Sistemas Operativos

Curso
2020-2021

Grado en Ingeniería del Software

Revisión: Programación en C

Profesores de SSOO

¿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 la introducción

- Introducir/Revisar 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 I

```c
#include <stdio.h>

void main(void)
{
  printf("Hello World. \n \t and you! \n");
  /* print out a message */
  return;
}
```

Salida por terminal:

```
$ ./example1
Hello World.
        and you !
$
```

Ejemplo I

```
#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 */ ");
```

- Utilizar '\' delante de los caracteres especiales:
  - '\n' = salto de línea
  - '\t' = tabulador

Tipos de datos simples

Tipo	Bytes	Rango	Formato
char	1	$[-128, 127]$ o $[0, 255]$	%c
unsigned char	1	$[0, 255]$	%uc
short (int)	2	$[-(2^{15}-1), (2^{15}-1)]$	%hd
int	2	$[-(2^{15}-1), (2^{15}-1)]$	%d
	4	$[-(2^{31}-1), (2^{31}-1)]$	
long (int)	4	$[-(2^{31}-1), (2^{31}-1)]$	%ld
long long (int)	8	$[-(2^{63}-1), (2^{63}-1)]$	%lld
float	4	$\pm$[1.2E-38, 3.4E+38]	%f
double	8	$\pm$[2.3E-308, 1.7E+308]	%lf
long double	10	$\pm$[3.4E-4932, 1.1E+4932]	%Lf

Usar siempre `sizeof(<tipo_de_dato>)` y definiciones de `limits.h` (p.ej. `INT_MAX`)

Datos en memoria

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

Tag	x	y	f	g	c	d
	5	10	12.5	9.8	'c'	'd'
Addr	0x4300	0x4304	0x4308	0x430c	0x4310	0x4311

Ejemplo II

```c
#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 por terminal:

```
$ ./example2
How many students does Cornell have ?: 20000 (enter)
Cornell has 20000 students.
$
```

Operadores

- Aritméticos: `+ - / * % ++ --`
  - `i = i+1; i++; i--; i *= 2; i = i%3;`
- Operadores de bits: `& | ^ << >> ~`
  - `i = i&0x0f; i |= 0x1; i ^= i; i = i<<2;`
- Operadores relacionales: `< > <= >= == !=`
- y operadores lógicos: `&& || !`
  - `if (((i<100) && (i!= 4)) || !finish) {...}`

Sintaxis básica

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

Ejemplo III

```
#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;
}
```

Vectores (Arrays 1D)

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

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

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

Más sobre arrays

- Los strings terminan en '\0':

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

- Inicialización equivalente a

```
char message[] = "hello";
```

- Arrays multi-dimensionales

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

- ¡Es necesario inicializarlos antes de usarlos!

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

Más sobre arrays

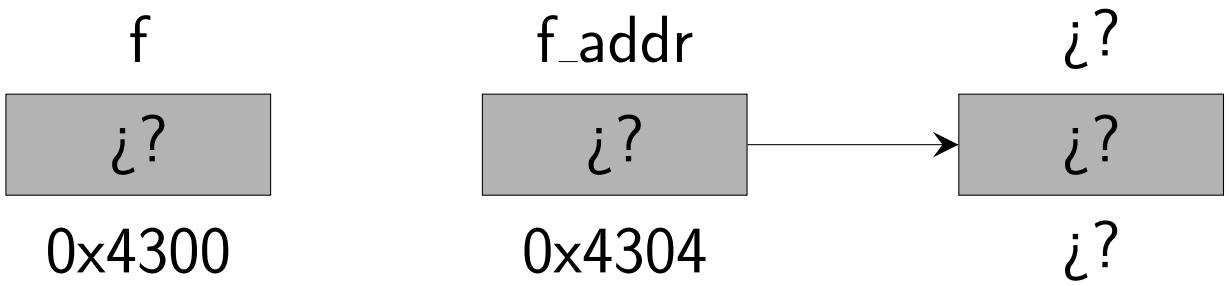
- Funciones de librería (`strings.h`):
  - strcpy, strncpy,
  - strcmp, strncmp,
  - strcat, strncat,
  - strlen, ...
- Ejemplo:

```
char message[100];
char msg_hello[] = "Hello ";
char msg_world[] = "World!";
strncpy(message, msg_hello, 100);
strncat(message, msg_world, 100-strlen(msg_hello));
prinft("%s", message);
```

Punteros

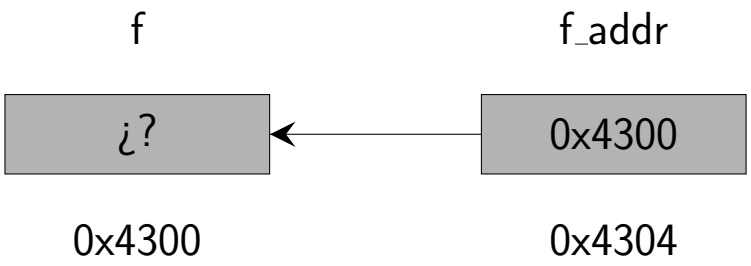
- Puntero = variable que almacena una dirección de memoria (ej: dirección de otra variable o cualquier dirección):

```
float f;          /* data variable */
float *f_addr;     /* pointer variable */
```

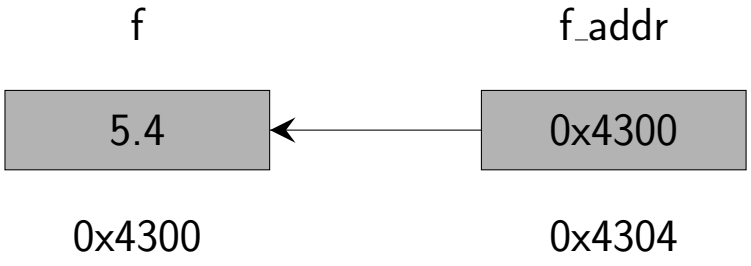


Punteros

f_addr=&f;



*f_addr=5.4;



Asignación dinámica de memoria

- La asignación y liberación 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/C++, no hay '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

- ¿Cuando usarlas?
  - Programa demasiado largo
  - Para facilitar:
    - Programación y depuración
    - Reutilización de código
- ¿Cómo usarlas?
  - Paso de parámetros
    - Por valor y por referencia
  - Valores de retorno
    - Por valor y por referencia

Ejemplo sencillo

```c
#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

```c
#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);
}

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

¿Por qué se usan punteros? I

```c
#include <stdio.h>

void swap(int, int);

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

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

Código incorrecto

¿Por qué se usan punteros? II

```c
#include <stdio.h>

void swap(int *, int *);

void main(void) {
  int num1 = 5, num2 = 10;
  swap(&num1, &num2);
  printf("num1 = %d and num2 = %d\n", num1, num2);
}
/* passed and returned by reference */
void swap(int *n1, int *n2) {
  int temp;
  temp = *n1;
  *n1 = *n2;
  *n2 = temp;
}
```

¿Por qué es incorrecto?

```c
#include <stdio.h>

void dosomething(int *ptr);

void main(void) {
  int *p;
  dosomething(p)
  printf("%d", *p);      /* will this work ? */
}
/* passed and returned by reference */
void dosomething(int *ptr){
  int temp=32+12;
  *ptr = temp;
}
```

Posible error durante la ejecución

Solución 1

```c
#include <stdio.h>

void dosomething(int *ptr);

void main(void) {
  int a;
  int *p=&a;
  dosomething(p)
  printf("%d", *p);      /* will this work ? */
}
/* passed and returned by reference */
void dosomething(int *ptr){
  int temp=32+12;
  *ptr = temp;
}
```

Solución 2

```c
#include <stdio.h>

void dosomething(int *ptr);

void main(void) {
  int *p = malloc(sizeof(int));
  dosomething(p)
  printf("%d", *p);      /* will this work ? */
  free(p);
}
/* passed and returned by reference */
void dosomething(int *ptr){
  int temp=32+12;
  *ptr = temp;
}
```

Solución 3

```c
#include <stdio.h>

void dosomething(int *ptr);

void main(void) {
  int a;
  dosomething(&a)
  printf("%d", a);      /* will this work ? */
}
/* passed and returned by reference */
void dosomething(int *ptr){
  int temp=32+12;
  *ptr = temp;
}
```

Uso de arrays como argumento

```c
#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

my_pgm.h:

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

my_pgm.c:

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

int mydata=0;

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

main.c:

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

void main(void){
  printf("%d", mydata);
  myproc();
}
```

Declaraciones externas

```
#include <stdio.h>
extern char user2line[20]; /*global def. in another file*/
char user1line[30];        /*global for this file (posible
                             for others)*/
static int a=3;            /*global for this file (private)*/

static void dummy(void);   /*function prototype (private)*/

void main(void) {
  char user1line[20];      /*different from earlier*/
  . . .                    /*restricted to this func*/
}
static void dummy(){       /*not visible for other modules*/
  extern char user1line[]; /*the global user1line[30]*/
  static int i;            /*keeps its value between invocations*/
  . . .
}
```

Estructuras

- Equivalentes a las clases Java/C++ pero sólo con datos (sin métodos), pero pueden contener punteros a función.

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

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

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

Paso de estructuras

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

/* 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

- Con `typedef` se pueden definir nombres de tipos

```
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 0x430*/

int *ptr = month + 2;
/* ptr points to month[2], => ptr is now (0x430+2*4)=0x438 */

month[3] = 7;
/* month address + 3 * sizeof(int) => int at (0x430+3*4) is 7 */

ptr[5] = 12;
/* int at (0x438+5*4) is now 12. Thus, month[7]=12 */

ptr++;
/* ptr <- 438 + 1 * sizeo(int) = 43C */

(ptr + 4)[2] = 12;
/* accessing ptr[6] i.e., month[9] */
```

Cadenas de caracteres

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

  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. Msg has a const. address */

  p = msg;            /* OK */

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

}
```

Parámetros argc y argv

```c
#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 */
}
```

Salida por terminal:

```
$ ./example param1 param2 3 4 5 'Hello world'
Argument #0->|./example|
Argument #1->|param1|
Argument #2->|param2|
Argument #3->|3|
Argument #4->|4|
Argument #5->|5|
Argument #6->|Hello world|
```

Punteros a función

- Ofrecen mayor flexibilidad en el código

```
/* 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