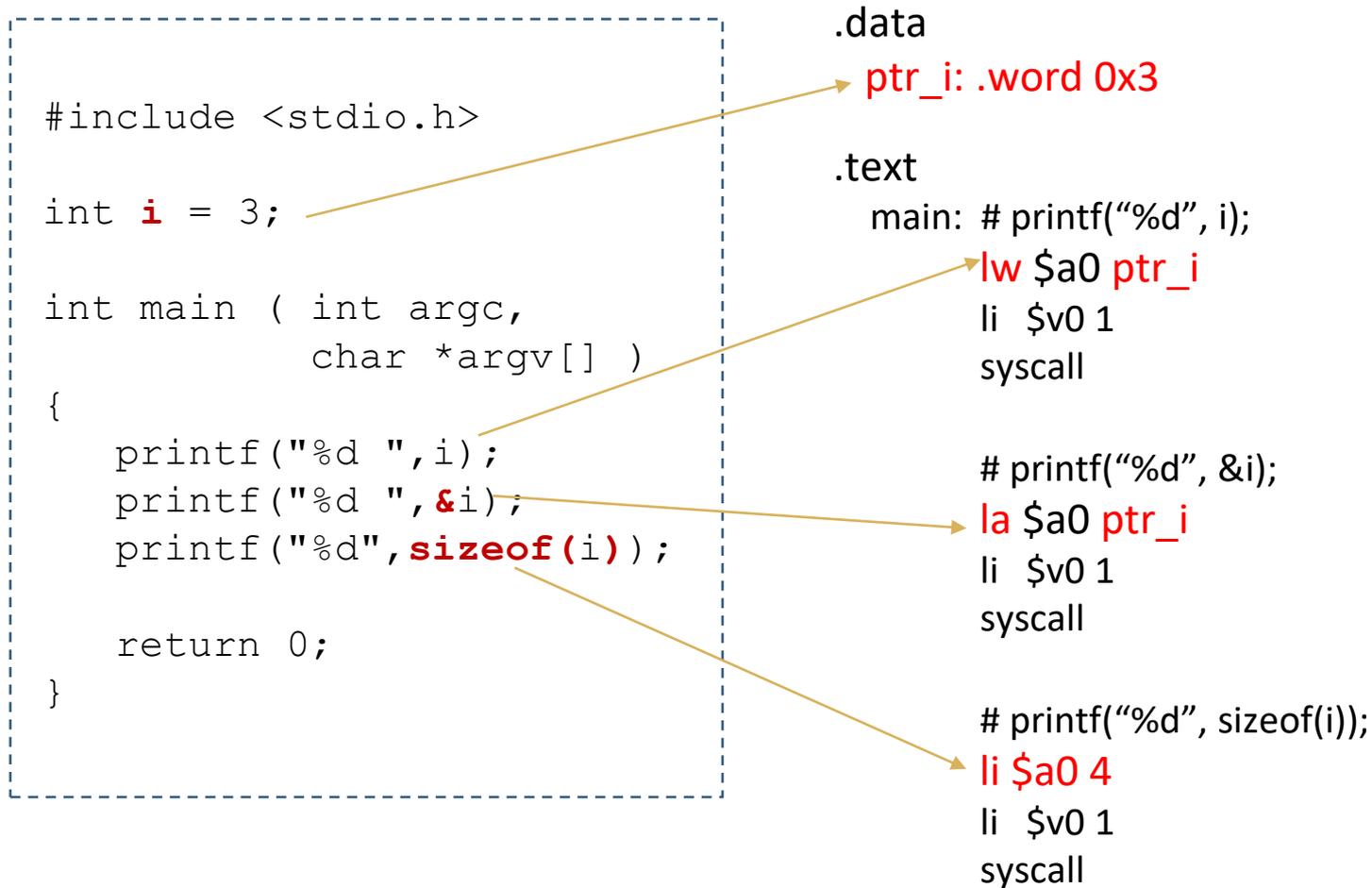
Asignada en tiempo de
compilación (ro)

Dirección de...

en uso: **sizeof(<var>)**

Tamaño en bytes de...

Valor, dirección, tamaño: ejemplo 1



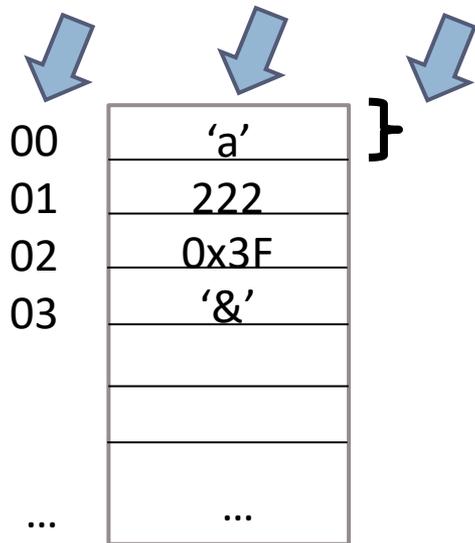
Tipo de datos puntero

dirección que apunta a un elemento de tipo x...



Félix García Carballera,
Alejandro Calderón Mateos

121



variable **puntero** (*) representa la dirección a un elemento de un tipo:

```
int * ptr_int = 0x0 ;
```

Y tiene asociada una terna (d, v, t) 😊

- **Valor**
 - Elemento guardado en memoria a partir de una dirección, y que ocupa un cierto tamaño para ser almacenada.
- **Dirección**
 - Número que identifica la posición de memoria (celda) a partir de la cual se almacena el valor de un cierto tamaño.
- **Tamaño**
 - Número de bytes necesarios a partir de la dirección de comienzo para almacenar el valor

Valor, dirección, tamaño: ejemplo 2

si hay puntero entonces hay dirección de memoria



Félix García Carballera,
Alejandro Calderón Mateos

122

```
#include <stdio.h>

int main ( int argc,
           char *argv[] )
{
    int i = 3;
    int *pi = &i;

    printf("%ld ", pi);
    printf("%d ", *pi);
    printf("%d", sizeof(*pi));
    return 0;
}
```

en definición: **int *pi;**

Puntero a entero

en uso: **<var>**

Dirección de...

en uso: *** <exp>**

Asignada en tiempo de ejecución (rw)

Valor contenido en...

en uso: **sizeof(<tipo>)**

Tamaño en bytes de...

Uso básico de la memoria

interfaz funcional



Félix García Carballeira,
Alejandro Calderón Mateos

123

00	'a'
01	222
02	0x3F
03	'&'
...	...

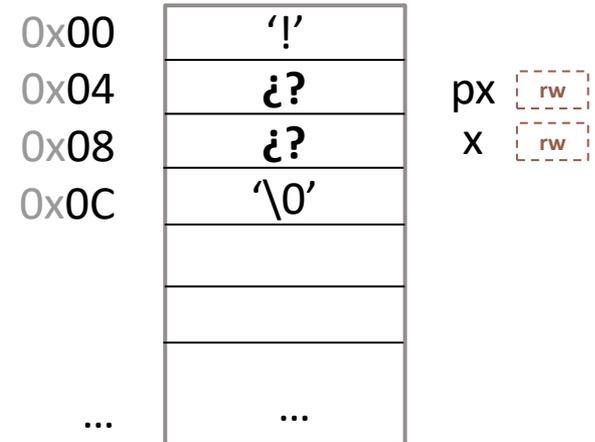
- ▣ **valor = mem_leer** (dirección)
- ▣ **mem_escribir** (dirección, valor)

NOTA: antes de acceder a una dirección, esta tiene que apuntar a una zona de memoria previamente reservada.

Uso de memoria en C: ejemplo 1

☆ Dos variables

- `int *px;`
- `int x;`



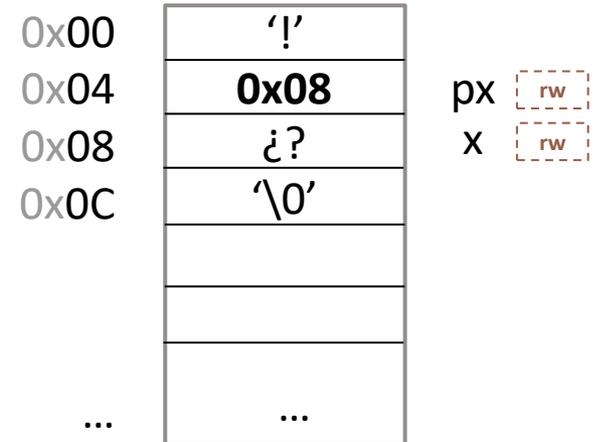
Uso de memoria en C: ejemplo 1



☆ Dos variables

- `int *px;`
- `int x;`

☆ `px = &x;`



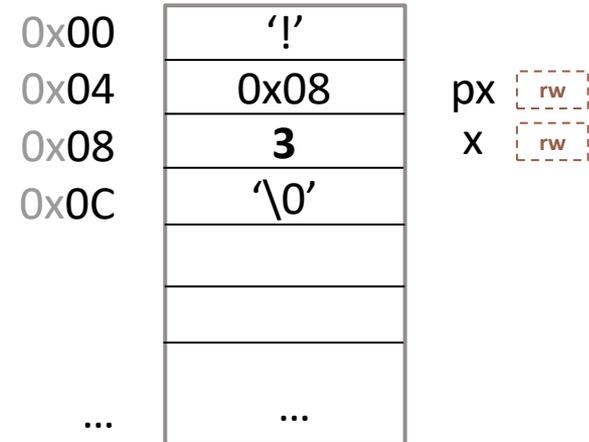
Uso de memoria en C: ejemplo 1



☆ Dos variables

- `int *px;`
- `int x;`

```
☆ px = &x;  
  *px = 3;
```



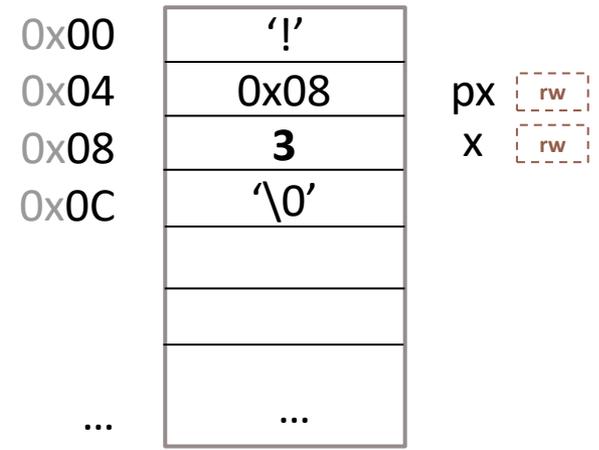
Uso de memoria en C: ejemplo 1



☆ Dos variables

- `int *px;`
- `int x;`

```
☆ px = &x;  
  *px = 3;  
  printf("x:%d\n", x) ;
```



mem_leer y mem_escribir en C: ejemplo 2



```
#include <stdio.h>

void imprimir ( int val ) {
    printf("v:%d\n", val);
}

int main ( int argc,
          char *argv[] )
{
    int i = 3;
    imprimir(i);
    return 0;
}
```

9 Lectura de variable

+ <var> => Valor de...

9 Escritura de variable:

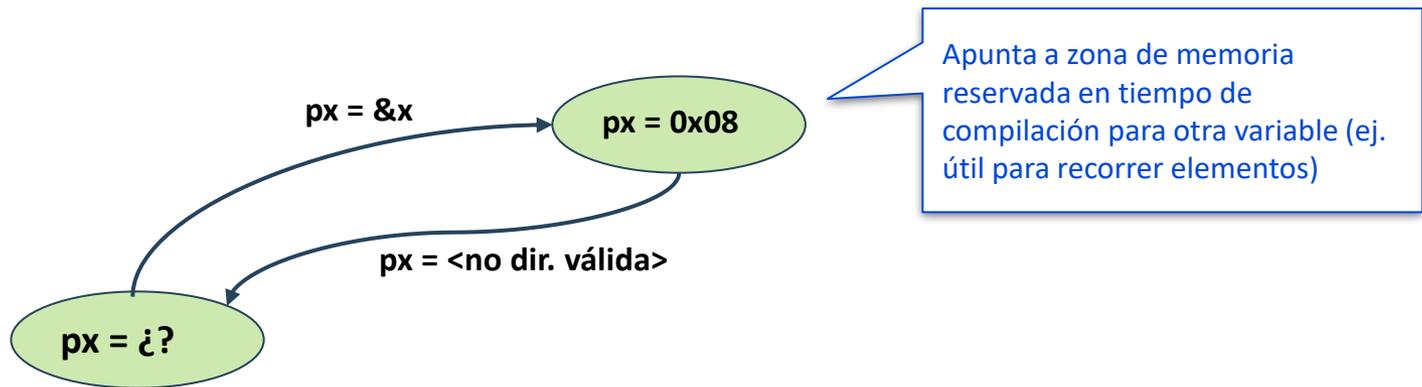
+ en **asignación**: **int i = 3 ;**

& Asignar valor...

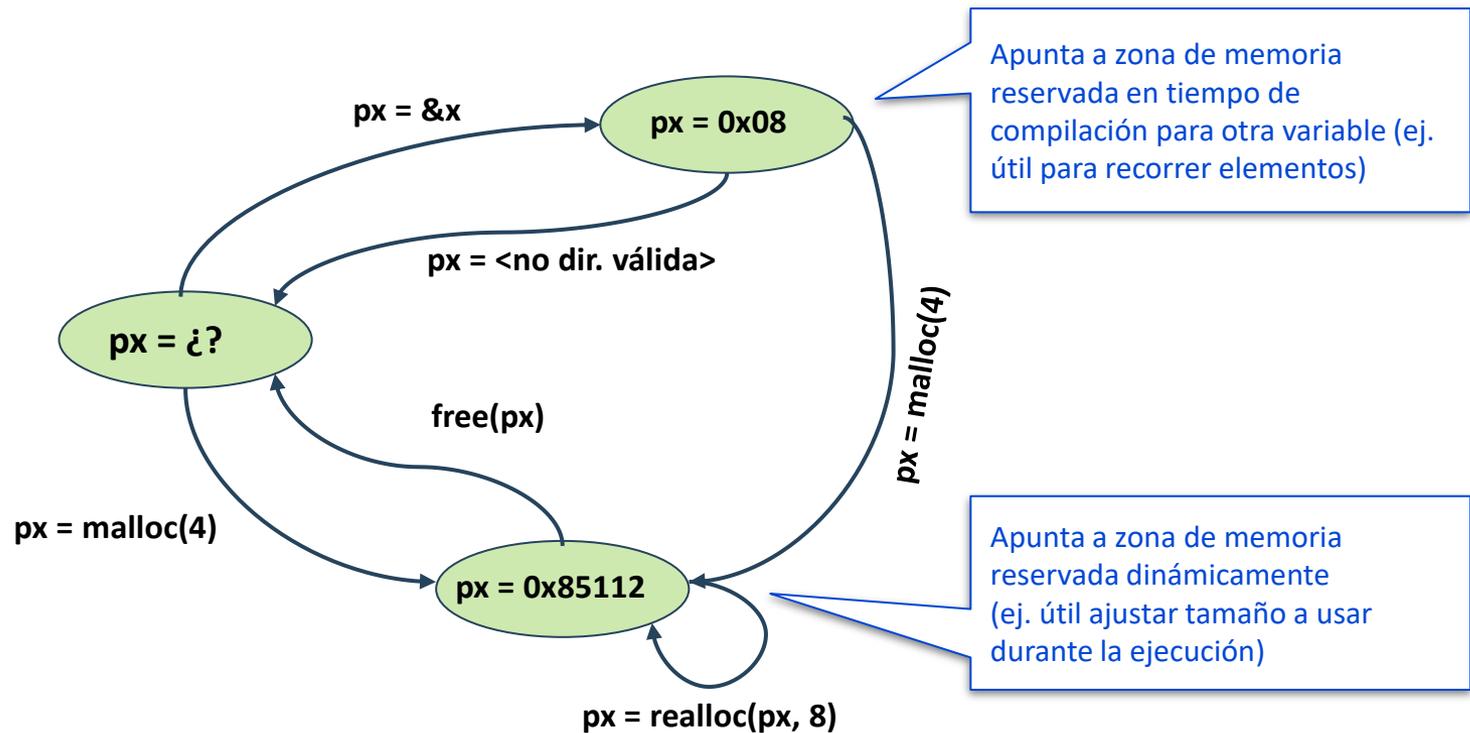
+ en **paso de parámetros**: <var>

& Asignar valor de...

Por defecto, TÚ tienes que llevar la gestión... máquina de estado finito para cada puntero



Por defecto, TÚ tienes que llevar la gestión... máquina de estado finito para cada puntero



Contabilidad errónea -> SIGSEGV



☆ Dos variables

- `int *px;`
- `int x;`

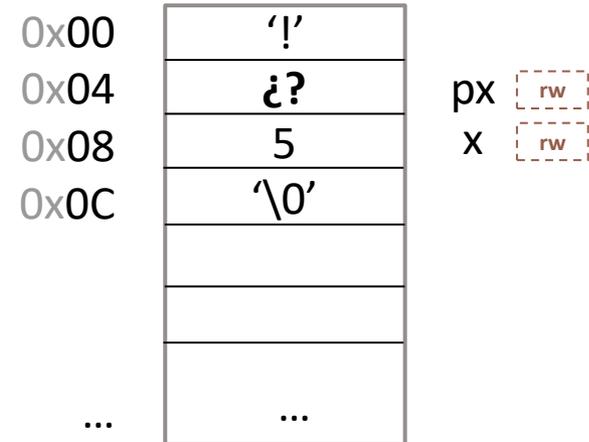
☆ `*px = x;`

- **MAL:** Si no se ha reservado memoria.

```
int *px;  
*px = 5;
```

- **BIEN:** Si se ha reservado previamente.

```
int *px;  
px = &x; // espacio x ya reservado  
*px = 5;
```



Contabilidad errónea -> SIGSEGV

☆ Gestor de memoria apropiado:

- ▣ libc malloc
- ▣ dlmalloc, jemalloc, hoard, etc.
- ▣ Electric Fence: https://elinux.org/Electric_Fence
<http://ithare.com/testing-memory-allocators-ptmalloc2-tcmalloc-hoard-jemalloc-while-trying-to-simulate-real-world-loads/>

☆ Herramientas de asistencia apropiadas:

- ▣ valgrind
- ▣ gdb, etc.

Valgrind: ejemplo típico de uso

133

<https://valgrind.org/docs/manual/quick-start.html#quick-start.interpret>



Félix García Carballera,
Alejandro Calderón Mateos

Here's an example C program, in a file called a.c, with a memory error and a memory leak.

```
#include <stdlib.h>

void f(void)
{
    int* x = malloc(10 * sizeof(int));
    x[10] = 0;          // problem 1: heap block overrun
}                    // problem 2: memory leak -- x not freed

int main(void)
{
    f();
    return 0;
}
```

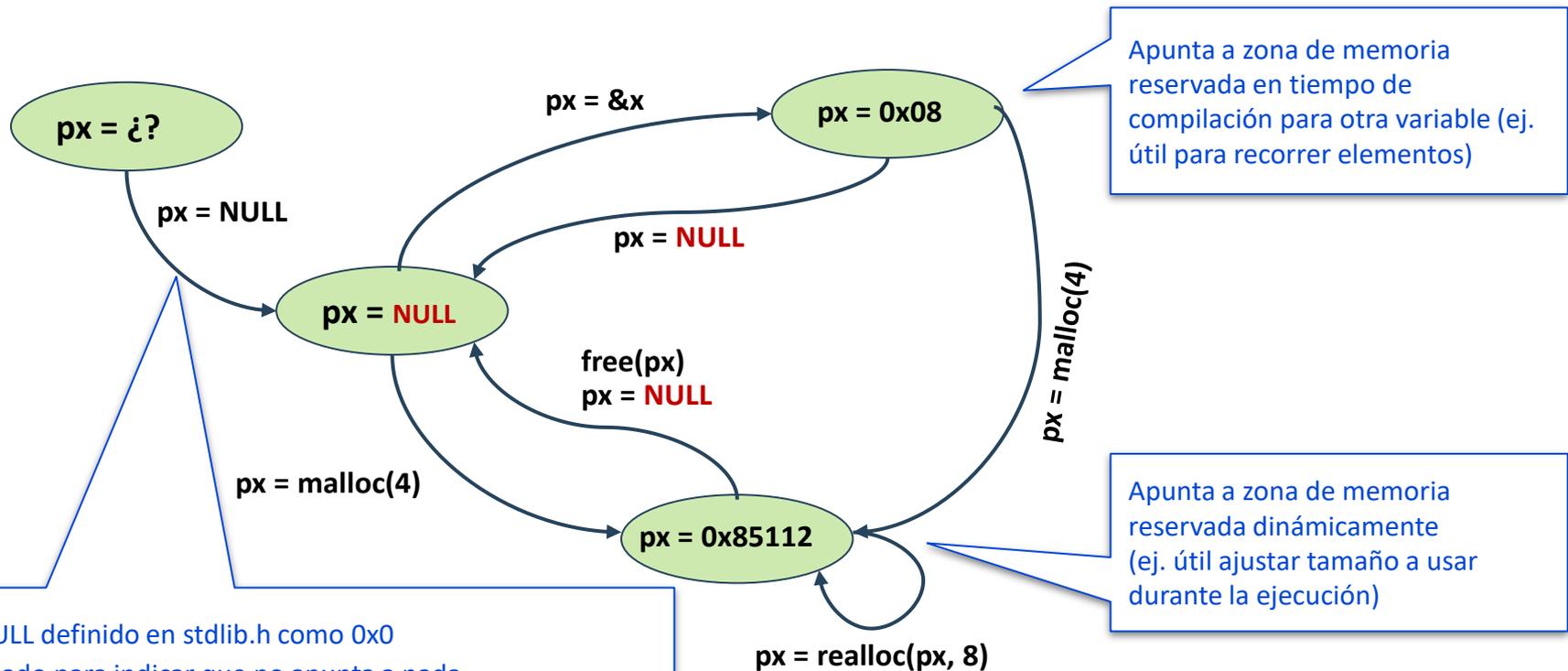
Most error messages look like the following, which describes problem 1, the heap block overrun:

```
==19182== Invalid write of size 4
==19182==    at 0x804838F: f (example.c:6)
==19182==    by 0x80483AB: main (example.c:11)
==19182== Address 0x1BA45050 is 0 bytes after a block of size 40 alloc'd
==19182==    at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==    by 0x8048385: f (example.c:5)
==19182==    by 0x80483AB: main (example.c:11)
```

Things to notice:

- There is a lot of information in each error message; read it carefully.

Por defecto, TÚ tienes que llevar la gestión... máquina de estado finito para cada puntero



- NULL definido en stdlib.h como 0x0
- Usado para indicar que no apunta a nada
- Buena práctica: inicializar toda variable puntero a NULL :
`int * p = NULL`
- Buena práctica: comprobar toda variable puntero:
`if (p == NULL) {...`

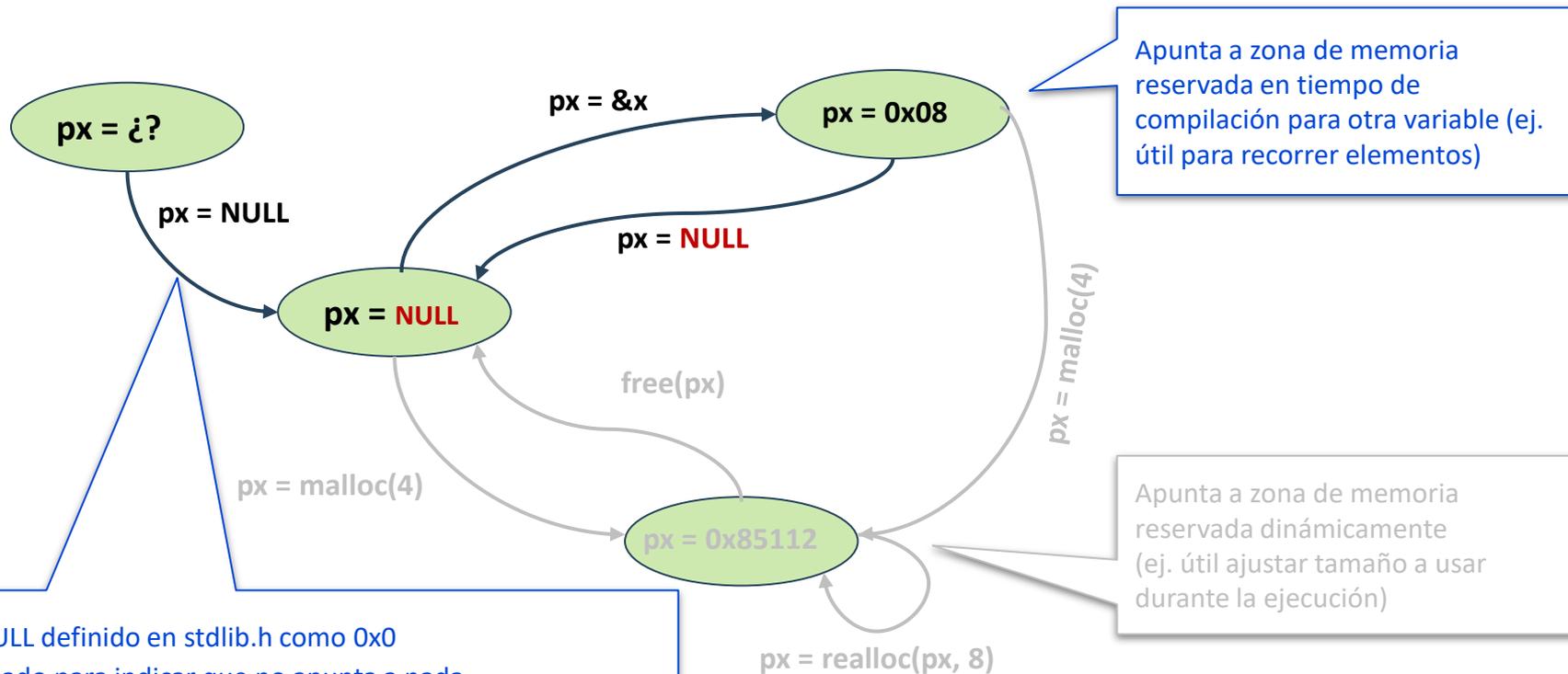
Apunta a zona de memoria reservada en tiempo de compilación para otra variable (ej. útil para recorrer elementos)

Apunta a zona de memoria reservada dinámicamente (ej. útil ajustar tamaño a usar durante la ejecución)

Contenidos

- Introducción a punteros
- **Casos de uso típicos:**
 - **Iterador: `for (int *p=v; p<&(v[10]); p++) ...`**
 - Memoria dinámica: `p = malloc(10); ...`
 - Paso de parámetros: `f1 (&a, &b) ;`
- Aritmética de punteros

Por defecto, TÚ tienes que llevar la gestión... máquina de estado finito para cada puntero



- NULL definido en stdlib.h como 0x0
- Usado para indicar que no apunta a nada
- Buena práctica: inicializar toda variable puntero a NULL :
`int * p = NULL`
- Buena práctica: comprobar toda variable puntero:
`if (p == NULL) {...`

Apunta a zona de memoria reservada en tiempo de compilación para otra variable (ej. útil para recorrer elementos)

Apunta a zona de memoria reservada dinámicamente (ej. útil ajustar tamaño a usar durante la ejecución)

dirección y contenido en C: array vs pointers

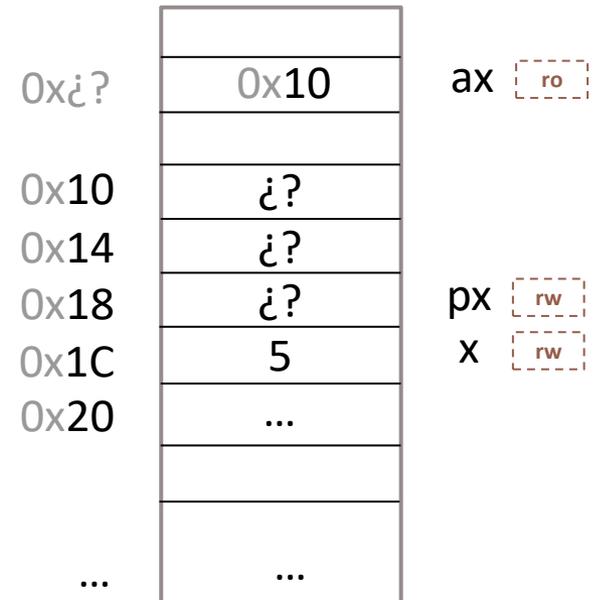
137



Félix García Carballeira,
Alejandro Calderón Mateos

☆ Tres variables

- `int ax[2];`
- `int *px;`
- `int x = 5;`



dirección y contenido en C: array vs pointers

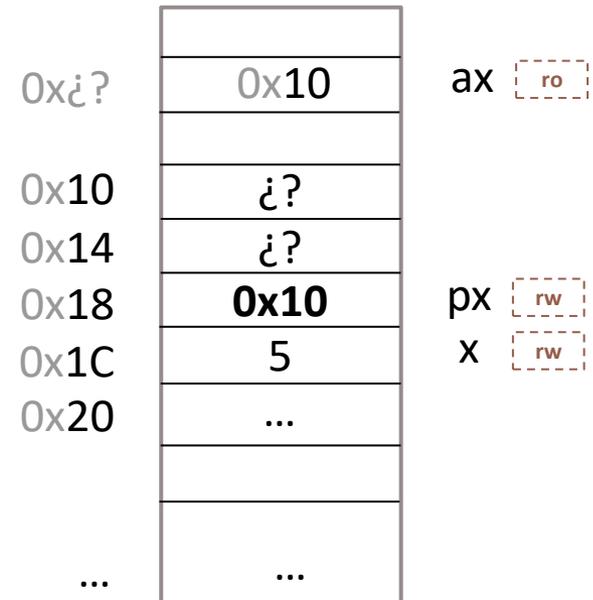
☆ Tres variables

- `int ax[2];`
- `int *px;`
- `int x = 5;`

☆ `px = ax; // ax == &ax`

☆ `ax = px; // ERROR`

```
☆ for (int i=0; i<2; i++) {  
    px[i] = i; // ptr como arr  
    ax[i] = i; // arr como arr  
    *(px+i) = i; // ptr como ptr  
    *(ax+i) = i; // arr como ptr  
}
```



dirección y contenido en C: array vs pointers

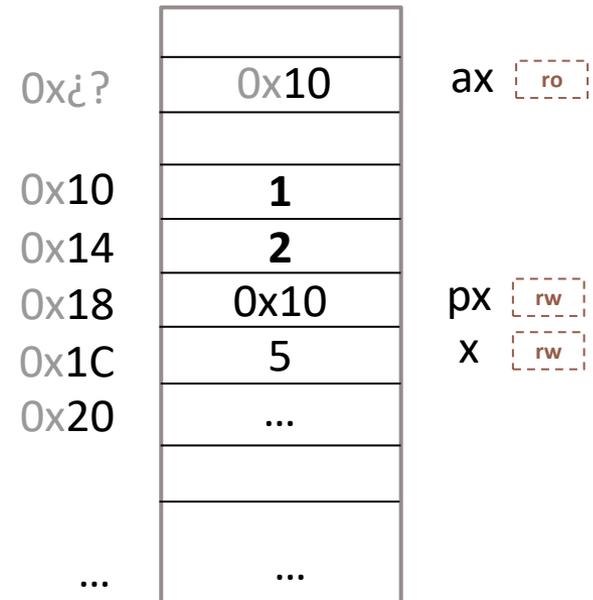
☆ Tres variables

- `int ax[2];`
- `int *px;`
- `int x = 5;`

☆ `px = ax; // ax == &ax`

☆ `ax = px; // ERROR`

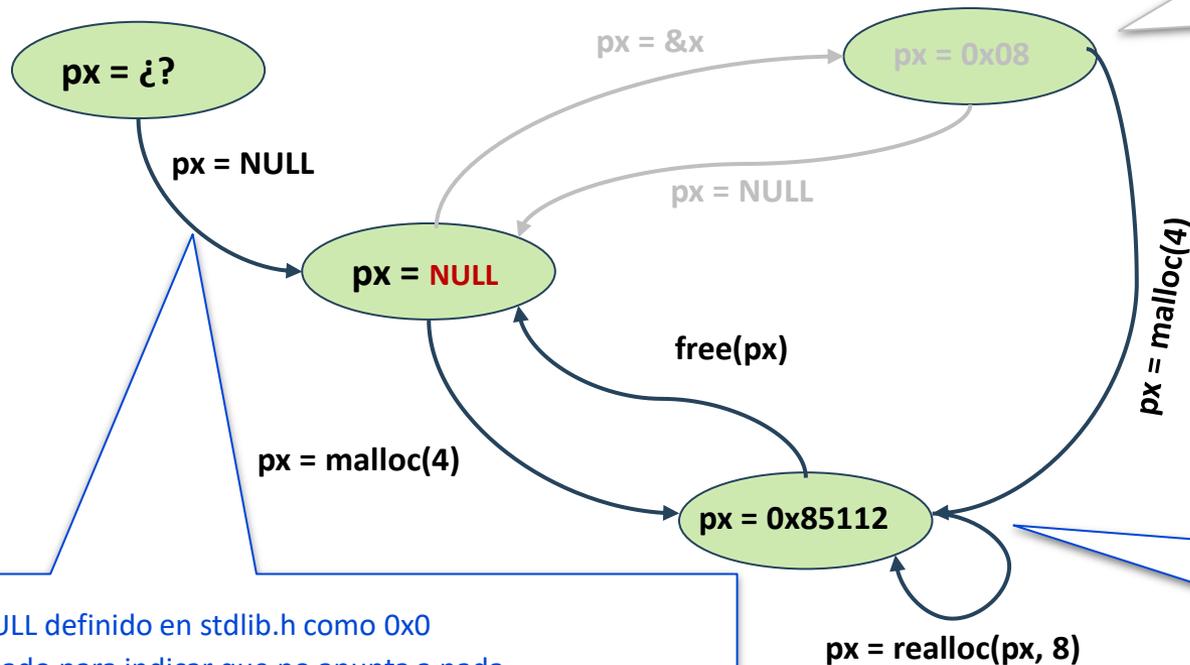
```
☆ for (int i=0; i<2; i++) {  
    px[i] = i+1; // ptr como arr  
    ax[i] = i+1; // arr como arr  
    *(px+i) = i+1; // ptr como ptr  
    *(ax+i) = i+1; // arr como ptr  
}
```



Contenidos

- Introducción a punteros
- **Casos de uso típicos:**
 - Iterador: `for (int *p=v; p<&(v[10]); p++) ...`
 - **Memoria dinámica: `p = malloc(10); ...`**
 - Paso de parámetros: `f1 (&a, &b) ;`
- Aritmética de punteros

Por defecto, TÚ tienes que llevar la gestión... máquina de estado finito para cada puntero



Apunta a zona de memoria reservada en tiempo de compilación para otra variable (ej. útil para recorrer elementos)

Apunta a zona de memoria reservada dinámicamente (ej. útil ajustar tamaño a usar durante la ejecución)

- NULL definido en `stdlib.h` como `0x0`
- Usado para indicar que no apunta a nada
- Buena práctica: inicializar toda variable puntero a NULL :
`int * p = NULL`
- Buena práctica: comprobar toda variable puntero:
`if (p == NULL) {...`

dirección y contenido en C: malloc (1/2)



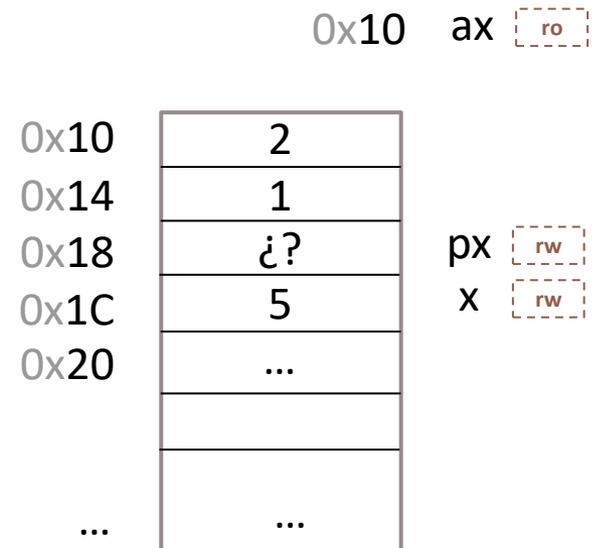
Félix García Carballera,
Alejandro Calderón Mateos

142

☆ Tres variables

- `int ax[2];`
- `int *px = NULL;`
- `int x = 5;`

- ☆ `px = (int *)malloc(4*sizeof(int));`
- ☆ `px[2] = 1; // *(px+2) = 1;`
- ☆ ...
- ☆ `free(px) // liberar NO ES AUTOMÁTICO`
- ☆ `px = NULL;`



dirección y contenido en C: malloc (1/2)

☆ Tres variables

- `int ax[2];`
- `int *px = NULL;`
- `int x = 5;`

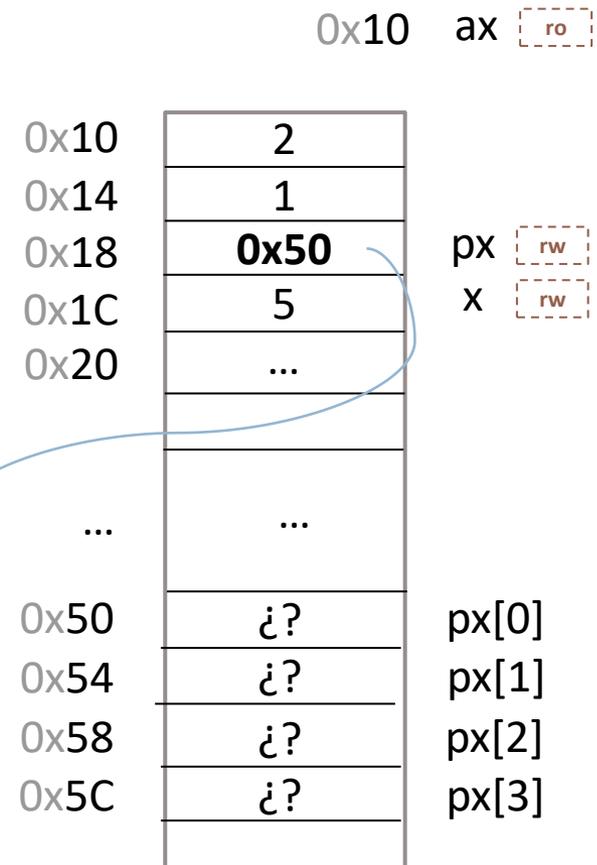
☆ `px = (int *)malloc(4*sizeof(int));`

☆ `px[2] = 1; // *(px+2) = 1;`

☆ ...

☆ `free(px) // liberar NO ES AUTOMÁTICO`

☆ `px = NULL;`

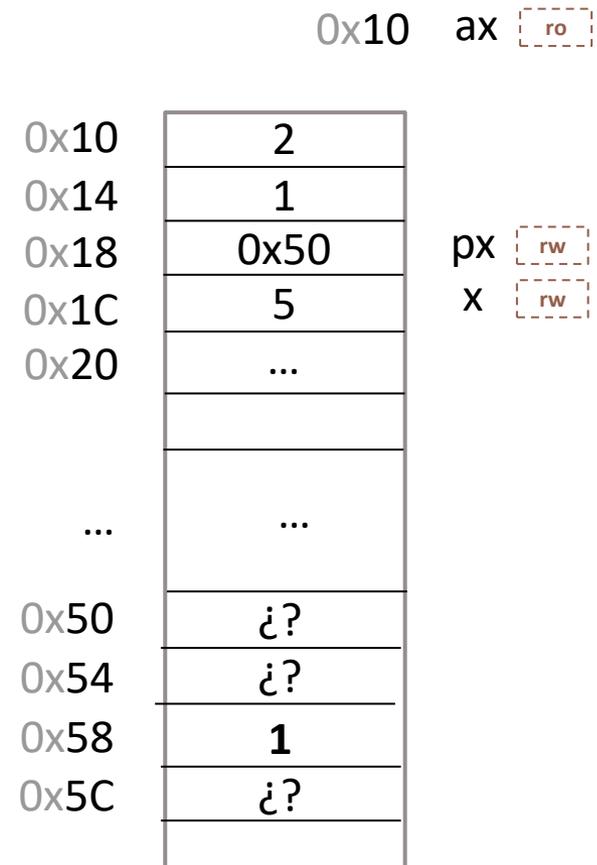


dirección y contenido en C: malloc (1/2)

☆ Tres variables

- `int ax[2];`
- `int *px = NULL;`
- `int x = 5;`

- ☆ `px = (int *)malloc(4*sizeof(int));`
- ☆ `px[2] = 1; // *(px+2) = 1;`
- ☆ `...`
- ☆ `free(px) // liberar NO ES AUTOMÁTICO`
- ☆ `px = NULL;`

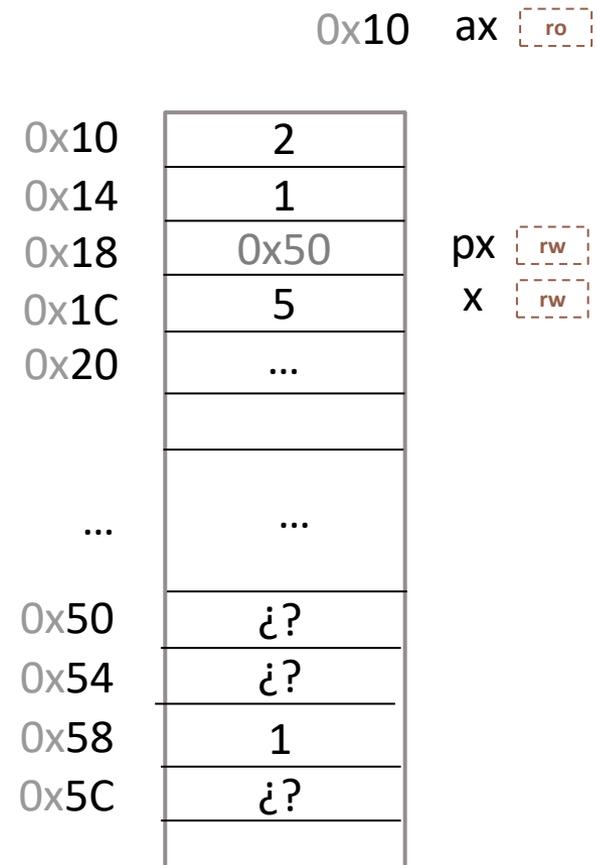


dirección y contenido en C: malloc (1/2)

☆ Tres variables

- `int ax[2];`
- `int *px = NULL;`
- `int x = 5;`

- ☆ `px = (int *)malloc(4*sizeof(int));`
- ☆ `px[2] = 1; // *(px+2) = 1;`
- ☆ `...`
- ☆ `free(px) // liberar NO ES AUTOMÁTICO`
- ☆ `px = NULL;`



dirección y contenido en C: malloc (2/2)



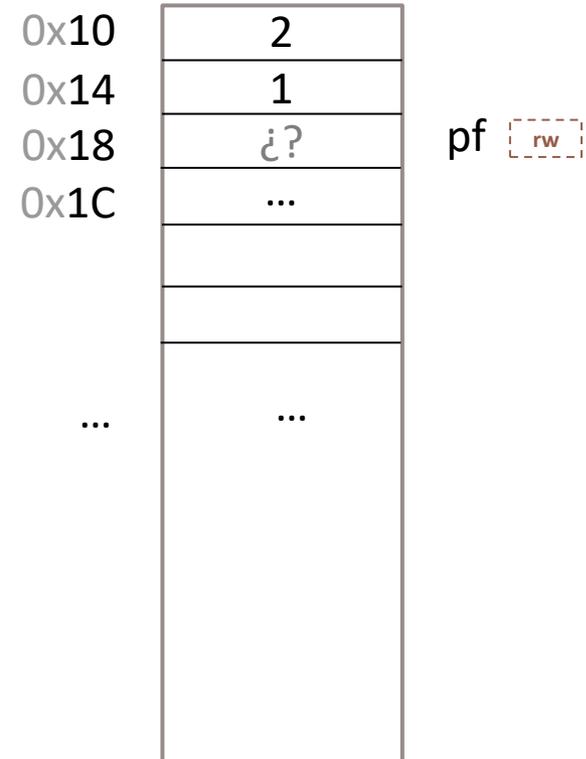
Félix García Carballera,
Alejandro Calderón Mateos

146

☆ Tres variables

```
• struct ficha {  
    int id;  
    char *nombre;  
};  
• struct ficha *pf = NULL;
```

```
☆ pf = malloc(sizeof(struct ficha));  
☆ (*pf).nombre = malloc(12*sizeof(char));  
☆ pf->id = 100 ;  
☆ strcpy(pf->nombre, "nombre");  
☆ ...  
☆ free(pf) // liberar NO ES AUTOMÁTICO  
☆ pf = NULL;
```



dirección y contenido en C: malloc (2/2)



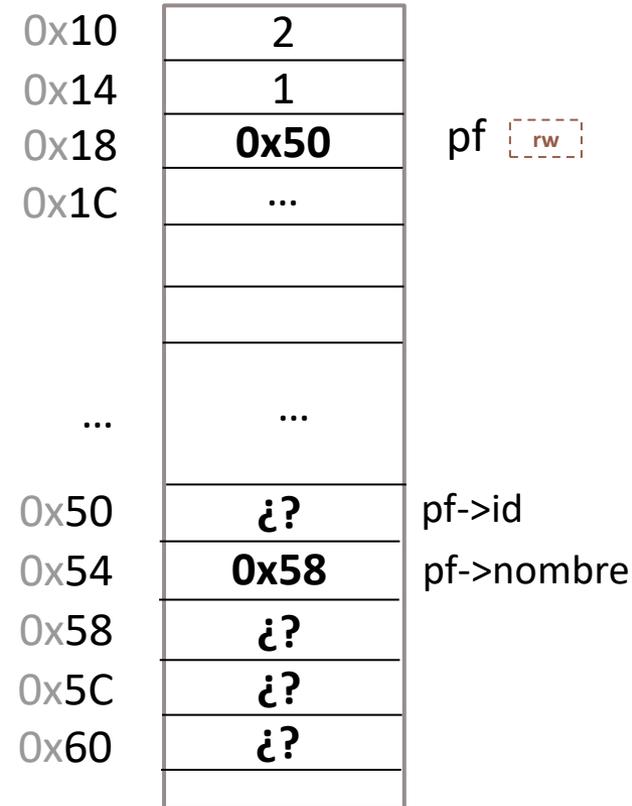
Félix García Carballera,
Alejandro Calderón Mateos

147

☆ Tres variables

```
• struct ficha {  
    int id;  
    char *nombre;  
};  
• struct ficha *pf = NULL;
```

```
☆ pf = malloc(sizeof(struct ficha));  
☆ (*pf).nombre = malloc(12*sizeof(char));  
☆ pf->id = 100 ;  
☆ strcpy(pf->nombre, "nombre");  
☆ ...  
☆ free(pf) // liberar NO ES AUTOMÁTICO  
☆ pf = NULL;
```



dirección y contenido en C: malloc (2/2)



Félix García Carballera,
Alejandro Calderón Mateos

148

☆ Tres variables

```
• struct ficha {  
    int id;  
    char *nombre;  
};  
• struct ficha *pf = NULL;
```

```
☆ pf = malloc(sizeof(struct ficha));  
☆ (*pf).nombre = malloc(12*sizeof(char));  
☆ pf->id = 100 ;  
☆ strcpy(pf->nombre, "nombre");  
☆ ...  
☆ free(pf) // liberar NO ES AUTOMÁTICO  
☆ pf = NULL;
```

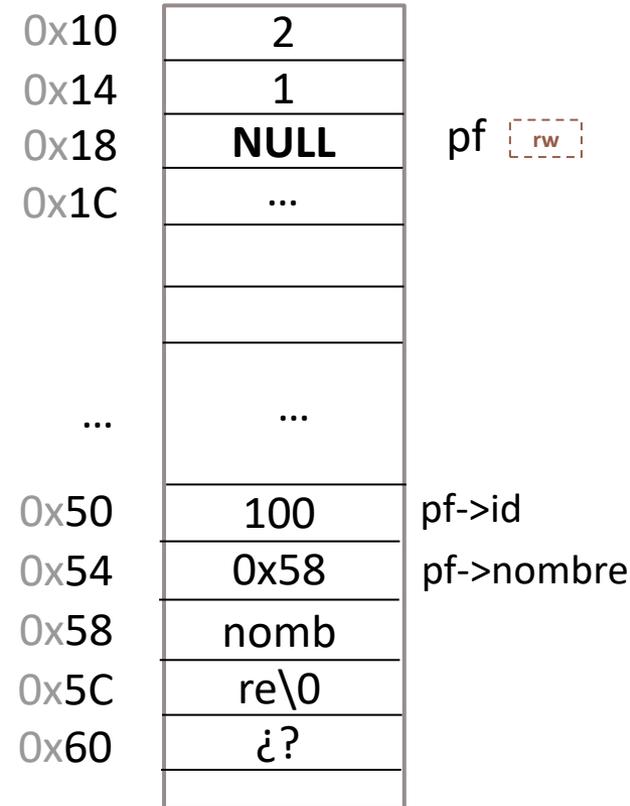
0x10	2	pf rw
0x14	1	
0x18	0x50	
0x1C	...	
...	...	
0x50	100	pf->id
0x54	0x58	pf->nombre
0x58	nomb	
0x5C	re\0	
0x60	¿?	

dirección y contenido en C: malloc (2/2)

☆ Tres variables

```
• struct ficha {  
    int id;  
    char *nombre;  
};  
• struct ficha *pf = NULL;
```

```
☆ pf = malloc(sizeof(struct ficha));  
☆ (*pf).nombre = malloc(12*sizeof(char));  
☆ pf->id = 100 ;  
☆ strcpy(pf->nombre, "nombre");  
☆ ...  
☆ free(pf) // liberar NO ES AUTOMÁTICO  
☆ pf = NULL;
```



Contenidos

- Introducción a punteros
- **Casos de uso típicos:**
 - Iterador: `for (int *p=v; p<&(v[10]); p++) ...`
 - Memoria dinámica: `p = malloc(10); ...`
 - **Paso de parámetros: `f1(&a, &b) ;`**
- Aritmética de punteros

Paso de parámetros

```
i = 10 ;  
float PI = 3.14 ;  
  
prueba2 ( i+2, 'a', PI, &i ) ;  
...
```

```
void prueba2 ( int j, char c, float f, int pj )  
{  
    /* ... */  
}
```

Paso de parámetros

```
i = 10 ;  
float PI = 3.14 ;  
  
prueba2 ( i+2, 'a', PI, &i ) ;  
...
```

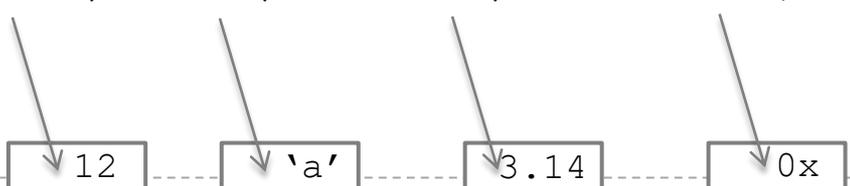
```
void prueba2 ( int j, char c, float f, int pj )  
{  
    /* ... */  
}
```

1) Se crea en pila las variables formales

Paso de parámetros

```
i = 10 ;  
float PI = 3.14 ;
```

```
prueba2 ( i+2, 'a', PI, &i ) ;  
...
```



```
void prueba2 ( int j, char c, float f, int pj )  
{  
    /* ... */  
}
```

2) Se copia el valor de los parámetros reales

Paso de parámetros

154

Félix García Carballeira,
Iderón Mateos

Siempre se realiza una copia de los parámetros

```
i = 10 ;  
float PI = 3.14 ;
```

```
prueba2 ( i+2, 'a', PI, &i ) ;  
...
```

12 'a' 3.14 0x

```
void prueba2 ( int j, char c, float f, int pj )  
{  
    /* ... */  
}
```

Paso de parámetros



Paso de parámetro por valor

```
#include <stdio.h>

int main (void)
{
    int i = 10;

    /* ... */
    inc(i) ;
    /* ... */
}
```

1) se copia
i en j

```
void inc ( int j )
{
    j = j + 1 ;
}
```

2) se modifica
j (la copia)

Paso de parámetro por referencia

```
#include <stdio.h>

int main (void)
{
    int i = 3;

    /* ... */
    inc(&i) ;
    /* ... */
}
```

```
void inc ( int *j )
{
    *j = *j + 1 ;
}
```

Paso de parámetros



Paso de parámetro por valor

```
#include <stdio.h>

int main (void)
{
    int i = 10;

    /* ... */
    inc(i);
    /* ... */
}
```

1) se copia i en j

```
void inc ( int j )
{
    j = j + 1 ;
}
```

2) se modifica j (la copia)

Paso de parámetro por referencia

```
#include <stdio.h>

int main (void)
{
    int i = 3;

    /* ... */
    inc(&i);
    /* ... */
}
```

1) se copia &i en j

```
void inc ( int *j )
{
    *j = *j + 1 ;
}
```

2) se modifica i a través de *j

Paso de parámetros por referencia uso aunque no se modifiquen valores



El array está guardado en memoria (consigna) y pasamos la referencia (llave) para llevar menos peso (evitar copia de datos en paso por valor).

Paso de parámetro por **referencia**

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

```
int main (void)
```

```
{
```

```
    int ai[]={1,2,3,4,5};
```

```
    int x=asuma(5, &ai);
```

```
}
```

1) se copia
&ai en arr

```
int asuma(int n, int *arr)
{
    int r=0;
    for (int i=0; i<n; i++)
        r = r + arr[i];
}
```

2) se accede a
través de arr

Paso de parámetros: ejemplo

```
#include <stdio.h>

void inc ( int *j )
{
    *j = *j + 1 ;
}

int main (void)
{
    int i = 3;

    inc(&i) ;
    printf("%d\n",i) ;

    return 0;
}
```

- La función *inc* incrementa el valor pasado por referencia en *j*
- La función *main* define una variable *i*, incrementa su valor y lo imprime

Paso de parámetros: ejemplo

```
#include <stdio.h>

void inc ( int *j )
{
    *j = *j + 1 ;
}

int main (void)
{
    int i = 3;

    inc(&i) ;
    printf("%d\n",i) ;

    return 0;
}
```

- **gcc -Wall -g -o e2 e2.c**
 - -Wall:
mostrar todas las advertencias
 - -g:
añadir información de depuración
 - -o:
establecer el nombre del ejecutable

- **./e2**
 - El directorio actual (.) no está en la variable PATH

Paso de parámetros: ejemplo



```
#include <stdio.h>

void inc ( int *j )
{
    *j = *j + 1 ;
}

int main (void)
{
    int i = 3;

    ! inc( i) ;
    ● printf("%d\n",i) ;

    return 0;
}
```

- **gcc -Wall -g -o e3 e3.c**
 - -Wall:
mostrar todas las advertencias
 - -g:
añadir información de depuración
 - -o:
establecer el nombre del ejecutable

- **./e2**
 - El directorio actual (.) no está en la variable PATH

Paso de parámetros: ejemplo

```
Violación de segmento
acaldero@phoenix:/tmp$ gdb e13
GNU gdb (Ubuntu/Linaro 7.2-1ubuntu11) 7.2
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-linux-gnu".
Para las instrucciones de informe de errores, vea:
<http://www.gnu.org/software/gdb/bugs/>...
Leyendo símbolos desde /tmp/e13...hecho.
(gdb) run
Starting program: /tmp/e13

Program received signal SIGSEGV, Segmentation fault.
0x080483ca in inc (j=0x3) at e13.c:5
5          *j = *j + 1 ;
(gdb) 
```

Contenidos

- Introducción a punteros
- Casos de uso típicos:
 - Iterador: `for (int *p=v; p<&(v[10]); p++) ...`
 - Memoria dinámica: `p = malloc(10); ...`
 - Paso de parámetros: `f1 (&a, &b) ;`
- **Aritmética de punteros**

Aritmética de punteros...



163

<https://pixabay.com/photos/concepts-journey-outdoor-barefoot-316725/>



Félix García Carballeira,
Alejandro Calderón Mateos

- Aritmética de punteros con entero permite saltar elementos (entero multiplicado por sizeof(tipo base))

★ Tres variables

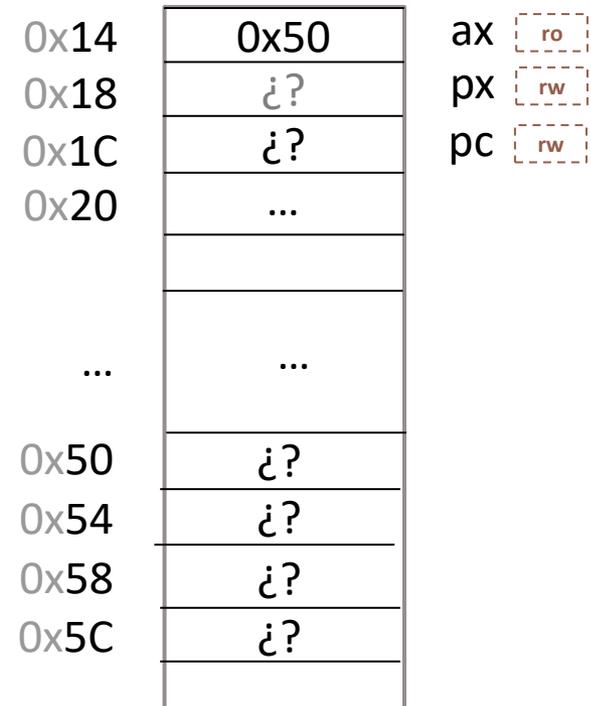
- `int ax[4];`
- `int *px;`
- `char *pc;`

★ `px = &(ax[3]);`

★ `px-- ; // apunta al anterior entero`

★ `pc = &ax; // px = &(ax[0])`

★ `pc++ ; // apunta al siguiente char`



Puntero a void

- Puntero genérico:
 - `void * p = NULL ;`

- Permite casting implícito/explicito desde “puntero a x”:
 - `void *p ;`
 - `int *pi ; char *pc ;`
 - `p = pi ; p = pc ;`

- Permite casting explícito a “puntero a x”:
 - `pi = (int *)p ;`

- NO permite des-referenciar:
 - `*p = 3 ;`

Contenidos

- **Introducción al lenguaje C**
 - Preprocesador
 - Comentarios
 - Tipos de datos básicos, variables y constantes
 - Asignación y conversión de tipos (casting)
 - Tipos compuestos: array y struct
 - Definición de tipos
 - Sentencias de control
 - Funciones
 - Punteros
- **Aspectos de interés sobre estructuras de datos**

sizeof(struct)

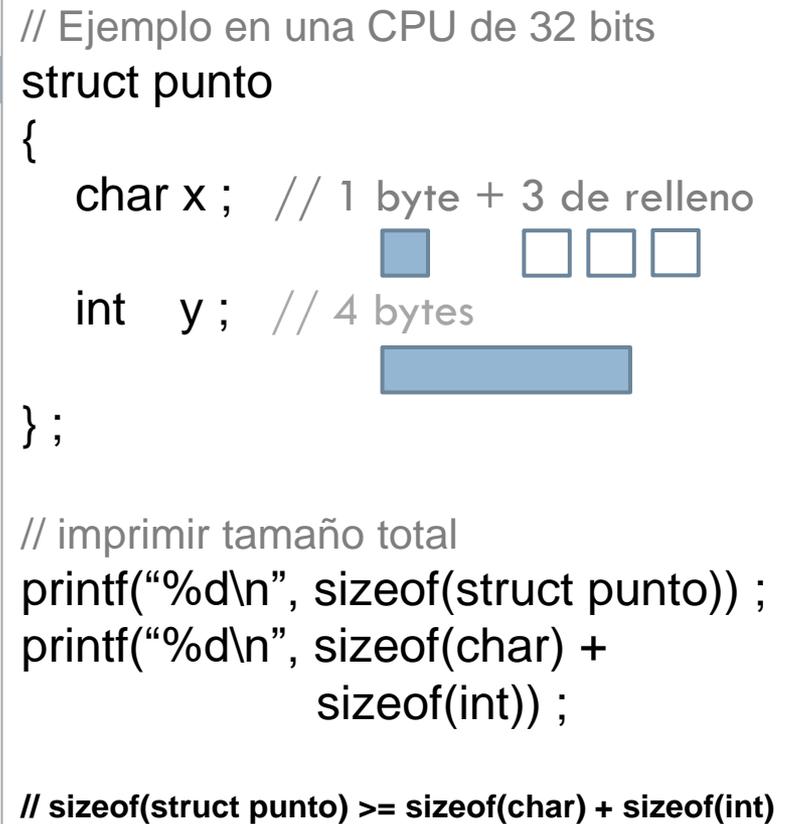
166

- El tamaño de un struct puede ser mayor que la suma del tamaño de sus campos.
- ▣ Puede ser debido a efecto de relleno (*padding*) para alinear datos en memoria.

```
// Ejemplo en una CPU de 32 bits
struct punto
{
    char x ; // 1 byte + 3 de relleno
    int  y ; // 4 bytes
};

// imprimir tamaño total
printf(“%d\n”, sizeof(struct punto)) ;
printf(“%d\n”, sizeof(char) +
        sizeof(int)) ;

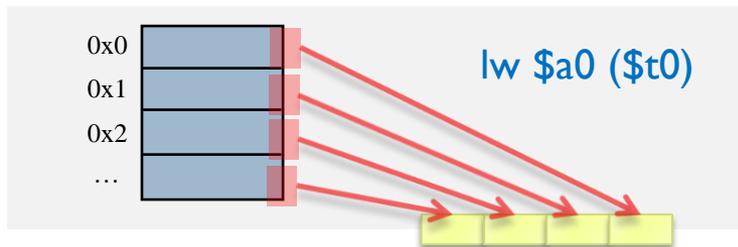
// sizeof(struct punto) >= sizeof(char) + sizeof(int)
```



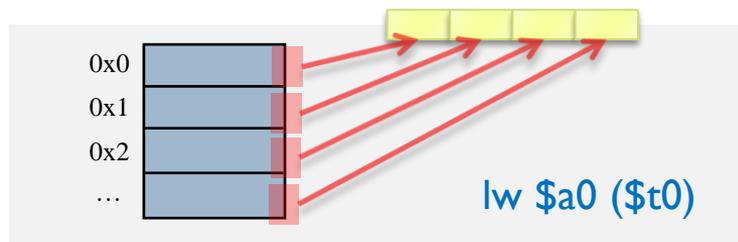
Orden de bytes

▶ Hay dos tipos de ordenamiento de bytes:

▶ Little-endian (Dirección 'pequeña' termina la palabra...)



▶ Big-endian (Dirección 'grande' termina la palabra...)



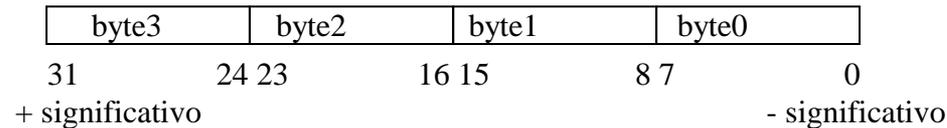
IBM
PowerPC™ (bi-endian)



Orden de bytes

- ▶ Hay dos tipos de ordenamiento de bytes:

Palabra de 32 bits



A	byte3
A+1	byte2
A+2	byte1
A+3	byte0

BigEndian

A	byte0
A+1	byte1
A+2	byte2
A+3	byte3

LittleEndian

A	00000000
A+1	00000000
A+2	00000000
A+3	00011011

BigEndian

A	00011011
A+1	00000000
A+2	00000000
A+3	00000000

LittleEndian

Orden de bytes

169



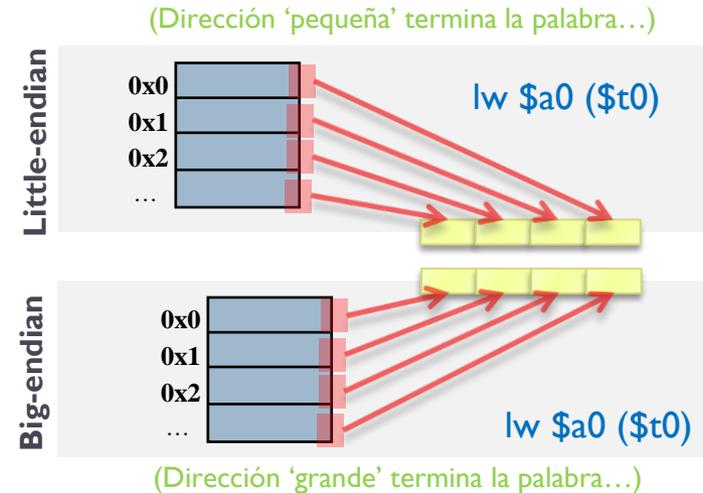
Félix García Carballeira,
Alejandro Calderón Mateos

```
#include <stdio.h>
#include <stdbool.h>
#include <math.h>
```

```
int main()
{
    int x = 4 ;
    unsigned char *p ;
```

```
    p = (unsigned char*) &x ;
    printf("En dirección %p se almacena %d\n", &x, x) ;
    printf("En dirección %p se almacena %d\n", p, *p) ;
    printf("En dirección %p se almacena %d\n", p+1, *(p+1)) ;
    printf("En dirección %p se almacena %d\n", p+2, *(p+2)) ;
    printf("En dirección %p se almacena %d\n", p+3, *(p+3)) ;
```

```
}
```



Orden de bytes

170



Félix García Carballeira,
Alejandro Calderón Mateos

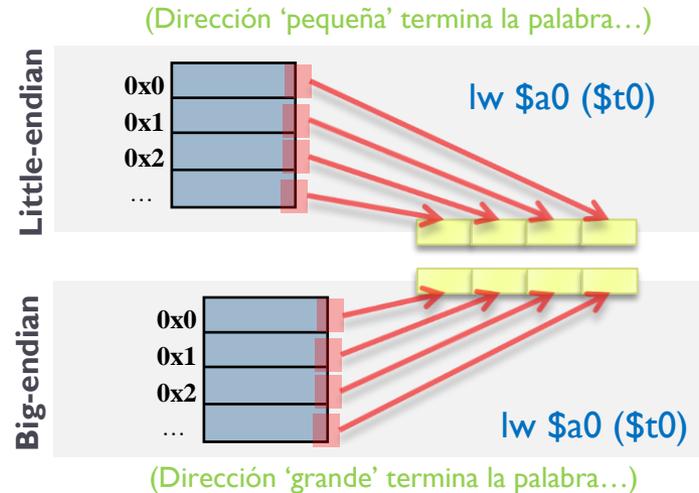
```
#include <stdio.h>
#include <stdbool.h>
#include <math.h>
```

```
int main()
{
    int x = 4 ;
    unsigned char *p ;
```

```
    p = (unsigned char*) &x ;
```

```
En dirección 0x7ffc30babdbc se almacena 4
En dirección 0x7ffc30babdbc se almacena 4
En dirección 0x7ffc30babdbd se almacena 0
En dirección 0x7ffc30babdbe se almacena 0
En dirección 0x7ffc30babdbf se almacena 0
```

```
&x, x) ;
p, *p) ;
p+1, *(p+1)) ;
p+2, *(p+2)) ;
p+3, *(p+3)) ;
```



Orden de bytes

- ▶ Problemas en la comunicación entre computadores con arquitectura distinta:

El número $27_{(10)} = 11011_{(2)} = 00000000000000000000000000011011$



A	00000000
A+1	00000000
A+2	00000000
A+3	00011011

BigEndian

A	
A+1	
A+2	
A+3	

LittleEndian

Orden de bytes

- Problemas en la comunicación entre computadores con arquitectura distinta:

El número $27_{(10)} = 11011_{(2)} = 00000000000000000000000000011011$

A	00000000
A+1	00000000
A+2	00000000
A+3	00011011

BigEndian



Transmisión de
datos por la red

A	
A+1	
A+2	
A+3	

LittleEndian

Orden de bytes

- ▶ Problemas en la comunicación entre computadores con arquitectura distinta:

El número $27_{(10)} = 11011_{(2)} = 00000000000000000000000000011011$

A	00000000
A+1	00000000
A+2	00000000
A+3	00011011

BigEndian

A	00000000
A+1	00000000
A+2	00000000
A+3	00011011

LittleEndian



El número almacenado es: **00011011**000000000000000000000000 ; que no es el 27!

Orden de bytes

- ▶ Problemas en la comunicación entre computadores con arquitectura distinta:

El número $27_{(10)} = 11011_{(2)} = 00000000000000000000000000011011$





INTRODUCCIÓN AL LENGUAJE C

ARCOS . INF . UC3M . ES

FÉLIX GARCÍA CARBALLEIRA,
ALEJANDRO CALDERÓN MATEOS