Redes de Computadores
Sistemas Operativos

Curso rápido de C

N. Ayuso, J.L. Briz, P. Ibáñez,
T. Monreal, J. Segarra, J. Alastruey

Departamento de
Informática e Ingeniería
de Sistemas
Universidad Zaragoza

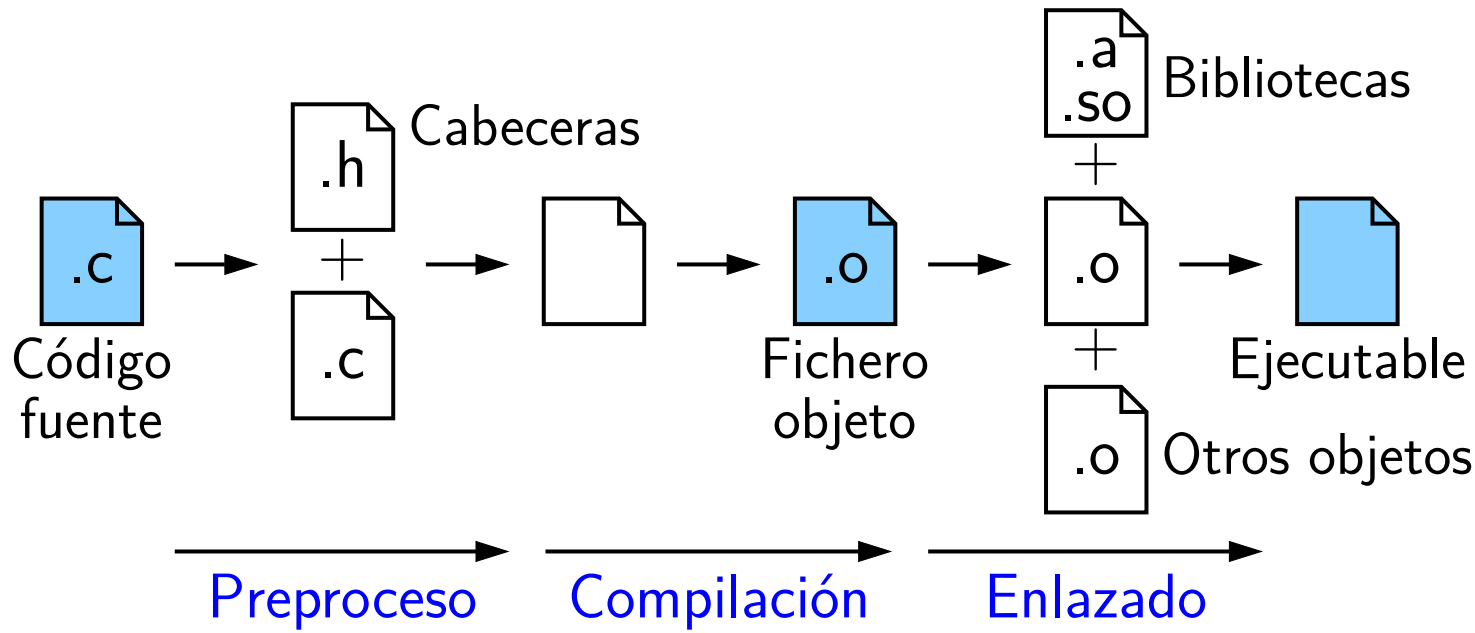
Índice

1 Introducción

- Lenguaje imperativo desarrollado junto con el sistema operativo (SO) UNIX
- Muy ligado al SO
  - Todas las funciones del SO accesibles desde C
  - Todos los SO proporcionan compilador (`$ cc`) y manual (`$ man`) de C
- En este curso rápido (2h) asumimos:
  - Nivel básico de programación imperativa
  - Conocimientos básicos de C/C++ (sintaxis, estructuras de control y operadores)

2 Compilación

- Compilador: programa (gcc, clang, icx ...) que traduce de alto nivel a código máquina
- Fases de compilación:
  1. Preproceso: modifica texto en código fuente
  2. Compilación: traduce a código máquina (específico para el procesador de ese equipo)
  3. Enlazado: genera un ejecutable con el código compilado más el de las funciones de biblioteca necesarias



2 Compilación (II)

- Compilación básica:
  `$ cc fichero.c` → `a.out` (ejecutable)
- Compilación con destino (outfile):
  `$ cc -o fich fich.c` → `fich` (ejecutable)
- Compilación de múltiples fuentes:
  `$ cc -o fich fichero1.c fichero2.c` → `fich` (ejec.)
- Compilación por pasos:
  `$ cc -c fich1.c` → `fich1.o` (compilado, no ejecutable)
  `$ cc -c fich2.c` → `fich2.o` (compilado, no ejecutable)
  `$ cc -o fich fich1.o fich2.o` → `fich` (ejecutable)
  - Último paso, con fuentes ya compilados, realiza enlazado
  - Evita compilar todo cuando sólo se modifica una parte
  - Automatizado en fichero `Makefile` (`$ make` / `$ gmake`)

2 Compilación (III)

Opciones de compilación interesantes:

`-Wall/-Wextra`: muestra más avisos (warnings)

`-S`: genera ensamblador, no código máquina

`-l<biblio>`: incluye la biblioteca `<biblio>` al enlazar

`-O`: optimiza según nivel (-O0,1,2,3,s)

`-g`: incluye información para depurar

`-pg`, `--coverage`, etc.: al ejecutar el programa compilado, generará datos de rendimiento (veces que ejecuta cada instrucción, tiempo que pasa en cada subrutina, etc.), legibles por otros programas (e.g. `$ gprof`, `$ gcov`)

`-fsanitize=<grupo>`: incluye código para detectar errores, según `<grupo>` (e.g. `-fsanitize=address` detecta accesos fuera de rango)

3 Estructura de programa

- Elementos en el código fuente:
  - Inclusión de ficheros (preprocesador)
    - Estándar entre <> (`#include <stdio.h>`)
    - Propios entre "" (`#include "misdefiniciones.h"`)
  - Definición de constantes (preprocesador)
    - `#define FALSO 0 /*C++: const int falso=0; */`
  - Definición de nuevos tipos de datos: `typedef`
  - Alusión/Cabecera de funciones
  - Declaración de funciones (todas al mismo nivel)
  - Función `main()` como programa principal
- Programas grandes divididos en programa principal más módulos (parejas .c/.h)
  - Ficheros .h (*header*) con definiciones
  - Ficheros .c (*c code*) con declaraciones

3.1 Entrada/salida básica

```
printf("formato",variables);
scanf("formato",variables);
getchar();
putchar(char);
```

Elemento	Formato
carácter	%c
decimal	%d
dec. sin signo	%u
real	%f
string	%s
salto	\n
tabulador	\t
otros	...

(En C++: cin>> y cout<<)

```
#include <stdio.h>

char v[6]="LUNES";
int i=245;
float r=3.1416;

printf("%c\n",v[3]); // E
printf("%d\n",i); // 245
printf("%d\n",v[3]); // 69
printf("%x\n",i); // f5
printf("%f\n",r); // 3.1416
printf("%s\n",v); // LUNES
```

3.2 Ejemplo

```c
/* fichero prog.c */
#include <stdio.h>

#define CONST 3
#define NULO 0

int suma(int a, int b);

main() {
  int i=7, j=CONST, k; // resultado
  if (i != NULO) { k=suma(i,j); }
  printf("%d+%d=%d",i,j,k);
}

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

```
$ cc prog.c -o prog
```

3.2 Ejemplo (II)

```c
/* fichero prog.c */
#include <stdio.h>

#define CONST 3
#define NULO 0

int suma(int a, int b);

main() {
  int i=7, j=CONST;
  int k; // resultado

  if (i != NULO) {
    k=suma(i,j);
  }
  printf("%d+%d=%d",i,j,k);
}
```

Separación del módulo «suma» del programa principal

```c
/* fichero suma.c */
int suma(int a, int b) {
  int aux;
  aux=a+b;
  return aux;
}
```

```
$ cc -c suma.c
```

```
$ cc -c prog.c
```

```
$ cc -o prog prog.o suma.o
```

3.2 Ejemplo (III)

División del módulo «suma» en .c/.h

```c
/* fichero prog.c */
#include <stdio.h>
#include "suma.h"

#define CONST 3

main() {
   int i=7, j=CONST;
   int k; // resultado

   if (i != NULO) {
     k=suma(i,j);
   }
   printf("%d+%d=%d",i,j,k);
}
```

```c
/* fichero suma.h */

#define NULO 0

int suma(int a, int b);
```

```c
/* fichero suma.c */
#include "suma.h"

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

3.2 Ejemplo (IV)

Comentarios para la generación automática de documentación

```
/* fichero suma.h */

#define NULO 0

int suma(int a, int b);
```

```
$ doxygen -g
```

```
$ doxygen
```

```
/**
 * @file suma.c suma.h
 * @brief Ejemplo de C
 * @author Profesores UZ
 */

/**
 * Valor para evitar la suma
 */
#define NULO 0

/**
 * Suma dos enteros
 *
 * @param[in] entero A
 * @param[in] entero B
 * @return A+B
 */
int suma(int a, int b);
```

3.2 Ejemplo (V)

My Project

Main Page | Files | Search

File List

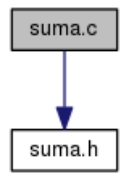
Functions

suma.c File Reference

Ejemplo de C. More...

```
#include "suma.h"
```

Include dependency graph for suma.c:



Functions

int **suma** (int a, int b)

Detailed Description

Ejemplo de C.

suma.h

Author

Profesores UZ

Generated by doxygen 1.8.11

4 Tipos de datos

	Tipo	Bits	Declaración	Uso
entero	char	≥ 8	`char c;`	`c=97; c='a';`
	int	≥ 16	`int i;`	`i=023; //oct`
	short (int)	≥ 16	`short int i;`	`i=0x30A; //hex`
	long (int)	≥ 32	`long int i; long j;`	`i=-5L;`
	long long (int)	≥ 64	`long long i;`	
	int$N$_t	N	`int64_t i;`	$N \in \{8, 16, 32, 64\}$
	unsigned		`unsigned long i;`	`i` $\in \mathbb{N}$
	uint$N$_t		`uint16_t i;`	
	size_t		`size_t tam;`	`tam` $\in \mathbb{N}$
	ssize_t		`ssize_t tam;`	`tam` $\in \mathbb{Z}$
	puntero *		`int *p; char *q;`	
real	float	32	`float f;`	`f=-5.3e8;`
	double	64	`double f;`	`f=-5.3;`
	long double	≥ 80	`long double f;`	`f=-5;`

4.1 Conversión de tipos

- Implícita por *widening*:
  ```
  short i=2; long j; j=i;  //rellena con ceros a la izquierda
  ```
- Implícita a tipo de operando más genérico:
  ```
  num=3*2.1;                    // float 2.1 → mult. float → 3 float
  ```
- Implícita por asignación:
  ```
  int i; i=2.8;                                    // trunca a 2
  ```
- Explícita (cast):
  ```
  num = 3.0 + (float)1/2;
  ```
- ¡Cuidado!
  ```
  num = 3.0 + 1/2;          // división (entera) antes que suma
  num = 3.0 + (float)(1/2);       // división antes que cast
  int i = 2.8;                 // pérdida de información (trunca)
  char j = 300;               // pérdida de información (módulo)
  unsigned char c=20; int i = -c; c =- c;   // ¿i == c?
  ```

5.1 Atributos de variables

- Variable declarada fuera de funciones: global
  (visible desde varias funciones)

  auto (por defecto): visible desde todas
  static: visible sólo desde las del propio fichero
  extern: especifica variables declaradas en otros ficheros
- Variable declarada dentro de una función: local
  (visible sólo dentro la función)

  auto (por defecto): se reserva espacio en cada llamada y se libera (pierde su valor) al salir
  static: sólo se inicializa en la primera llamada y mantiene su valor de una llamada a otra

5.1 Atributos de variables (II)

- Otros atributos:

  register: indicación al compilador para que mapee la variable sobre un registro físico de la máquina (si es posible)

  volatile: indicación al compilador para evitar optimizaciones sobre esta variable. Se usa para variables cuyo valor puede cambiar de forma ajena al programa

  const: indicación al compilador para que dé error de compilación si detecta código que la modifica

- Asignación/liberación dinámica de memoria:
  - `new()` $\Rightarrow$ `malloc()`
  - `delete()` $\Rightarrow$ `free()`

5.2 Ejemplo

```c
#include <stdio.h>

int i=0;
int j=0;

void incrementar() {
  int j=0; // ojo! hay una j global
  static int k=0;

  i++;
  j++;
  k++;
  printf("i=%d, j=%d, k=%d\n",i,j,k);
}

int main() {
  incrementar(); incrementar(); incrementar();
}
```

6 Vectores

Vector: conjunto de variables del mismo tipo que se referencian por un nombre común (no existe la clase vector como en C++)

Declaración: `char v[5]; // Declaración de 5 elementos (v[0]...v[4])` que ocupan posiciones consecutivas de memoria

Inicialización: `char v[5]={'a','e','i','o','u'};`

Uso: `v[4]='z'; // Asigno el valor 'z' al 5º elemento del vector v`

Cálculo de posición de memoria:

$$dirBase + índice \cdot tamElemento$$

- Nombre de un vector $\equiv$ Dirección de su primer elemento (es una constante)
- Elemento `v[3]` en dirección `v+3*sizeof(char)`
- `char M[2][5]; // 10 elementos consecutivos por filas: M[0][0], M[0][1]... M[0][4], M[1][0]... M[1][4]`

6.1 Strings

String: vector de caracteres acabado con el carácter `'\0'`. No existe el tipo string como ocurre en C++

Inicialización: `char v[6] = {'L','U','N','E','S','\0'};`

```
char v[6] = {'L','U','N','E','S',0};
char v[6] = "LUNES"; // Una constante string es una cadena de caracteres entre comillas dobles
```

¡Cuidado!

```
char v[6]; v="LUNES"; // Mal: operador = no copia vectores
char v[6]="LUNES"; v[2]=0;        // Bien: "LU" en string v
char v[6]="LUNES"; v[5]='a';   // Ojo: v ya no es un string
char v[6]="LUNES"; v[8]='a';        // Ojo: fuera de rango
```

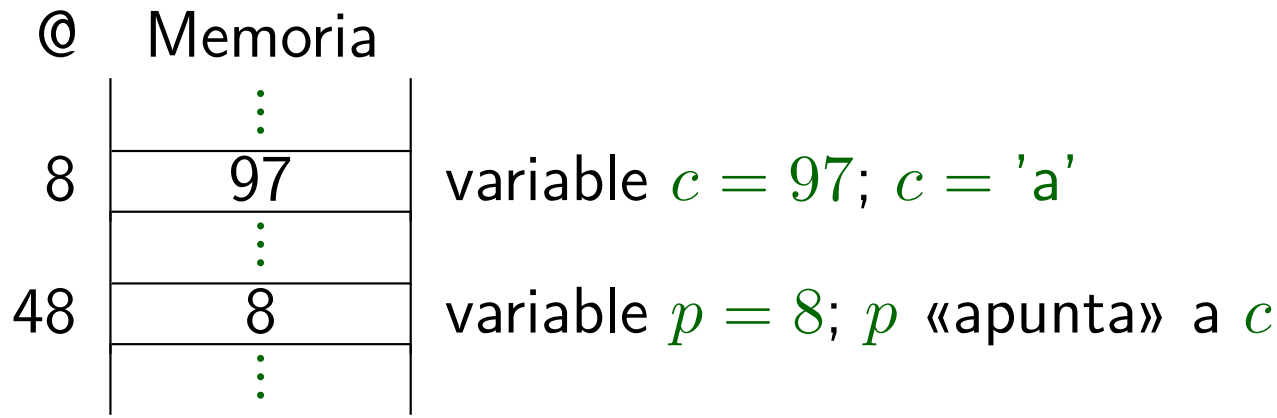
6.2 Punteros

Puntero: variable que contiene la dirección de memoria de una variable de un tipo dado

Declaración: `char *p; //`declaración (sin inicializar) de un puntero a una variable de tipo carácter

Operador &: devuelve la dirección de memoria del operando: `char c; p = &c;`

Operador *: devuelve el valor almacenado en la dirección especificada: `char c; c=*p;`

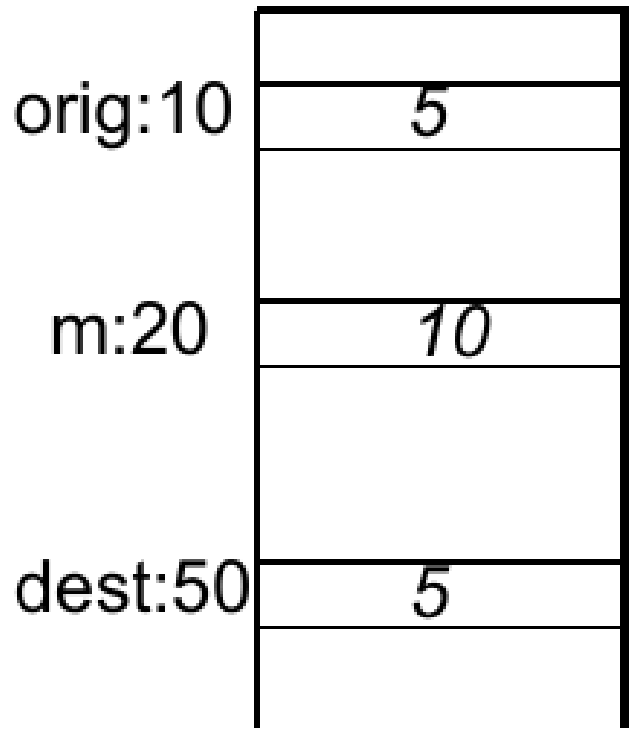


6.3 Ejemplo

```c
#include <stdio.h>

main() {
  int dest, orig;
  int *m;

  orig=5;
  m=&orig;
  dest=*m;
  printf("%d %d",orig,dest);
  *m=6;
  printf("%d %d",orig,dest);
  dest=orig;
  printf("%d %d",orig,dest);
}
```



Salida

```
5 5
6 5
6 6
```

6.4 Operac. con vectores y punteros

Recordar: $dirBase + índice \cdot tamElemento$

- Nombre de un vector (sin corchetes) ($dirBase$) $\equiv$ dirección (constante) de su primer elemento:
  `char v[10]; v` $\equiv$ `&v[0]`
- Todas las operaciones con vectores y punteros se hacen de acuerdo a su tipo base:
  `int *m; m+1` $\equiv$ `m+(1*sizeof(int))`
  `int *m,i; m+i` $\equiv$ `m+(i*sizeof(int))`
  `int *m; m[3]` $\equiv$ `*(m+(3*sizeof(int)))`
  `int v[3]={1,5,15}, p=v;` ¿`*(p+2)`? ¿`*p+2`?

6.4 Operac. con vectores y punteros (II)

- Cualquier variable puntero puede indexarse con corchetes como un vector:

```
int *q, v2[5];

q = v2;
*(q + 1) = 100;
q[1] = 100;
```

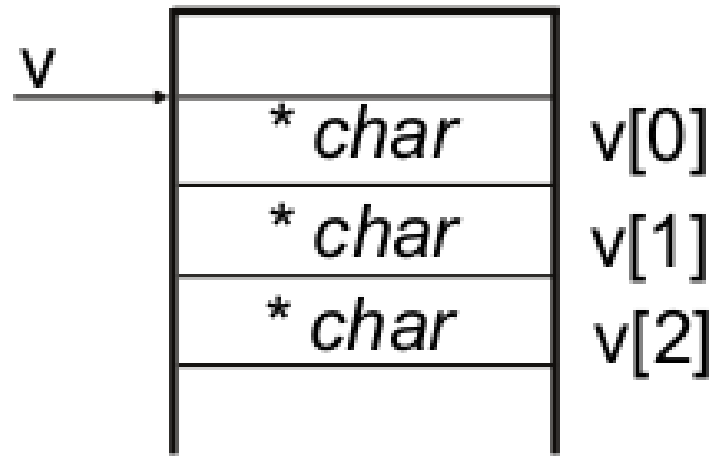
- Cuidado

```
char v[5], m[5];
char *p, v1[10];
```

```
v = m; /* ERROR: v es una constante */
v1 = p; /* lo mismo */
v++;   /* lo mismo */
v[0] = m[0]; /* ? */
```

6.5 Vectores de punteros

- Reserva de espacio para 3 punteros a carácter:
  `char *v[3];`



- Ejemplo:

```
const char * array[] = {
    "First entry",
    "Second entry",
    "Third entry",
};
```

6.6 Punteros a funciones

- El nombre de una función es una constante cuyo valor es la @ inicial de la función
- Se puede declarar una variable para que contenga la @ de una función:

```
int (*p)( );
/* p: puntero a funcion que devuelve un entero */
```

- Cuidado:

```
int *p( );
/* p: funcion que devuelve un puntero a entero */
```

- Ejemplo:

```
void suma(f1,f2,d);/* la misma de antes */
void (*p)();
p = suma;
(*p)(i,j,&k); /* lo mismo que: suma(i,j,&k); */
```

7 Paso de parámetros

Paso por valor: la llamada copia un valor, especificado como parámetro, a una variable dentro de la función

- La modificación de la copia no modifica el original

> En cualquier computador, el paso de parámetros es siempre por valor

Paso por referencia: paso por valor de la dirección de memoria (puntero) donde reside un dato

- La modificación del contenido de esa dirección modifica el dato referenciado
- Si un parámetro ocupa muchos bytes puede ser recomendable pasarlo por referencia, incluso si no va a modificarse, para evitar copiarlo

7 Paso de parámetros (II)

Paso por valor: misma sintaxis en C y C++

```
int incrementa(int var) {
  return var++;
}
```

```
int a a=incrementa(10); a=incrementa(a);
```

Paso por referencia en C++: sintaxis abreviada

```
void incrementa(int & var) {
  var++; return;
}
```

```
int a=10; incrementa(a); // incrementa(10) MAL
```

Paso por referencia en C: sintaxis literal (punteros)

```
void incrementa(int * var) {
  (*var)++; return;
}
```

```
int a=10; incrementa(&a); // incrementa(&10)
    MAL
```

8 Structs

struct: estructura de datos compuesta por elementos individuales (llamados campos o miembros) que pueden ser de distinto tipo

Operador = : copia

Operador . : acceso a campo

Operador -> : acceso a campo desde puntero a struct

```
struct fecha {
  int mes;
  int dia;
  int anyo;
};

struct fecha hoy,ayer;
struct fecha *manyana;

hoy.dia=1;
hoy.mes=2;
hoy.anyo=3;

ayer=hoy;
manyana=&hoy;

(*manyana).dia=2;
manyana->dia=2;
```