

# Lenguaje C

© 2014 José Miguel Santos, Alexis Quesada,  
Fran Santana, Eduardo Rodríguez

# Historia del C

- Creado en 1972 por Brian Kernighan y Dennis Ritchie, dentro del proyecto UNIX.
- Se propagó rápidamente como lenguaje de programación de sistemas.
- Estándar ANSI83.
- Nuevo estándar ISO C99 (último).
- Muchos lenguajes derivados: C++, Objective-C, Java, C#, etc.

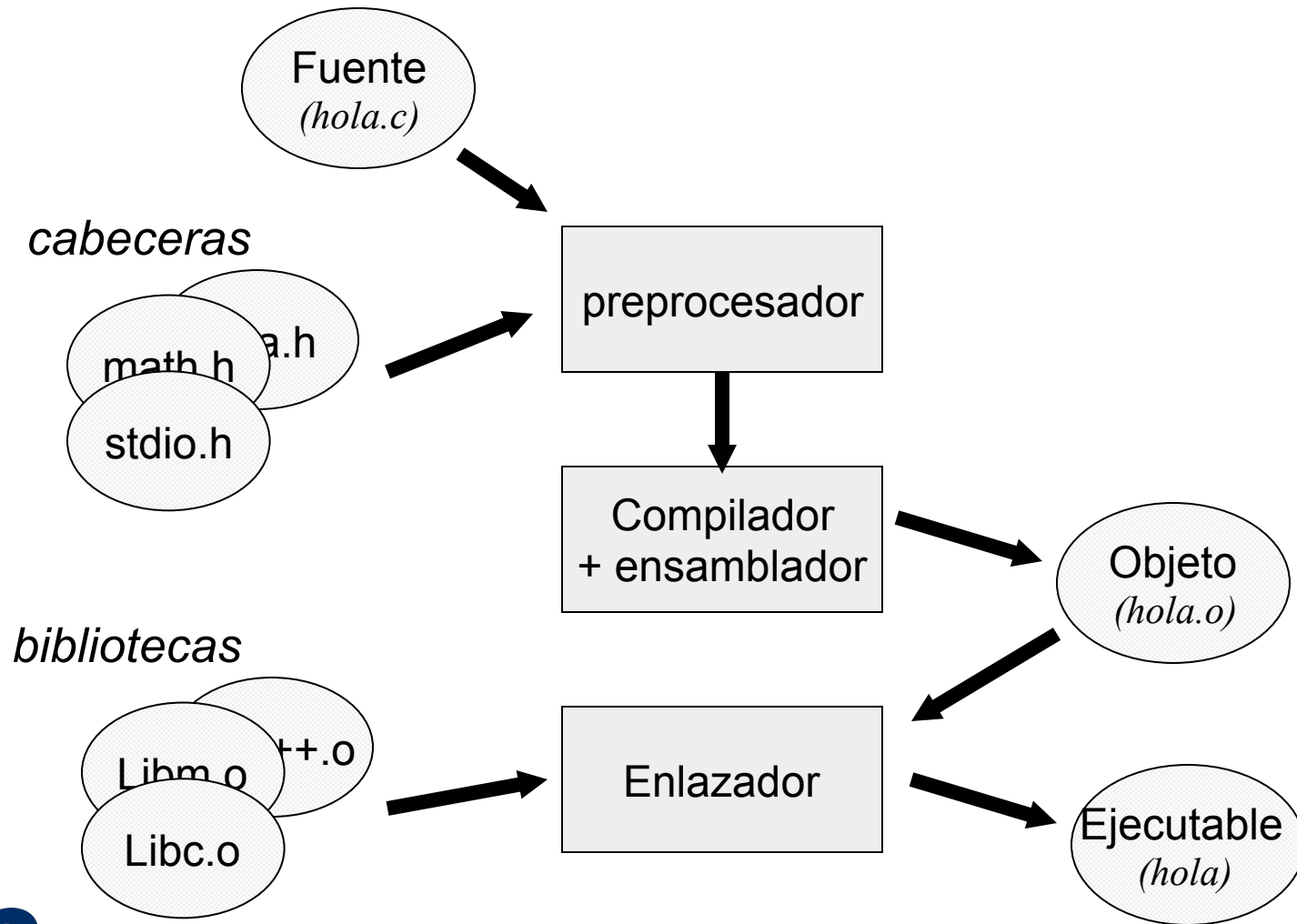
# Características del lenguaje

- Lenguaje de programación estructurada.
- Extremadamente simple.
- Permite generar código pequeño y eficiente.
- Poca comprobación de errores en el compilador (ej. tipado débil, punteros).
- Manejo de memoria a cargo del programador.
- Da mucha libertad al programador.

# Programa simple

```
#include <stdio.h>
main()
{
    /* un comentario
       de varias líneas
    */
    printf ("hola, mundo\n");
    // un comentario de una línea
}
```

# Modelo de compilación en C



# Cómo compilar (Linux)

- **gcc programa.c**  
*genera un ejecutable llamado «a.out»*
- **gcc programa.c -o programa**  
*genera un ejecutable llamado «programa»*
- **gcc xxx.c yyy.c zzz.c -o programa**  
*compila un programa a partir de varios fuentes*

# El preprocesador

- El **preprocesador** transforma el programa fuente de esta forma:
  - elimina los comentarios
  - incluye en el fuente el contenido de los ficheros declarados con **#include <fichero.h>**  
(a estos ficheros se les llama **cabeceras**)
  - sustituye en el fuente las macros declaradas con **#define** (ej. **#define CIEN 100**)

# El compilador

- El **compilador** convierte el fuente en un archivo en lenguaje máquina: **fichero objeto**.
- **gcc -c fichero.c**  
*genera el fichero objeto «fichero.o»*
- Algunos compiladores pasan por una fase intermedia en lenguaje ensamblador.
- **gcc -S fichero.c**  
*genera un fuente en ensamblador «fichero.s»*



# El enlazador (*linker*)

- El **enlazador** genera el ejecutable binario, a partir del contenido de los ficheros objetos y de las **bibliotecas**.
- Las bibliotecas contienen el código de funciones precompiladas, a las que el archivo fuente llama.
- **gcc xxx.o yyy.o zzz.o -lm -lpthread -o hola**  
*enlaza varios objetos \*.o, la biblioteca matemática y la biblioteca de hilos y genera el ejecutable «hola»*

# Bibliotecas estándares

- El estándar de C define bibliotecas para operaciones básicas y comunes. Ejemplos:
  - `<stdio.h>` entrada/salida: `printf`, `scanf`, `fopen`, `fclose`...
  - `<string.h>` cadenas: `strcpy`, `strcat`, `strlen`...
  - `<stdlib.h>` memoria: `malloc`, `free`, `memcpy`...
- Para usar las funciones de una biblioteca, se usa **`#include <fichero.h>`**

# Estructura del programa

- Ésta es la estructura habitual de un programa en C:
  - inclusión de ficheros cabeceras
  - declaraciones de tipos, variables y funciones
  - *definiciones* de funciones
- Todo ejecutable debe tener una función **main()**.

# Elementos de programación

- Estructuras de datos:
  - literales, tipos básicos, tipos enumerados
  - tipos estructurados (struct, union)
  - punteros y vectores
- Construcciones algorítmicas:
  - construcciones condicionales (if,switch)
  - construcciones iterativas (while,for,do...while)
  - subrutinas (funciones)

# Literales

<b>Nombre</b>	<b>Descripción</b>	<b>Ejemplos</b>
Decimal	entero en base 10	<b>1234</b>
Hexadecimal	entero en base 16	<b>0x1234</b>
Octal	entero en base 8	<b>01234</b>
Carácter	byte en ASCII	<b>'A'</b>
Coma flotante	número real en c.f.	<b>1.25</b>
		<b>3.456e6</b>
		<b>3.456e-6</b>
Cadena	texto literal	<b>"hola, mundo"</b>

# Tipos básicos

- Sólo tipos numéricos:
  - **int**
  - **char**
  - **float**
  - **double**
- No hay tipo booleano, ni cadenas (*strings*)
- Modificadores:
  - **signed**
  - **unsigned**
  - **short**
  - **long**
  - **long long**

# Declaraciones de variables

*tipo nombre {, nombre} ;*

*tipo nombre=valor\_inicial;*

¡Ojo!, distingue mayúsculas.

Ejemplos:

```
int una=1, otra, Una, UNA=3;
```

```
long entero_largo;
```

```
unsigned char letra='A' ;
```

# Expresiones

- Una expresión es una operación sobre literales o variables, que devuelve un resultado.
- Para construir expresiones se emplean **operadores** (+,-, etc.), paréntesis, etc.



# Operadores

- Aritméticos:  $+$   $-$   $*$   $/$   $\%$
- Manip. de bits:  $|$   $\&$   $^$   $\sim$   $<<$   $>>$
- Lógicos:  $==$   $!=$   $>$   $<$   $>=$   $<=$   $||$   $\&\&$   $!$
- Pre-post in/decremento:  $x++$ ,  $x--$ ,  $++x$ ,  $--x$
- Asignación compuesta:
  - $a+=3$  es como  $a=a+3$
  - $a*=x+1$  es como  $a=a*(x+1)$

# Expresiones: ojo

- No se detectan desbordamientos.
- En las divisiones, si los operandos son enteros, se devuelve sólo el cociente (sin decimales):  
*7/3 devuelve un 2*  
*(float)7/2 devuelve un 2.3333*
- Las operaciones booleanas devuelven un cero o un uno (en C no existe tipo booleano):  
*(3>2) devuelve un 1*  
*(3==2) devuelve un 0*

# Asignaciones

- La asignación es **a=b**
- **OJO:** la comparación de igualdad es **a==b**
- Las asignaciones son expresiones.
  - Se pueden usar dentro de expresiones más complejas. En este caso, devuelven el valor del lado derecho.

$$C = 20 - (B=2*(A=5)+4)$$

*A valdrá 5; B valdrá  $2*5+4=14$ ; C valdrá  $20-14=6$*

# Pre/post incremento/decremento

- Las expresiones con pre in/decremento devuelven el valor nuevo de la variable:

`x=1;`

`a = ++x;    // x valdrá 2; a valdrá 2`

- Las expresiones con post in/dec. devuelven el valor previo de la variable:

`x=1;`

`a = x++;    // x valdrá 2; a valdrá 1`

# Leer y escribir: scanf y printf

```
int x;  
Printf("%d\n",x); // escribe el valor de x  
scanf("%d",&x);   // lee x ; ojo al & !!
```



## Cadenas de formato:

%d	entero decimal	%ld	entero largo (long)
%x	entero hexadec.	%lx	ent. largo hex.
%f	float	%ud	entero corto (short)
%lf	double		
%c	char		
%s	cadena		

# Sentencias

- Sentencia simple:  
*expresión;* (siempre termina en punto y coma)
- Sentencia compuesta. Entre llaves.

```
{  
  sentencia  
  sentencia  
  ...  
}
```

# Estructuras algorítmicas

- **if** (*expresión*) *sentencia* [ **else** *sentencia* ]
- **while** (*expresión*) *sentencia*
- **do** *sentencia* **while** (*expresión*)
- **for** (*expresión*;    <- inicialización  
          *expresión*;    <- condición de mantenimiento  
          *expresión*)    <- acción en cada paso  
          *sentencia*

# Sentencia switch

- Similar al *case* de otros lenguajes.

```
switch (expresión)  
{  
    case valor1:  
        ... sentencias ...  
        break;  
    case valor2: ... break;  
    ... ..  
    default:  
        ... sentencias ...  
}
```



# break, continue y goto

- Sirven para salir prematuramente de un bucle:
  - **break** sale definitivamente del bucle
  - **continue** abandona la iteración actual y entra en la siguiente iteración
- También existe **goto** *etiqueta*

# Vectores y matrices

*tipo variable [dimensión1][dimensión2]...*

Ejemplos:

```
int vector [10]; float M[3][3];
```

```
vector[5]=1; M[0][2]=18.5;
```

- Los índices van de 0 a N-1
- ¡No se comprueban accesos fuera de rango!

# Cadenas

- Vectores de caracteres: **char cadena[5];**
- Toda cadena termina en un carácter nulo (0)
- El C no tiene operaciones con cadenas; se usan funciones de la biblioteca <string.h>
  - strcpy(s1,s2)      s1:=s2
  - strcat(s1,s2)      s1:=s1+s2
  - strcmp(s1,s2)      compara s1 con s2
  - strlen(s)          longitud de s

# Ejemplo

```
#include <stdio.h>
main()
{
    int x;
    int vec[10] = {9,3,4,8,1,6,2,0,7,5};
    for (x=0;x<10;x++)
    {
        if (vec[x]==5)
            printf("encontré un cinco\n");
        else printf("v[%d] vale: %d\n",x,vec[x]);
    }
}
```

# Funciones

*Tipo nombre\_función (parámetros)  
{ definición }*

Tipo de la función

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

parámetros

Instrucción de retorno

# Funciones

- Se sale con una sentencia **return**
- Los procedimientos son las “funciones void”  
**void** función() { ... }
- Los parámetros siempre se pasan **por valor** (el paso por referencia se hace mediante punteros).

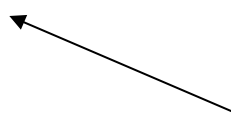
# Funciones

- En la llamada siempre se usan paréntesis:  
    `a = siguiente_valor();`
- En la llamada, no es obligatorio recoger el valor.
- En C no se permite anidar funciones (definir una función dentro de otra).
- Sí se permiten llamadas recursivas.

# Estructuras de datos


```
struct Persona  
{  
    char nombre[30];  
    int edad;  
};
```

Declaración  
de un tipo de datos  
estructurado




```
...  
struct Persona pepe;
```

Declaración  
de una variable



```
...  
pepe.edad = 27;  
strcpy(pepe.nombre, "Pepe Lotilla");
```

Uso de la  
variable





# Tipos enumerados

- Sirven para declarar conjuntos de valores.
- Los valores son constantes enteras consecutivas, comenzando desde cero.

```
// tipo enumerado  
enum puntos_cardinales  
{ norte, sur, este, oeste };
```

```
// variable de tipo enumerado  
enum puntos_cardinales x;  
x = este; // x internamente valdrá 3
```

# Ámbitos y existencia

- Identificadores: tipos, variables o funciones
- Un identificador puede ser **global** o **local**.
  - Id. local: declarado dentro de un bloque {...}
  - Id. global: declarado fuera de bloques.
  - Las funciones sólo pueden ser globales

# Ámbito y existencia

- Ámbito de un identificador:
  - desde el punto en que se declara hasta el final del bloque en el que se declara
- Existencia de un identificador:
  - global: toda la vida del programa
  - local: desde que se entra hasta que se sale del bloque en el que está declarado

# Ámbitos y existencia

```
int x,y; // variables globales

void func()
{
    int x; // variable local
    x = 1; // es la variable local
    y = 1; // la variable global
    {
        int y; // variable más local todavía
        y = 5; // la variable global
    }
}
```

# Variables static

- De ámbito local pero existencia global:

```
int siguiente()
{
    static int contador = 0;
    return contador++;
}
main()
{
    int a = siguiente(); // a vale cero
    int b = siguiente(); // b vale uno
}
```

```
char* cp (char *p, char  
{  
    char *aux = p;  
    while (*q++ = *p++);  
    return aux;  
}
```

## PUNTEROS EN C

# Punteros

- Un puntero es un valor que representa una dirección de memoria donde hay un dato de un determinado tipo.
- El C emplea muchísimo más los punteros que otros lenguajes de programación.

# Declaración de punteros

Declaración: *tipo \* puntero;*

Ejemplo:

**`int* ptr; // puntero a enteros`**



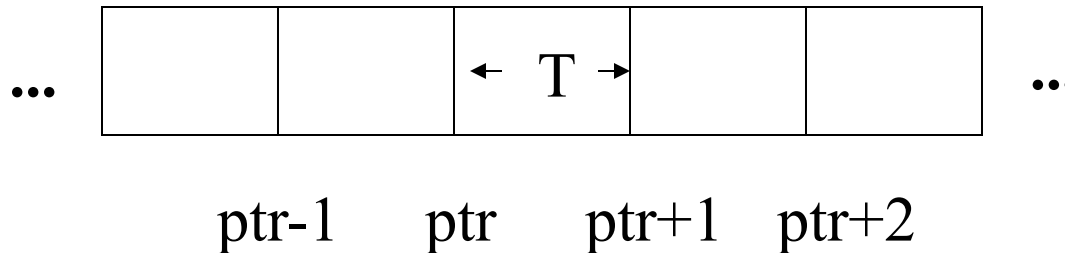
# Uso de punteros

- Operadores:
  - dirección de una variable: **&var**
  - *desreferenciar* un puntero: **\*ptr**
- Ejemplos:

```
int* ptr;  
int var=15;  
ptr = &var; // ptr apunta a var  
*ptr = 33;  // igual que var=33
```

# Aritmética de punteros

- A un puntero se le pueden sumar o restar cantidades enteras:  $ptr+=3$ ,  $ptr--$ , etc.
- Si  $ptr$  es un puntero al tipo  $T$ , la expresión  $ptr+N$  apunta a la dirección de  $ptr$  más  $N$  elementos de tipo  $T$ .



# Aritmética de punteros

- Los punteros pueden usarse como vectores.
- La expresión  $ptr[N]$  equivale a  $*(ptr+N)$
- Por tanto, un puntero de tipo T puede considerarse que apunta a un vector de elementos consecutivos de tipo T.

# Vectores y punteros

- El nombre de un vector es un puntero *constante* al primer elemento, y se puede usar como cualquier puntero:

```
int vec[30];
```

```
int *ptr;
```

```
ptr=vec;           // como ptr=&(vec[0])
```

```
ptr=vec+5;         // ptr apunta a vec[5]
```

```
*(vec+2) = 5;      // igual que vec[2]=5
```

```
vec++;             // ILEGAL: vec es constante
```

# Aritmética de punteros: ejemplos

- Ejemplo: rellenar un vector de enteros.

```
int* ptr; int vec[30];  
for (ptr=vec; ptr<vec+30; ptr++)  
    *ptr = 0;
```

- Ejemplo: copiar dos vectores.

```
int *p,*q; int a[30],b[30];  
for ( p=a,q=b; p<a+30; )  
    *(p++) = *(q++);
```

# Punteros a estructuras

```
struct Cosa* ptr;  
struct Cosa A;  
ptr=&A;  
ptr->campo = xxx;  
// igual que (*ptr).campo=xxx
```

# Paso de parámetros por referencia

- En C se realiza mediante punteros:

```
void duplica ( int* variable )  
{  
    *variable = *variable * 2;  
}
```

```
int a=3;  
duplica(&a); // a valdrá seis
```

# Tipos complejos

- Regla de *derecha-izquierda*.

**int \*ptr [30];** vector de 30 punteros a *int*

**int (\*ptr) [30];** puntero a vectores de 30 enteros

**int \*\*ptr;** puntero a punteros a *int*

**int \*\*ptr [30];** vector de punteros a ptrs. a *int*

**int \*(\*ptr) [30];** puntero de vectores a  
punteros a *int*



# Punteros a funciones

```
int suma (int a,int b) { return a+b; }  
int resta (int a,int b) { return a-b; }
```

```
// puntero a función  
int (* ptr_fun) (int,int);
```

```
ptr_fun = suma;      // sin paréntesis  
x = ptr_fun(4,3);    // x valdrá 7
```

```
ptr_fun = resta;  
x = ptr_fun(4,3);    // x valdrá 1
```

# Errores típicos con punteros

- Punteros sin inicializar
- Accesos fuera de rango
- Punteros que apuntan a variables locales desaparecidas
- Confusión de tipos

# Memoria dinámica: malloc y free

- Para trabajar con memoria dinámica se dispone de dos funciones de biblioteca.

```
ptr = malloc(1000); // reserva 1000 bytes  
free(ptr);          // libera el área de memoria
```

- **¡No hay recogedor de basura!**
- **sizeof(T)** devuelve el tamaño del tipo T:  

```
int* ptr;  
// para reservar 100 enteros  
ptr = malloc(100*sizeof(int));
```