

# PROGRAMACIÓN EN C

impartido por: Raúl Izquierdo Castañedo.

redactado por: Miguel Herrero Obeso (<http://www.miguelherrero.cjb.net>)

actualizado el: 09/02/2004

## PRIMER PROGRAMA

```
#include <stdio.h> /* ejemplo */
int main (void){
    float base, altura;
    printf("teclea base y altura:");
    scanf("%f %f",&base,&altura);
    printf("El área es %f \n",base * altura);
    return 1;
}
```

La primera línea incluye el fichero de funciones de salida estándar y muestra como se crean los comentarios en C, esto es usando /\* aquí escribes el comentario \*/. La cuarta línea imprime una cadena pasada como parámetro con printf. La quinta línea lee dos argumentos con scanf, su sintaxis es esta:

```
scanf("cadena de control",argumentos);
```

se escribe % en la cadena de control por cada argumento que se va a pedir y detrás se pone el tipo del dato (%d si entero, %s si cadena, %f si real, %c si carácter y %p si puntero). Luego se hace corresponder los valores % con los argumentos. En la sexta línea printf imprime el área del rectángulo utilizando de un modo similar los % y terminando con \n que es una secuencia de escape. Las secuencias de escape más frecuentes son:

```
\n salto de línea
\t tabulador
\" comilla doble (para distinguirlas de las comillas de inicio y fin de cadena)
```

En la línea dos se puede sustituir int por void en el método main para que no tenga valor de retorno. Entonces hay que eliminar la línea return 1; para que funcione. La mayoría de compiladores acepta el cambio pero no todos.

## Tipos de datos

Son prácticamente igual que en Java. Existen dos tipos: simples y compuestos. Los tipos simples son int, char y float. Los tipos int y char son tratados ambos como enteros y se les aplica los modificadores signed y unsigned (para que almacenen, o no, el signo). Para los char cada compilador tiene un trato "por defecto", unos lo tratan con signo y otros sin él por lo que usando signed y unsigned en el código hacen que éste sea más portable.

Para el tipo int está estandarizado en todos los compiladores que se trate como signed int. También se puede hacer uso de los modificadores short y long para variar el tamaño de los tipos como se ve en el siguiente ejemplo:

```
unsigned long int;
```

El tamaño de los tipos depende del compilador así que se establecieron las siguientes relaciones para que se mantuviera una consistencia:

```
int >= 16 bits
short <= int <= long
```

long >= 32 bits

Y para los reales:

float >= 32 bits  
double >= 64 bits  
long double >= 80 bits

En C no existe el tipo boolean por lo que se suele utilizar por convenio el int. De esta forma si el entero vale 0 es falso y si es distinto de 0 es cierto.

Para esta expresión:

```
c=5/2;
```

si c es un real, almacenaría 2 y no 2.5. C realiza primero la división y comprueba que los dos números son enteros luego hace una división entera y el resultado lo convierte a real y lo almacena en c. Para forzar la división real se puede escribir:

```
c=5/2.0
```

También se puede hacer uso del ahormado o casting que funciona igual que en Java:

(tipo) expresión

luego la expresión anterior, utilizando ahormado resultaría así:

```
c=5/(float)2
```

También hay que tener en cuenta que en C no hay redondeo implícito por lo que si:

```
int a;  
a=5.9;
```

a no almacenaría 6 sino 5. Se trunca el número, no se redondea.

## Bucles

Todos los bucles tienen la misma sintaxis que los de Java salvo por una pequeña diferencia. Como ya hemos dicho en C no hay booleanos por lo que las sentencias a verificar por los bucles son expresiones enteras. Por ejemplo se realizan comparaciones del tipo `a>2`. Por ejemplo:

```
if (a+b) { }
```

se cumple cuando `a+b` sea distinto de 0.

Hay un error frecuente en este tipo de expresiones:

```
if (a=b) { }
```

Ésta expresión no compara la igualdad porque no usa el operador de igualdad `==` por lo que se trata de una asignación. La variable `a` toma el valor de `b` y comprueba si `a` es cero o distinto de él para entrar o no en el bucle.

## Entrada y salida estándar

La función `getchar` lee un carácter de la entrada estándar.  
La función `putchar` escribe un carácter en la salida estándar.

ejemplo:

```
int c;
while ((c=getchar())!=-1)
    putchar(c);
```

Como se puede ver, la función `getchar` devuelve un entero para poder almacenar los caracteres especiales aparte de las letras del alfabeto tradicional.

## Constantes

5 es int.  
5.4 es double.  
5.4f es real.

Por lo demás la definición de constantes es igual que en Java.

## Funciones

```
<tipo> nombre (parámetros) {
    [variables locales]
    [sentencias]
    return
}
```

Si no hay parámetros se pone **void** como parámetro. Todos los parámetros se pasan por valor (salvo que sean punteros). Para invocar una función debe estar definida antes de la llamada, o estarlo su prototipo.

El **prototipo** es una declaración normal pero sin las sentencias.

## Modos de almacenamientos

Variables automáticas (locales): se definen en un bloque. Se crean al entrar al bloque y se destruyen al salir.

Variables externas (globales): se definen fuera de un bloque y es visible para todas las funciones por debajo de ella en el mismo fichero fuente.

Las variables locales tienen preferencia sobre las globales.

Para acceder a una variable global de otro fichero se usa el modificador **extern**:

```
extern int i;
```

Para acceder a una función de otro fichero se escribe el prototipo. Aunque se puede escribir también **extern** es algo redundante, con el prototipo es más que suficiente.

A las variables se les puede poner el identificador **static**, que hace que las variables locales no se destruyan (por lo que al entrar a una función por segunda vez se 'acuerda' del valor).

Si usas **static** en variables globales las proteges de que otros ficheros la usen (inutiliza el **extern**).

## El preprocesador de C

Procesa las instrucciones específicas para él. Las instrucciones para él comienzan por el carácter #.

```
#include <nombrefichero>
```

incluye 'nombrefichero' en el fichero actual (copia el fichero en el actual).

Ejemplo:

z.txt contiene int a;

```
#include <z.txt>
void f(void) {
    printf("hola");
}
```

es equivalente a

```
int a;
void f(void) {
    printf("hola");
}
```

nombrefichero se puede poner entre <> o entre "" y su significado es:

<nombrefichero>: busca nombrefichero en el directorio estándar

"nombrefichero": busca nombre fichero en el directorio del programa y si no lo encuentra, en el directorio estándar.

#define identificador cadena

sustituye el identificador por cadena cada vez que lo encuentre en el código.

ejemplos:

```
#define PI 3.1416f
#define WRITE printf
```

parámetros:

```
#define id(arg1,...,argn) cadena
```

sustituye id por cadena pero usando los parámetros al sustituir.

ejemplos:

```
#define DOBLE(x) x+x

int a = DOBLE(2);
```

el preprocesador lo sustituye por

```
int a = 2 + 2;
```

Ejercicio:

```

#define CUADRADO(x) x*x

int main(void){
    int a=CUADRADO(2);
    int b=CUADRADO(a+2);
    int c=8/CUADRADO(2);
    int d=CUADRADO(++a);
}

```

¿Valor de a, b, c y d?

$a=2*2=4$

$b=a+2*a+2$  (tiene preferencia el producto) =  $a+2a+2=4+8+2=14$ ;

$c=8/2*2$  (tiene preferencia el cociente) = 8

$d=++a * ++a =5*6=30$

Para evitar problemas, al definir parámetros, SIEMPRE entre paréntesis cada parámetro y toda la expresión entre paréntesis:

$((x)*(x))$

Nunca pasar como argumento una función (o lo que sea) que cause efectos laterales.

Otras directivas son:

`#if`, `#else`, `#ifdef`, `#ifndef`, `#undef`, `#elseif`, `#endif`

## Arrays

<tipo> nombre[tamaño]

```
int v[10];
```

```
float x[10][20];
```

El tamaño debe ser constante, no vale usar variables porque el compilador necesita saberlo en tiempo de compilación.

C no comprueba límites de arrays (se puede escribir en código, datos, o lo que haya en memoria).

C no inicializa a cero los elementos del array.

Los arrays no pueden ser el retorno de una función, pero sí pueden ser parámetros:

```

void inicializa (int v[], int lenght) {
    int a;
    for (a=0; a<lenght, a++)
        v[a]=0;
}

```

Los arrays se pasan por referencia.

En la definición del array se debe indicar el tamaño, pero al pasarlo como parámetro no hace falta indicarlo; los corchetes sirven para señalar que es un array.

## Strings

En C los strings son arrays de caracteres cuyo último elemento es el número cero.

```
char nombre[5]={'H','o','l','a',0};
```

Para evitar esto al introducir textos largos, C permite hacer lo siguiente:

```
char nombre[]="Hola";
```

y el compilador se encarga de rellenar el array y de poner al final el cero.

Ejemplo:

```
int len(char s[]){ //función que cuenta los caracteres de una cadena.  
int i=0;  
while (s[i]!=0)  
    i++;  
return i;
```

y para usarlo escribiremos:

```
len("Hola");
```

## Punteros

**&** es el operador de dirección. Sirve para conocer la dirección de una variable.

Ejemplo: `&a`

Los punteros son variables que guardan la dirección de otra variable.

Para declarar un puntero la sintaxis es la siguiente:

```
int *p;  
float *f;
```

Si no sabemos el tipo al que va a apuntar una variable podemos hacer...

```
void *g;
```

Para darle valor a un puntero escribimos:

```
p=&a;
```

### Acceso a memoria mediante punteros

El operador `*` aplicado a un puntero, nos da acceso al valor que haya en la dirección del puntero (tanto para leer como para escribir).

Ejemplo:

```
int *p;  
int a=5;  
p=&a;
```

ahora `p` apunta a la variable `a`.

```
printf("%p %d",&a, a) //muestra por pantalla 800 5  
printf("%p %d",p, *p) //muestra por pantalla 800 5  
  
*p=8;
```

```
printf( "%d", a)
```

muestra por pantalla 8

## Operaciones con punteros

-Suma y resta de constante:

suponemos que p apunta a un entero en la posición 800

```
p = p+1;
```

esto no pone p a 801 sino que avanza una unidad del tipo al que esté apuntando el puntero. En este caso avanza un entero (4 bytes) por lo que p vale 804.

Con los punteros void no se puede operar (no se sabe de qué tipo son)

Para asignar:

```
double *g;  
g=1000; //esto da error pues 1000 es un puntero a int.  
//pero se puede hacer un cast  
g=(double *)1000
```

-diferencia de punteros:

```
int *p;  
int *q;  
p=&a; //10  
q=&b; //18
```

entonces q-p devuelve dos (hay 8 bytes que son dos enteros)

p-q devolvería -2

-Incremento y decremento de punteros:

```
p==q  
p>q  
p<q  
p!=q
```

ejemplo:

```
int a=8, b=10;  
int *q, *p;  
p=&a;  
q=&b;  
printf( "%d", p==q); // falso  
printf( "%d", *p==*q); //falso (8==10);
```

ejemplo: inicializar un array a cero (con punteros)

versión 1:

```
int v[10];  
int i;  
int *p=&v[0];
```

```
for (i=0;i<10;i++)
    *(p+i)=0;
```

versión 2:

```
int v[10];
int *p=&v[0];
while (p<=&v[9]){
    *p=0; //estas dos instrucciones se pueden sustituir por *p+=0;
    p++;
}
```

Ejemplo:

```
int a=5;
int *p=&a;
p=p+1;
*p=6; //esto compila pero es peligroso porque no se conoce qué almacena
esta
//posicion de memoria
```

Ejemplo 2:

```
int *p;
*p=8; //apunta a ;Dios sabe que trozo de memoria!
```

Hay que tener cuidado al moverse con punteros y limitar el movimiento en el espacio de memoria reservado.

## Relaciones entre arrays y punteros

El nombre del array es un puntero

```
int v[10];
```

v es lo mismo que `&v[0]`

es un puntero constante porque podemos leer su dirección pero no modificarla.

Los arrays pueden usar notación de punteros

`*v` es lo mismo que `v[0]`

de hecho se puede acceder al array así

`*(v+n)` siendo n la posición del array a consultar / modificar

los punteros pueden usar notación de array

```
int v[10];
int *p;
p=v;
```

`p[0]` equivale a `*p`

`p[1]` a `*(p+1)`

Ejercicio: Hacer dos formas de inicializar un array con las relaciones anteriores

## Arrays con punteros

```
int *v[10]
```

v es un array de 10 punteros a enteros

## Paso de punteros por parámetros

ejemplo:

```
void f(int *p){
    *p=5;
}

void main(void) {
    int a=8, b=10;
    f(&a);
    f(&b);
    printf("%d %d", a,b);
}
```

mostraría por pantalla : 5 5

ejemplo 2:

```
void intercambia(int *a, int *b) {
    int inter;
    inter = *a;
    *a = *b;
    *b = inter;
}
```

y la llamada se realizaría así: f(&a,&b);

## Memoria dinámica

```
void *malloc (long tamaño);
```

busca memoria libre. Tamaño es en bytes. Devuelve la dirección del primer byte.

Nota: void \* es una declaración a un puntero de tipo indeterminado.

```
void free (void *bloque);
```

libera el bloque de memoria asignado al puntero pasado como parámetro.

ejemplo: Simular la creación de arrays en Java asignando el tamaño en tiempo de ejecución.

Nota: el puntero null en C no existe, se usa el número 0.

```
int main (void) {
    int tamaño,i;
    int *p1;
    printf("Introduce el tamaño:");
    scanf("%d",&tamaño);
    p1 = (int*) malloc(tamaño*sizeof(int)); //conversión de puntero y
    //tamaño de tipo
    for (i=0;i<tamaño;i++)
        *(p1+i)=i+1;
    free(p1);
}
```

```
}
```

## Entrada y salida (funciones auxiliares)

```
char *gets(char *);
```

Lee una línea de la entrada estándar. Si hay error devuelve 0.

```
char *puts(char *);
```

Escribe una línea en la salida estándar.

```
int strlen(char *cadena);
```

Devuelve el número de caracteres de la cadena.

ejemplo:

```
char nombre[40] = "Pepe";  
printf ("%d %d", sizeof(nombre), strlen(nombre));
```

esto imprime: 40 4 (Sizeof da el tamaño total del array y strlen la longitud de la cadena)

ejemplo 2:

```
char linea[50];  
gets(linea);  
linea = linea + "FIN";  
/* la linea anterior suma la direccion de linea con la direccion de fin.  
Sumar dos punteros no tiene sentido luego da un error de compilacion */
```

```
void strcat (char *dest, char *origen);
```

Concatena dos cadenas.

ejemplo:

```
strcat(linea, "Final");  
  
int strcmp (char *c1, char *c2);
```

Compara dos cadenas. Devuelve 0 si las cadenas son iguales, mayor que 0 si c1>c2 y menor que 0 si c1<c2. La comparación es sensible a mayúsculas.

ejemplo:

```
char linea[50];  
puts("Adivina mi nombre:");  
gets(linea);  
while (strcmp(linea, "pepe")!=0){  
    puts("Noooo");  
    gets(linea);  
}
```

ejemplo 2:

```
char linea[80];
```

```
char copia[80];
gets(linea);
copia=linea;
```

copia es un puntero constante y no se puede cambiar la dirección a la que apunta.

```
void strcpy (char *destino, char *origen);
```

copia la cadena origen en la dirección destino.

ejercicio: implementar strcpy y strcat nosotros mismos.

strcpy modo 1:

```
void strcpy (char * destino, char *origen){
    int i=0;
    while (i<=strlen(origen)){
        destino[i]=origen[i];
        i++;
    }
}
```

strcpy modo 2:

```
void strcpy (char * destino, char *origen){
    while(*destino++=*origen++);
}
```

strcat:

```
void strcat (char *destino, char *origen){
    destino=destino+strlen(dat);
    while(*destinio++=*origen++);
}
```

## Estructuras

Una estructura es una agrupación de variables de distintos tipos.

```
struct Persona{
    char nombre[30];
    int edad;
}; //acordarse de poner punto y coma al final de la declaración
```

declaración de variables de tipo estructura:

```
struct Persona pepe;
```

para utilizar los campos de la estructura:

```
pepe.edad=2;
strcpy(pepe.nombre, "Pepe");
```

también se pueden igualar estructuras:

```
juan=pepe;
```

y se pueden usar arrays de ellas:

```

struct Persona v[10];
v[1].edad=8;
strcpy(v[1].nombre, "nombre");

```

Las estructuras pueden ser parámetros y tipos de retorno.

ejemplo:

```

void imprime(struct Persona pers){
    printf("%s %d",pers.nombre,pers.edad);
} //se pasa por valor, es decir, los campos del original no se modifican

struct Persona leeDatos(void){
    struct Persona aux;
    scanf("%s %d",aux.nombre,&aux.edad);
    return aux;
}

```

se puede crear un puntero a una estructura (para poder pasarlo por dirección):

```

struct Persona *p;
p = &pepe;

```

en vez de \*p.edad, para acceder a los campos a través de un puntero se escribe así:

```

p->edad=8;

```

esta sentencia modifica el valor real de la variable.

ejemplo:

```

strcpy(p->nombre, "pepe");

```

también se pueden declarar como si fuera un array, teniendo en cuenta el orden de los campos:

```

struct Persona pepe = {"pepe", 4};

```

## Uniones

son iguales a las estructuras salvo que en las uniones, se reserva memoria para la variable del mayor tipo. Solo puede almacenar uno de los campos porque en cuanto se almacena otro, se escribe encima de la variable anterior. Esto ocupa mucho menos que una estructura pero solo almacena una variable. Es muy útil cuando no sabemos exactamente el tipo de la variable que tenemos que almacenar.

Por lo demás la declaración y creación de uniones es igual a la de estructuras.

## Paso de parámetros por el Main

```

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

```

argc: contador de parámetros. En la cuenta se incluye el nombre del programa.

```

prueba.exe fichero.txt -r // 3 parámetros

```

argv: array de punteros a cadenas que contienen los parámetros.

ejemplo:

```

int main(int argc, char *argv[]){
    int i;
    for(i = 0, i < argc, i++)
        puts(argv[i]);
    return 0;
}

```

## Ficheros

```
FILE * fopen(char * nombre, char * modo);
```

Sirve para abrir un fichero. Devuelve un puntero a FILE (una estructura). Si hay algún error se devuelve NULL. Los modos son

Modo Texto:

r: lectura, r+: lectura y escritura

w: escritura, w+: escritura y lectura

a: añadir al fichero, a+: añadir y lectura.

Modo Binario:

rb: lectura, rb+: lectura y escritura

wb: escritura, wb+: escritura y lectura

ab: añadir al fichero, ab+: añadir y lectura.

ejemplo:

```

FILE *f;
f = fopen("texto.txt", "w");
...
...
fclose(f);

```

ejemplo 2:

```

FILE *f;
if ((f=fopen("t.txt", "w"))==NULL)
    printf("Error");
else{
    //sentencias
    fclose(f);
}

int fgetc(FILE * fichero);

```

Lee un carácter del fichero. Devuelve el carácter o -1 si ha llegado al final de fichero.

```
void fputc(int c, FILE *fichero);
```

Escribe un carácter.

Hay tres punteros a FILE predefinidos en C:

stdin: entrada estándar

stdout: salida estándar

stderr: salida de error estándar

Nota: EOF es una constante que vale -1

ejemplo:

```
int c;
FILE *f1, *f2;
f1 = fopen("t1.txt", "r");
f2 = fopen("t2.txt", "w");
while ((c = getc(f1)) != -1)
    fputc(c, f2);
fclose(f1);
fclose(f2);
```

```
int fprintf(FILE *f, char * control, ...);
```

Hace lo mismo que printf pero escribiendo en fichero (de hecho printf llama a esta función pasándole la salida estándar).

```
int fscanf(FILE *f, char * control, ...);
```

Hace lo mismo que scanf pero leyendo desde fichero.

ejemplo:

```
int i=5;
float f = 34.56;
char c = 'a';
FILE *out;
out = fopen("guardar.txt", "w");
fprintf(out, "%d %f %c \n", i, f, c); //resultado 5 34.56 a
```

ejemplo 2:

```
FILE *in;
int d;
float f;
char c;
in = fopen("guardar.txt", "r");
fscanf(in, "%d %f %c", &d, &f, &c);
fclose(in);

char * fgets(char * linea, int limite, FILE *);
```

línea: dirección de memoria donde guarda la cadena.

límite: controla los caracteres máximos (el tope es límite-1, para dejar un espacio al cero).

devuelve NULL si no hay más líneas.

ejemplo:

```
//abrir ficheros
char linea[80];
while(fgets(linea, sizeof(linea), f1) != NULL)
    fputs(linea, f2);
// cerrar ficheros

long ftell(FILE *);
```

Dice la posición del fichero en que se encuentra la cabeza de lectura/escritura. Devuelve bytes.

```
int fseek(FILE *, long desplazamiento, int origen);
```

Sitúa la cabeza de lectura/escritura en la posición deseada. El desplazamiento es en bytes. Puede ser positivo o negativo.

origen: el origen del desplazamiento. Hay tres valores predefinidos:

SEEK\_SET que representa el inicio del fichero.

SEEK\_CUR que representa la posición de la cabeza de lectura/escritura.

SEEK\_END que representa el final del fichero.

ejemplo:

```
fseek(f,10,SEEK_SET); //se sitúa en la posición 11 (ya que aquí se cuenta
//como en los arrays.
```

```
fseek(f,-10,SEEK_CUR); //se vuelve al principio
fseek(f,-40,SEEK_END); //se retrocede 40 valores desde el final.
```

ejercicio: implementar una función que retorne el tamaño de un fichero pasado por parámetro (no hay función estándar para esto)

```
long size;
fseek(f,0,SEEK_END)M
size = ftell(f);
```

### Escribir variables en un fichero

```
int fwrite(void *buffer, int bytesElemento, int numElementos, FILE *);
```

Escribe los datos en un fichero.

buffer: la posición de memoria donde se encuentra el primer dato.

bytesElemento: los bytes que ocupa un elemento.

numElementos: número de elementos a copiar.

FILE \*: fichero donde se quiere escribir.

```
int fread(void *buffer, int bytesElemento, int numElementos, FILE *);
```

Lee datos de un fichero.

buffer: la posición de memoria donde se desea guardar el primer dato.

bytesElemento: los bytes que ocupa un elemento.

numElementos: número de elementos a copiar.

FILE \*: fichero de donde se quiere leer.

ejemplo:

```
//abrir fichero
int v[80];
fwrite(v,sizeof(int),80,f); //otra forma es fwrite(v,sizeof(v),1,f);
//cerrar fichero
```

ejemplo 2:

```
//abrir fichero
int v[80];
fread(v,sizeof(int),80,f);
//cerrar fichero
```

ejemplo 3:

```

struct Persona pepe;
fwrite(&pepe, sizeof(pepe), 1, f);

```

## Interpretación de declaraciones

```

punteros *      int *a;
arrays [] int a[8];
funciones ()   int a();

```

La prioridad de un modificador es mayor cuanto más cerca está del nombre de la variable. Los corchetes y paréntesis tienen mayor prioridad que el asterisco. Esta precencia se puede cambiar utilizando paréntesis.

() : función que devuelve ...  
 \* : puntero a...  
 [] : array de ...

ejemplos:

```

int (*a)[10]; puntero a array de 10 enteros
int *a[3][4]; array de tres punteros a array de cuatro enteros
int (*a)[3][4]; puntero a un array 3x4 de enteros
char (*a)(); puntero a función que devuelve char
char *a[3](); array de tres puntero a función que devuelve char
int *a[10]; array de 10 punteros a entero
int *a(); función que devuelve un puntero a entero.

```

## Definición de tipos personalizados

typedef <declaracion>;

```

typedef int entero; //crea un tipo entero que es un int
typedef int *pInt; //crea una variable puntero a entero
typedef int (*pfunc)(struct Persona); //puntero a función (con
//estructura de parametro) que devuelve int

```

ejemplo: PAS puntero a un array de cinco estructuras persona.

```

typedef struct Persona (*PAS)[5];

```

ejercicio: dado una entrada de palabras, contar las veces que se repite cada una.

hola que tal tal hola hola >>> hola = 3, que = 1, tal = 2

```

struct Palabra {
    char texto[10];
    int num;
    struct Palabra *sig;
};

struct Palabra *cab = 0;

void inserta (char *palabra){
    struct Palabra *p = cab;
    while (p!=0){
        if (strcmp(palabra, p->texto) == 0){
            p->contador++;
            return;
        }
    }
}

```

```
    }  
    p = (struct Palabra *) malloc (sizeof(struct Palabra);  
    strcpy(p->texto,palabra);  
    p->contador = 1;  
    p->sig = cab;  
    cab = p;  
  }  
}
```

[www.miguelherrero.cjb.net](http://www.miguelherrero.cjb.net)