Introducción a C

Grupo de Sistemas y Comunicaciones

gsyc-profes@gsyc.es

Febrero 2008

C

Características:

- Programación imperativa estructurada.
- Relativamente de “bajo nivel”.
- Lenguaje simple, la funcionalidad está en las bibliotecas.
- Utilización natural de las funciones primitivas del sistema.
- Básicamente maneja números, caracteres y direcciones de memoria.
- No tiene tipado fuerte.

El programa más sencillo

```
/* Esto es un comentario */

#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    exit(EXIT_SUCCESS);
}
```


Compilación de un programa en C

Compilador más habitual: `gcc`

Uso: `gcc` *`[opciones] fichero-fuente`*

`-c`	Solo compila (no enlaza)
`-o fichero`	Nombre del fichero de salida
`-Wall -Wshadow`	Activa avisos
`-g`	Activa información de depuración

```
$ gcc -g -Wall -c hello.c
$ gcc -o hello hello.o
$ ./hello
Hello, world!
$ _
```

Compilación de un programa con módulos

- En general, un programa se compone de varios ficheros fuente:
  - `entrada.c`
  - `calculo.c`
  - `salida.c`
- Primero hay que compilar primero cada uno de los ficheros:

```
$ gcc -g -Wall -c entrada.c
$ gcc -g -Wall -c calculo.c
$ gcc -g -Wall -c salida.c
```

- Y luego enlazarlos para generar el ejecutable:

```
$ gcc -o programa entrada.o calculo.o salida.o
```

- Si todo ha ido bien, podemos ejecutarlo:

```
$ ./programa
```


Directivas de preprocesador

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

- Necesarias para utilizar funciones en el programa, como `printf()` y `exit()`.
- Hay que mirar la página de manual de cada función que vayamos a usar, para saber qué líneas "`#include <...>`" tenemos que poner.
- Las más habituales son: "`<stdio.h>`", "`<stdlib.h>`", "`<string.h>`", "`<unistd.h>`", "`<fcntl.h>`"...

Uso del manual

- Para buscar una función en el manual:

  ```
  $ man N título
  ```

  Busca "*título*" en la sección "*N*" del manual.
- Hay varias secciones distintas; las que nos interesan aquí son:
  - 1. Comandos de usuario
  - 2. Llamadas al sistema
  - 3. Funciones de librería
- Para obtener información acerca de una sección en concreto:

  ```
  $ man N intro
  ```
- Para buscar páginas de manual que traten de una palabra en concreto:

  ```
  $ apropos palabra
  ```


Punto de entrada en el programa

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

- Función con un nombre especial, "`main`",
- Esta función devuelve un valor de tipo entero, y recibe dos parámetros.
- Siempre se declara de esa manera.
- Estas líneas forman la *cabecera*; el *cuerpo* (contenido) es lo que hay entre las llaves `{ ... }`

Sentencias

Una sentencia es típicamente una expresión o una llamada a una función, terminada por el carácter "`;`".

```
printf("Hello, world!\n");
```

- Llamada a la función "`printf()`", que imprime un texto por pantalla, con un parámetro, "`"Hello, world!\n"`"

```
exit(EXIT_SUCCESS);
```

- Llamada a la función "`exit()`", que termina el programa, con el parámetro "`EXIT_SUCCESS`"

Tipos de datos fundamentales

Nombre	Descripción	Ejemplo
Tipos enteros		
`char`	Carácter	`97, 'a'`
`int`	Entero	`32769`
Tipos reales		
`float`	Precisión simple	`3.1416`
`double`	Precisión doble	`3.141592653589`
Otros		
`void`	Conjunto vacío	

Declaración de una variable

tipo indentificador[, identificador]... ;

```
int contador;

int
main(int argc, char *argv[]){
    int dia, mes;
    char c, d;

    /* ... */
}
```


Uso de una variable

```
int contador = 1;

int
main(int argc, char *argv[]){
    int dia = 27, mes = 2;
    char c, d;

    c = 97;
    d = 'Z';
    /* ... */
    printf("Estamos en el dia %d\n", dia);
}
```

Tamaños de variables y tipos de datos

- C no define tamaños de tipos concretos
- Se debe usar siempre “`sizeof`”
- “sizeof(tipo)” me da el tamaño de un tipo
- “sizeof variable” me da el tamaño de una variable
- Si meto un tipo más grande en uno más pequeño, hay problemas.

printf()

```
#include <stdio.h>
printf(formato, arg, arg...);
```

formato es una cadena de caracteres `"como esta"`, que puede contener:

`\n`	Fin de línea
%d	El siguiente parámetro (lo muestra como un número decimal)
%c	El siguiente parámetro (lo muestra como un carácter)
...	...

Ejemplos:

```
printf("decimal = %d, caracter = %c\n", 100, 120);
printf("decimal = %d, caracter = %c\n", 'a', 'z');
```

Funciones

Definición y declaración

```
tipo-resultado
nombre-función([parámetros]) {
    declaraciones de variables locales;
    sentencias;
    [return [expresión]];
}
```


Paso de parámetros

```
#include <stdio.h>

void
calcula(int x, int y) {
    printf("resultado = %d\n", x * y);
}

int
main(int argc, char *argv[]) {
    calcula(35, 47);
    return 0;
}
```

Operadores

Operadores aritméticos

`+`	Suma.
`-`	Resta.
`*`	Multiplicación.
`/`	División.
`%`	Módulo (resto de la división entera).
`( ... )`	Agrupación de expresiones.

Las operaciones con enteros dan siempre un resultado entero

Operadores aritméticos

```
int
main(int argc, char *argv[]) {
    int c, a = 10, b = 3;

    c = (a / b) * b;  /* 9 */
    printf("(%d / %d) * %d = %d\n", a, b, b, c);
    exit(EXIT_SUCCESS);
}
```

Operadores de asignación

++	Incremento
--	Decremento
=	Asignación simple
+=	Suma y asignación
-=	Resta y asignación
*=	Multiplicación y asignación
/=	División y asignación
%=	Módulo y asignación

Operadores de asignación

```
int
main(int argc, char *argv[]) {
    int x = 10, n = 0, i = 0;

    x++;                /* Incrementa x en 1 */
    ++x;                /* Incrementa x en 1 */
    printf("%d\n", --n); /* Decrementa y luego asigna */
    printf("%d\n", n--); /* Asigna y luego decrementa */
    i += 2;             /* i = i + 2 */
    return 0;
}
```

Operadores de relación

a < b	“a” menor que “b”
a > b	“a” mayor que “b”
a <= b	“a” menor o igual que “b”
a >= b	“a” mayor o igual que “b”
a != b	“a” distinto de “b”
a == b	“a” igual a “b”

Operadores lógicos

a && b	AND. Cierto si “a” y “b” son ciertos
a || b	OR. Cierto si “a” o “b” son ciertos
!a	NOT. Cierto si “a” es falso, falso si es cierto

Operadores de relación, operadores lógicos

En C no existe el tipo “booleano”, todas las comparaciones devuelven un número entero (**int**);

- **Falso**, un valor igual a 0
- **Verdadero**, cualquier otro valor

Por lo tanto, estos operadores devuelven 0 o un valor distinto de 0 (normalmente 1), dependiendo de si se cumple o no la condición.

```
printf("Resultado=%d\n", (5>3 && 7<=10));
```

Esta línea imprime un 1.

Otros Operadores

&	(unario) Dirección-de. Da la dirección de su operando
*	(unario) Indirección. Acceso a un valor, teniendo su dirección
&	AND de bits
|	OR de bits
^	XOR de bits
~	NOT de bits
<<	Desplazamiento binario a la izquierda
>>	Desplazamiento binario a la derecha
?:	Operador ternario

Precedencia de operadores

( ) [ ] -> .	izquierda a derecha
! ++ -- * & ~ (unarios)	derecha a izquierda
* / %	izquierda a derecha
+ -	izquierda a derecha
< <= > >=	izquierda a derecha
== !=	izquierda a derecha
& ^ |	izquierda a derecha
&&	izquierda a derecha
||	izquierda a derecha
= += -= *= /= %= &= ...	derecha a izquierda
,	izquierda a derecha

Sentencias de control: if

```
if (expresión)
  sentencia;
```

```
if (expresión) {
  sentencia1;
  sentencia2;
  ...
}
```

Si "*expresión*" se evalúa como verdadera (esto es, distinta de 0), se ejecuta la sentencia (o sentencias, en el caso de un bloque encerrado entre llaves).

Sentencias de control: else

Solo se puede usar justo a continuación de un "`if`":

```
if (expresión)
  sentencia1;
else
  sentencia2;
```

```
if (expresión) {
  sentencia1;
  ...
} else {
  sentencia2;
  ...
}
```

Si "*expresión*" es verdadera, se ejecuta "*sentencia1*". En otro caso, se ejecuta "*sentencia2*".

else if

```
int a = 10, b = 20, c = 0, menor;
if (a < b) {
  if (a < c) {
    menor = a;
  }
  else {
    menor = c;
  }
} else if (b < c) {
  menor = b;
} else {
  menor = c;
}
printf("Menor = %d", menor);
```


while

```
while (expresión) {
  sentencias;
}
```

1. Se evalúa la "*expresión*". Si es verdadera (distinta de 0):
   - Se ejecutan las "*sentencias*".
   - Vuelve al punto 1.

Resumiendo, un "`while`" es un bucle que se repite siempre que la condición siga siendo verdadera.

Un bucle infinito se puede conseguir con "`while (1)`".

Se puede usar la orden "`break`" en cualquier momento para salir del bucle.

for

```
for ( inicialización ; condición ; actualización ) {
  sentencias;
}
```

1. Se ejecuta la “*inicialización*”
2. Se evalúa la “*condición*”; si es verdadera:
   - Se ejecutan las “*sentencias*”
   - Se ejecuta la “*actualización*”
   - Vuelve al punto 2.

Un ejemplo de un bucle que se repite 10 veces, usando la variable “i” (desde i=0 hasta i=9) sería:

```
for (i=0; i<10; i++) { /* ... */ }
```


Estructuras

struct nombre_estructura {
 /* Declaración de los campos */
};

```
struct alumno {
  int dia,mes,anno; /* nacimiento */
  int dni;
  char letra_nif;
};

struct alumno var;

printf("El año de nacimiento es %d\n", var.anno);
```

typedef: redefinición de tipos

Uso: "*`typedef` tipo_actual nuevo_nombre;*"

Lo utilizaremos para renombrar un tipo por otro.

Ejemplo:

```
typedef int entero;
typedef char byte;

entero a,b;
byte c,d;
```


Uso de typedef con estructuras

Es el uso más habitual de typedef.

Para usar una estructura, siempre tenemos que incluir la palabra "`struct`", a menos que se renombre el tipo. Ejemplo:

```
typedef struct alumno talumno;

typedef struct {
  int a, b;
} t2enteros;

talumno i,j;

t2enteros x,y;
```

enum

Es un tipo prefefinido, pero nosotros lo usaremos para definir constantes:

```
enum {
  INICIO=0,
  MITAD,
  FINAL,
  TAM=100
};

/* ... */

for (i=INICIO; i<TAM; i++) {
  /* ... */
}
```


Punteros

En C, toda variable se almacena en una dirección de memoria. Podemos saber cuál es esa dirección con el operador unario “&”:

`int a`	es un entero
`&a`	dirección de memoria donde se encuentra a

Ejemplo:

```
int a = 3;

printf("a = %d\n", a);
printf("&a = %d\n", &a);
```

Punteros (2)

- Hay tipos especiales que se usan para almacenar una dirección de memoria: los "*punteros*", y se declaran con "*tipo* ***** *variable*":

  `int *p` p es un "puntero a un entero" (puede contener la dirección de variables de tipo `int`)

  `p = &a` le asignamos la dirección de la variable a

- Al igual que tenemos el operador "`&`" para acceder a la dirección en la que está almacenada una variable, hay otro operador complementario, el "`*`", que se usa para acceder al *contenido* de una dirección de memoria.

  `*p` es el contenido de la dirección p, esto es, a

Punteros (3)

- El uso de punteros es la única manera de conseguir un "paso por referencia" en C:

```
#include <stdio.h>
void
intercambio(int * x, int * y) {
    int z = *x;
    *x = *y;
    *y = z;
}
/* ... */
    int a = 20, b = 30;
    intercambio(&a, &b);   /* Paso por referencia */
/* ... */
```

Punteros (4)

- Un puntero es la manera más cómoda de pasarle una estructura a una función:

```
struct ficha_alumno {
  int dni;
  char letra;
};
struct ficha_alumno var1, var2;

void funcion(struct ficha_alumno * ficha) {
  printf("%d\t%c\n", ficha->dni, ficha->letra);
}
/* ... */
  funcion(&var1);
/* ... */
```


Aritmética de punteros

Como hemos dicho, un puntero es una dirección de memoria, y C permite hacer algunas operaciones aritméticas con ellos:

- Suma y resta de un puntero y un entero

  Si **p** es un puntero y **i** es un entero, **(p+i)** es otro puntero del mismo tipo que **p**, y que apunta a una zona de memoria que está **i** posiciones por delante de **p**

- Resta entre punteros del mismo tipo

  Si **p** y **q** son dos punteros del mismo tipo, **(q-p)** es un entero que indica el número de valores de ese tipo que se pueden almacenar entre **p** y **q**

Estas son las únicas operaciones aritméticas que se pueden hacer con punteros.

malloc()

- Cuando declaramos un puntero de un tipo determinado, está preparado para apuntar a cualquier elemento de ese tipo, pero no está inicializado. No se puede usar directamente, primero tenemos que inicializarlo con la dirección de un elemento de ese tipo.
- Podemos reservar memoria para almacenar elementos de un tipo determinado con `malloc()`:

```
int *p;
p = malloc(sizeof(int));
*p = 3;
```

- `malloc()` reserva una zona de memoria de un tamaño determinado (en este caso, el tamaño que ocupa un **int**) y devuelve un puntero a esa zona.

- Una zona de memoria reservada con `malloc()` se debe liberar con `free()` cuando no vayamos a seguir usándola:

```
int *p;
p = malloc(sizeof(int));
/* ... */
*p = 58;
/* ... */
free(p);
```

- El valor `NULL` se utiliza para indicar que un puntero no apunta a ningún valor.

Atención: NUNCA se debe usar el contenido de un puntero (`*p`) antes de haberlo inicializado: dándole la dirección de una variable (`p = &i;`) o pidiendo memoria al sistema (`p = malloc(...);`).

Ejemplo: lista enlazada

```
struct nodo {
  int value;
  struct nodo *next;
};
struct nodo *list;
```

Ejemplo: árbol binario

```
struct nodo {
  int value;
  struct nodo *left, *right;
};
struct nodo *root;
```


Ámbito de una variable

Podemos almacenar contenido en tres sitios distintos:

- Variables globales

  El contenido dura hasta que termine el programa

- Variables locales

  El contenido dura hasta que salgamos del bloque en el que estén declaradas

- Memoria reservada con `malloc()`

  El contenido dura hasta que se llama a `free()`

Arrays

tipo nombre[tamaño];

- Un array es una zona de memoria (global o local) en la que se almacenan varios valores del mismo tipo
- La definición de un array significa “crea una zona de memoria en la que guardar **tamaño** elementos del tipo **tipo**, y un puntero **nombre** que apunta al primero de esos elementos”
- Por lo tanto, **nombre** es en realidad un puntero que apunta a elementos del tipo **tipo**.
- ***nombre** es el primero de esos elementos, ***(nombre+1)** es el segundo . . . ***(nombre+tamaño-1)** es el último

Arrays (2)

- Para referirse a un elemento determinado de un array se suele usar otro operador, los corchetes: **nombre[i]**, que es equivalente a escribir *(nombre+i).
- Por lo tanto, se suele acceder a los elementos de un array mediante la notación **nombre[0]** . . . **nombre[tamaño-1]**.

```
int lista[100]; /* Array de 100 elementos */

for (i=0; i<100; i++) {
  lista[i] = i+1;
  printf("lista[%d]=%d\n", i, lista[i]);
}
```

Arrays (3)

- Internamente los arrays no existen, solo hay punteros y direcciones de memoria.
- La expresión “**a[i]**” significa, exactamente, “***(a+i)**”.
- Por lo tanto, C no proporciona ningún tipo de control de límites de un array.

Acceder a “**array[-20]**” no da ningún tipo de error ni warning, pero casi seguro que no es lo que queremos hacer. Hay que tener mucho cuidado para no usar una zona de memoria que no nos corresponda.

Arrays multidimensionales

tipo nombre[tamaño1][tamaño2]...;

```
int matriz[10][10]; /* Array de 10x10 elementos */

for (i=0; i<10; i++) {
  sumafila = 0;
  for (j=0; j<10; j++) {
    sumafila += matriz[i][j];
  }
  printf("Fila %d, suma =%d\n", i, sumafila);
}
```

Cadenas de caracteres

- Una cadena de caracteres se representa siempre encerrada entre comillas dobles, `"como esta"`.
- Esa notación significa “guarda esos caracteres en una zona de memoria **constante**, seguidos por un carácter de fin de cadena, y dame un puntero que apunte al primero de esos caracteres”.
- Por lo tanto, podemos almacenar el resultado de esa expresión (ese puntero) en una variable de tipo “puntero a char”:

```
char * s = "Hola, mundo";
```

- Eso solo almacena la dirección del comienzo de la cadena, y no la cadena entera.
- Igual que si fuera un array, podemos acceder al carácter `i` mediante `s[i]`.

Cadenas de caracteres (2)

char nombre[tamaño];

```
char asignatura[100];

strcpy(asignatura, "sistemas operativos");

asignatura[0] = 'S';
printf("El nombre empieza por %c\n", asignatura[0]);
printf("Nombre de la asignatura: %s\n", asignatura);

/* .... */
```

Parámetros de main()

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

main() siempre recibe 2 parámetros:

- **argc**: Un entero que indica el número de argumentos con los que se ha llamado a nuestro programa, más uno.
- **argv**: Un array de punteros a char. Cada puntero apunta al nombre de uno de los argumentos con los que se ha llamado al programa, comenzando por el propio nombre del programa.

Ejemplo: si llamamos a nuestro programa "**hello**" y lo ejecutamos con "`./hello alpha beta gamma`":

```
                argc = 4
argv[0] = "./hello"  argv[2] = "beta"
argv[1] = "alpha"    argv[3] = "gamma"
```