

Sistemas Operativos

Curso
2016-2017

Revisión: Programación en C

A. Bautista, L. Piñuel, J. Recas, J.C. Sáez

¿Por qué aprender C ?

- Permite programación de alto y bajo nivel
- Mejor control de los mecanismos de bajo nivel
- Mayor rendimiento que lenguajes con mayor nivel de abstracción (Java, C++...)
 - Java/C++ ocultan muchos detalles necesarios para escribir código relacionado con el SO

Pero,....

- Es necesario asumir la responsabilidad de la gestión de memoria
- La inicialización de variables y el chequeo de errores deben ser explícitos

Objetivos de esta introducción



- Introducir/Revisar algunos conceptos básicos de C
 - Los detalles se irán descubriendo con el uso
- Advertir sobre los fallos típicos
 - Evitar perdidas de tiempo en la realización de las prácticas
- Conseguir que el alumno entienda rápidamente programas complejos
 - Y que sea capaz de escribir código basado en ellos

Ejemplo simple (1)



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

```
void main(void)
```

```
{
```

```
    printf("Hello World. \n \t and you ! \n ");
```

```
    /* print out a message */
```

```
    return;
```

```
}
```

Salida:

```
$Hello World.
```

```
    and you !
```

```
$
```

Ejemplo simple (2)



- **#include <stdio.h>**
 - Incluir fichero de cabecera *stdio.h*
 - No es necesario “;” al final
 - Sólo letra minúsculas

(C es sensible a mayúsculas/minúsculas)
- **void main(void){ ... }**
 - Código a ejecutar
- **printf(“ /* mensaje */ ”);**
 - \n = salto de línea \t = tabulador
 - Utilizar “\” delante de los caracteres especiales

Tipos de datos simples

Tipo	Bytes	Valores	Formato
int	4	$-(2^{31}-1) \leftrightarrow (2^{31}-1)$	%d
char	1	$-128 \leftrightarrow 127$	%c
float	4	$-128 \leftrightarrow 127$	%f
double	8	$1.7E+/-308$	%lf
long	4*	$-(2^{31}-1) \leftrightarrow (2^{31}-1)$	%l
short	2	$-(2^{15}-1) \leftrightarrow (2^{15}-1)$	

Disposición de los datos



```
int x = 5, y = 10;
float f = 12.5, g = 9.8;
char c = 'c', d = 'd';
```

x	y	f	g	c	d
5	10	12.5	9.8	c	d
4300	4304	4308	4312	4316	4317

Ejemplo



```
#include <stdio.h>
void main(void)
{
    int nstudents = 0;           /*Init. required */
    printf("How many students does Cornell have ?:");
    scanf ("%d", &nstudents);    /* Read input */
    printf("Cornell has %d students.\n", nstudents);
    return ;
}
```

Salida:

```
$How many students does Cornell have ?: 20000 (enter)
Cornell has 20000 students.
$
```


Operadores y Sintaxis Básica



■ Operadores:

■ Aritméticos

- `int i = i+1; i++; i--; i *= 2;`
- `+, -, *, /, %,`

■ Relacionales y lógicos

- `<, >, <=, >=, ==, !=`
- `&&, ||, &, |, !`

■ Sintaxis:

- `if () { } else { }`
- `while () { }`
- `do { } while ();`
- `for(i=1; i <= 100; i++) { }`
- `switch () {case 1: ... }`
- `continue; break;`

Ejemplo



```
#include <stdio.h>
#define DANGERLEVEL 5                /*C Preprocessor macro*/

void main(void)
{
    float level=1;

    if (level <= DANGERLEVEL){ /*replaced by 5*/
        printf("Low on gas!\n");
    }
    else printf("Good driver !\n");
    return;
}
```

Arrays de 1 dimensión



```
#include <stdio.h>
void main(void)
{
    int number[12];    /*12 cells, one cell per student*/
    int index, sum = 0;

    /* Always initialize array before use */
    for (index = 0; index < 12; index++) {
        number[index] = index;
    }
    /* now, number[index]=index; will cause error:why ?*/

    for (index = 0; index < 12; index = index + 1) {
        sum += number[index]; /* sum array elements */
    }
    return;
}
```

Más sobre arrays... (1)

■ Cadenas de caracteres (strings)

```
char message[6]={ 'H' , 'E' , 'L' , 'L' , 'O' , '\0' }; /*'\0' = end */  
printf("%s", message);                               /*print until '\0'*/
```

- Inicialización equivalente → `char message[] = "hello";`
- Funciones relacionadas con strings (<strings.h>): *strcpy, strncpy, strcmp, strncmp, strcat, strncat, strstr, strchr*

■ Arrays multi-dimensionales

```
int points[3][4];    /* NOT points[3,4] */  
points [1][3] = 12;  
printf("%d", points[1][3]);
```

Más sobre arrays... (2)



■ Inicialización

- ¡Es necesario inicializarlos antes de usarlos!

```
int number[12];  
printf("%d", number[20]);
```

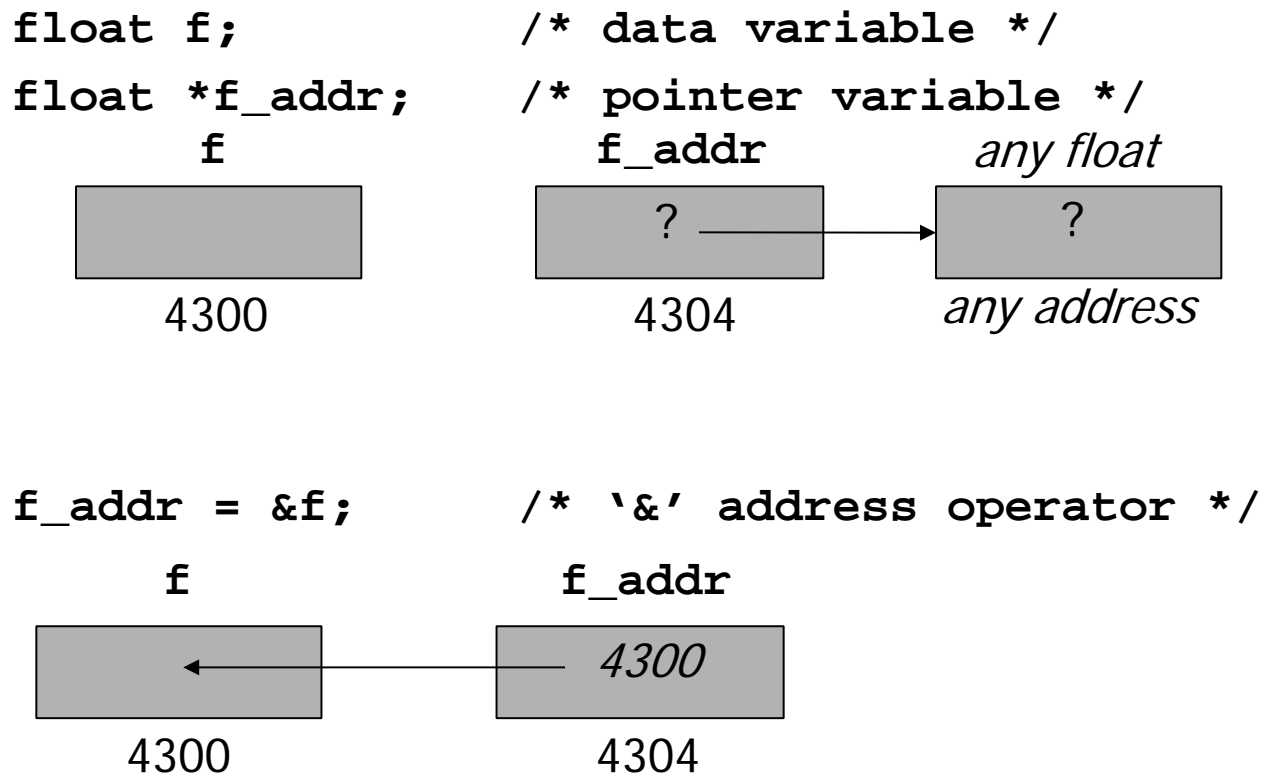
- Los strings terminan con el carácter '\0'

```
char msg[6]={ 'H' , 'E' , 'L' , 'L' , 'O' , '\0' };  
printf("%s", msg); /* print until '\0' */
```

Punteros



- *Puntero* = variable que almacena una dirección de memoria (ej: dirección de otra variable o cualquier dirección)



Asignación dinámica de memoria



- La asignación y desasignación deben ser explícitas

```
#include <stdio.h>
void my_function(void) {
    char c;
    int *ptr;

    /* allocate space to hold an int */
    ptr = malloc(sizeof(int));

    /* do stuff with the space */
    *ptr=4;

    /* free up the allocated space */
    free(ptr);
}
```

Gestión de errores

- A diferencia de Java, no existen “excepciones”
- Es necesario realizar la comprobación de errores de manera manual
 - Siempre que se use una función que no se haya escrito
 - Los errores pueden aparecer en cualquier sitio (¡Cuidado con el manejo de punteros!)

Funciones



- ¿Cuándo usarlas?
 - Programa demasiado largo
 - Para facilitar:
 - Programación
 - Depuración
 - Reutilización de código
- ¿Cómo usarlas?
 - Paso de parámetros
 - Por valor
 - Por referencia
 - Valores de retorno
 - Por valor
 - Por referencia

Ejemplo sencillo



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

```
/* function prototype at start of file */  
int sum(int a, int b);
```

```
void main(void){  
    int total = sum(4,5); /* call to the function */  
    printf("The sum of 4 and 5 is %d", total);  
}
```

```
int sum(int a, int b){    /* arguments passed by value*/  
    return (a+b);        /* return by value */  
}
```

Argumentos por referencia



```
#include <stdio.h>

/* function prototype at start of file */
int sum(int *pa, int *pb);

void main(void){
    int a=4, b=5;
    int *ptr = &b;
    int total = sum(&a,ptr); /* call to the function */
    printf("The sum of 4 and 5 is %d", total);
}

int sum(int *pa, int *pb){
    /* args passed by reference */
    return (*pa+*pb);      /* return by value */
}
```

¿Por qué se usan punteros? (1)



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

```
void swap(int, int);
```

```
void main() {  
    int num1 = 5, num2 = 10;  
    swap(num1, num2);  
    printf("num1 = %d and num2 = %d\n", num1, num2);  
}
```

¡Código incorrecto!

```
void swap(int n1, int n2) { /* passed by value */  
    int temp;  
    temp = n1;  
    n1 = n2;  
    n2 = temp;  
}
```

¿Por qué se usan punteros? (2)



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

```
void swap(int *, int *);
```

```
void main() {  
    int num1 = 5, num2 = 10;  
    swap(&num1, &num2);  
    printf("num1 = %d and num2 = %d\n", num1, num2);  
}
```

```
void swap(int *n1, int *n2) {  
    /* passed and returned by reference */  
    int temp;  
    temp = *n1;  
    *n1 = *n2;  
    *n2 = temp;  
}
```

¿Por qué es incorrecto este ejemplo?



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

```
void dosomething(int *ptr);
```

```
void main() {
```

```
    int *p;
```

```
    dosomething(p)
```

```
    printf("%d", *p);    /* will this work ? */
```

```
}
```

```
void dosomething(int *ptr){
```

```
    /* passed and returned by reference */
```

```
    int temp=32+12;
```

```
    *ptr = temp;
```

```
}
```

¡Error durante la ejecución!

Solución 1



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

```
void dosomething(int *ptr);
```

```
void main() {  
    int a;  
    int *p=&a;  
    dosomething(p)  
    printf("%d", *p);    /* will this work ? */  
}
```

```
void dosomething(int *ptr){  
    /* passed and returned by  
reference */  
    int temp=32+12;  
    *ptr = temp;  
}
```

Solución 2



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

```
void dosomething(int *ptr);
```

```
void main() {  
    int *p = malloc(sizeof(int));  
    dosomething(p)  
    printf("%d", *p);    /* will this work ? */  
    free(p);  
}
```

```
void dosomething(int *ptr){  
    /* passed and returned by  
reference */  
    int temp=32+12;  
    *ptr = temp;  
}
```


Uso de arrays como argumento



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

```
void init_array(int array[], int size) ;
```

```
void main(void) {  
    int i,list[5];  
    init_array(list, 5);  
    for (i = 0; i < 5; i++)  
        printf("next:%d", list[i]);  
}
```

```
void init_array(int array[], int size) { /* why size ? */  
    /* arrays ALWAYS passed by reference */  
    int i;  
    for (i = 0; i < size; i++)  
        array[i] = 0;  
}
```

Programas con varios ficheros



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

void main(void)
{
    myproc();
}
```

main.c

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

int mydata=0;

void myproc(void)
{
    mydata=2;
    . . . /* some code */
}
```

mypgm.c

```
void myproc(void);
extern int mydata;
```

mypgm.h

Declaraciones externas



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

```
extern char user2line [20];  
char user1line[30];
```

```
/*global defined in another file*/  
/*global for this file*/
```

```
void dummy(void);
```

```
void main(void) {  
    char user1line[20];  
    . . .  
}
```

```
/*different from earlier*/  
/*restricted to this func*/
```

```
void dummy(){  
    extern char user1line[];  
    . . .  
}
```

```
/*the global user1line[30]*/
```

Estructuras



- Equivalentes a las clases Java / C++ pero sólo con datos (sin métodos)

```
#include <stdio.h>

struct birthday{
    int month;
    int day;
    int year;
};

void main() {
    struct birthday mybday;      /* - no 'new' needed ! */
    mybday.day=1; mybday.month=1; mybday.year=1977;
    printf("I was born on
    %d/%d/%d\n", mybday.day, mybday.month, mybday.year );
}
```

Estructuras: “.” vs “->”



```
#include <stdio.h>
struct birthday{
    int month;
    int day;
    int year;
};

void main() {
    struct birthday mybday;
    struct birthday* ptrMybday=&mybday;

    mybday->day=1; mybday->month=1; mybday->year=1977;
    ...
    printf("I was born on
    %d/%d/%d\n",mybday.day,mybday.month,mybday.year);
}
```

Paso de estructuras



```

/* pass struct by value */
void display_year_1(struct birthday mybday) {
    printf("I was born in %d\n", mybday.year);
}
/* - inefficient: why ? */

/* pass struct by reference */
void display_year_2(struct birthday *pmybday) {
    printf("I was born in %d\n", pmybday->year);
    /* warning ! '->', not '.', after a struct pointer*/
}

/* return struct by value */
struct birthday get_bday(void){
    struct birthday newbday;
    newbday.year=1971; /* '.' after a struct */
    return newbday;
}

```

Tipos de datos con nombre



- Mayor claridad y facilidad de uso

```
typedef int Employees;  
Employees my_company;           /* same as int my_company; */
```

```
typedef struct person Person;  
Person me;                      /* same as struct person me; */
```

```
typedef struct person *Personptr;  
Personptr ptrtome;              /* same as struct person *ptrtome; */
```

Más punteros



```
int month[12];  
/* month is a pointer to base address 430*/  
  
int *ptr = month + 2;  
/* ptr points to month[2], => ptr is now (430+2*4)= 438 */  
  
month[3] = 7;  
/* month address + 3 * int_size=> int at (430+3*4) is 7 */  
  
ptr[5] = 12;  
/* int at (438+5*4) is now 12. Thus, month[7]=12 */  
  
ptr++;  
/* ptr <- 438 + 1 * size of int = 442 */  
  
(ptr + 4)[2] = 12;  
/* accessing ptr[6] i.e., month[9] */
```


Cadenas de caracteres



```
#include <stdio.h>
void main() {
```

```
    char msg[10];                /* array of 10 chars */
    char *p;                     /* pointer to a char */
    char msg2[]="Hello";        /* msg2 = 'H''e''l''l''o''\0' */

    msg = "Bonjour";            /* ERROR. msg has a const address.*/
    p   = "Bonjour";            /* address of "Bonjour" goes into p */
    msg = p;                     /* ERROR. Message has a constant address. */

    p = msg;                     /* OK */

    p[0] = 'H', p[1] = 'i', p[2]='\0'; /* *p and msg are now "Hi" */
}
```

Parámetros *argc* y *argv*



```
#include <stdio.h>

/* program called with cmd line parameters */
void main(int argc, char *argv[]) {
    int ctr;

    for (ctr = 0; ctr < argc; ctr = ctr + 1) {
        printf("Argument #%d->|%s|\n", ctr, argv[ctr]);
    }
    /* ex., argv[0] == the name of the program */
}
```

Punteros a función



- ¿Ventaja?
 - Mayor flexibilidad

```
/* function returning integer */  
int func(void);
```

```
/* function returning pointer to integer */  
int *func(int a);
```

```
/* pointer to function returning integer */  
int (*func)(void);
```

```
/* pointer to func returning ptr to int */  
int *(*func)(int);
```

Punteros a función - Ejemplo



```
#include <stdio.h>
void myproc (int d);
void mycaller(void (* f)(int), int param);

void main(void) {
    myproc(10);          /*call myproc with parameter 10*/
    mycaller(myproc,10); /* and do the same again ! */
}

void mycaller(void (* f)(int), int param){
    (*f)(param);         /* call function *f with param */
}

void myproc (int d){
    . . .                /* do something with d */
}
```

Por último ...



- SIEMPRE inicializar las variables antes de usarlas (especialmente los punteros)
- No devolver punteros a variables locales de la función
- Pedir memoria para las estructuras de datos dinámicas
- No utilizar punteros después de liberarlos con *free()*
- Comprobar errores (es mejor comprobar de más que quedarse corto)
- Un *array* es también un puntero, pero su valor es inmutable.
- Prestar atención a los ejemplos proporcionados en cada práctica.