

# Programación en C

Programación en C

1

## Índice

- Estructura de un programa C.
- Variables básicas.
- Operaciones arit. y lógicas
- Sentencias de control.
- Arrays y Strings.
- Funciones.
- Estructuras de datos.
- Entrada/Salida básica.
- Ejemplos I.
- Modificadores de ámbito de las variables.
- Punteros y memoria dinámica.
- Preprocesador C y compilación.
- Librerías estándar.
- Ejemplos II.

Programación en C

2

# Programación en C

## Estructura de un programa en C

Programación en C

3

## Estructura de un programa en C

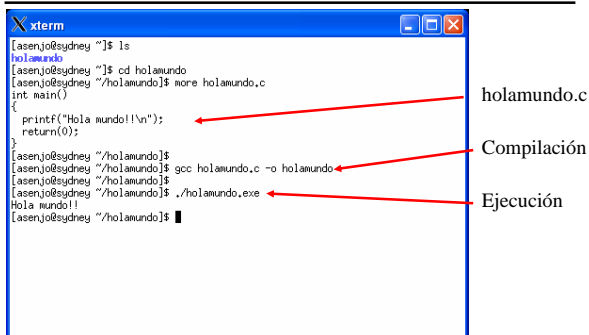
Función `main()`:

```
int main()  
{  
    printf("Hola mundo!!\n");  
    return(0);  
}
```

Programación en C

4

## Estructura de un programa en C



```
xterm  
[asenjo@sydney ~]$ ls  
holamundo  
[asenjo@sydney ~]$ cd holamundo  
[asenjo@sydney ~/holamundo]$ more holamundo.c  
int main()  
{  
    printf("Hola mundo!!\n");  
    return(0);  
}  
[asenjo@sydney ~/holamundo]$  
[asenjo@sydney ~/holamundo]$ gcc holamundo.c -o holamundo  
[asenjo@sydney ~/holamundo]$  
[asenjo@sydney ~/holamundo]$ ./holamundo.exe  
Hola mundo!!  
[asenjo@sydney ~/holamundo]$
```

Programación en C

5

## Las raíces del lenguaje C

- 1967. Martin Richards desarrolla BCPL
  - Sin tipos de datos. Uso intensivo de punteros y su aritmet.
- 1970. Ken Thompson implementa UNIX en leng. B
- 1972. Dennis Ritchie desarrolla C en los Bell Labs
  - UNIX se reescribe en C y se instala en una PDP-11
- El compilador de C se suministra con el S.O. Unix
  - Se hace muy popular, pero aparecen muchas variantes
- 1983. ANSI C. Estandarización del comité ANSI

Programación en C

6

## Consideraciones sobre C

- Se considera como un lenguaje de nivel-medio
  - Combina aspectos de alto-nivel con control de bajo nivel
    - Acceso a bits, interrupciones, direcciones de memoria
    - Lenguaje apropiado para programación de SW de sistemas
  - Es conciso (32 palabras clave) y por tanto portable
- Es modular soportando funciones y variables locales
- Compromiso entre nivel de programación y prestac.
  - Ej.: no chequea los límites de las var. dimensionadas
  - A cambio es rápido, flexible y poco restrictivo

Programación en C

7

## Características de C

- Sensible a mayúsculas y minúsculas: `sum` y `Sum`
- Indentación y espacios en blanco (recomendable)
- Sentencias (terminan con un punto y coma).
- Sent. compuestas o bloques (entre llaves { })
- Elementos de un programa:
  - Palabras reservadas (muy pocas: 32).
  - Funciones de librería estándar.
  - Variables y funciones definidas por el programador.

Programación en C

8

## Características de C

- El programa no contiene su nombre en el interior
  - Se le atribuye externamente en el nombre con ext. ".c"
- Todo programa debe incluir la función `main()`
- Las funciones se llaman por su nombre seguido de los argumentos entre paréntesis
  - Aunque no haya argumnts, los paréntesis son obligatorios
- Muchas funciones están en librerías estándar/extern.
  - Se puede usar escribiendo `#include <libreria.h>`

Programación en C

9

## Comentarios

- Los comentarios en C pueden ocupar varias líneas y se encuentran delimitados entre `/*` y `*/`.
- Comentario de una sola línea: Empieza con `//`

```
int main()
{
    /* Esto es un comentario de varias
       líneas.*/
    return(0); //Comentario de una línea
}
```

COMENTAR LOS PROGRAMAS

Programación en C

11

## Programación en C

### Variables básicas

Programación en C

12

## Tipos de variables

- Los tipos elementales de variables en C son:
  - Enteros (`int`).
  - Reales (`float`, `double`).
  - Caracteres (`char`).
  - Punteros (`*`).

**NO** existe un tipo booleano (en su lugar se usa `int` o `char`).

Programación en C

13

## Modificadores de tipos

- Ciertos tipos básicos admiten diversos modificadores:
  - `unsigned`: Sólo valores positivos (sin signo).
  - `signed`: Valores positivos y negativos (por omisión).
  - `long`: Formato largo
  - `short`: Formato corto (para enteros únicamente)

### Ejemplos:

`unsigned int`

`signed char`

`long int` (usualmente representado como `long`)

`unsigned long int`

Usar `sizeof(tipo)` para determinar el nº de bytes que necesita

Programación en C

14

## Resumen de tipos fundamentales

- Entre paréntesis: notación abreviada

Datos Enteros		signed char (char) [8bits]	unsigned char
	signed short int (short) [16 bits]	signed int (int) [32 bits]	signed long int (long) [32 bits]
	unsigned short int (unsigned short)	unsigned int (unsigned)	unsigned long int (unsigned long)
Datos Reales	float [32 bits]	double [64 bits]	long double [80 o 96 bits]

Programación en C

15

## Declaración de variables

- Declaración simple:
  - `char c;`
  - `unsigned int i;`
- Declaración múltiple:
  - `char c,d;`
  - `unsigned int i,j,k;`
- Declaración y asignación:
  - `char c='A', c1='\x41', c2='\101'; /* Hex. y Oct */`
  - `unsigned int i=133,j=1229;`
  - `int i, j=4, k;`
  - `float x=y=1.2e-3, z=1.5;`

Programación en C

16

## Identificadores

- Nombre de una variable o función. Reglas
  - Secuencia de letras [a-z, A-Z], dígitos [0-9] y/o `_`
  - No puede contener otros caracteres (`*`, `;`, `:`, `+`, etc)
  - El primer carácter tiene que ser una letra o `_`
  - Case-sensitive (distinción entre mayúsculas y minúsculas)
  - Hasta 31 caracteres de longitud
  - Ejemplos válidos
    - `tiempo`, `dist1`, `caso_A`, `PI`, `v_de_la_luz`
  - Ejemplos NO válidos
    - `i_valor`, `tiempo-total`, `dolares$`, `%final`

Programación en C

17

## Llamada `sizeof()`

- La llamada `sizeof()` se utiliza para determinar el número de bytes que ocupa una variable o un tipo:

```
int a;
num_bytes_a = sizeof(a);
num_bytes_int = sizeof(unsigned int);
```

Programación en C

18

## Ámbito de las variables

- La declaración de las variables lleva asociado un ámbito, dentro del cual la variable es visible:
  - Ámbito global: La variable es visible para todas las funciones del programa.
  - Ámbito local: La variable es visible sólo dentro de la función. (Tiene prioridad sobre el ámbito global)

Programación en C

19

## Ámbito de las variables

```
int x,y;
int main()
{
    float x,z;
    /* Aquí x y z son reales e y un entero */
}

/* Aquí x e y son variables enteras */
/* La variable z no existe fuera de la
función */
```

Programación en C

20

## Expresiones constantes

- El formato de las expresiones constantes es;
  - Un expresión real se puede dar tanto en notación decimal (2.56) como científica (2.45E-4).
  - A una expresión de tipo long se le añade un L al final (200L).
  - Una expresión de tipo carácter se define entre comillas simples ('A').

Programación en C

21

## Expresiones constantes

- Para definir las constantes de tipo carácter asociadas a caracteres especiales se usan secuencias de escape:
  - '\n': Retorno de carro. ASCII 10
  - '\t': Tabulador. ASCII 9
  - '\a': Bell. ASCII 7
  - '\r': Retorno de carro. ASCII 13
  - '\\': Barra invertida. ASCII 92
  - '\'' : Apóstrofo. ASCII 39
  - '\"' : Comillas dobles. ASCII 34
  - '\0': Carácter nulo. ASCII 0

Programación en C

22

## Expresiones constantes

- Las constantes enteras se pueden representar en diferentes bases numéricas:
  - Base decimal: 230.
  - Base hexadecimal:
    - 'x41' (para el carácter 'A')
    - 0x3A0 o 0X3A0 (comienza por cero-x).
  - Base octal:
    - '101' (para el carácter 'A')
    - 0210 (comienza por cero).

char c='A', d='x41', e='101', f=0x41, g=0101, h=65; → Todo 'A'

Programación en C

23

## Conversiones explícitas de tipo: Casting

- Casting**: mecanismo usado para cambiar de tipo expresiones y variables:

```
int a;
float b;
char c;

b=65.0;
a=(int)b; /* a vale 65 */
c=(char)a; /* c vale 65 (Código ASCII
de 'A') */
```

Programación en C

24

## Conversiones implícitas de tipo

- Cuando se hacen operaciones entre variables de distinto tipo:
  - Por ejemplo `b + c`, donde `b` es `int`, y `c` es `float`
  - Promoción de la variable de menor rango
    - `b` se promociona a `float` antes de hacer la suma con `c`.
    - Rangos: `long double > double > float > long > int > char`
- En las asignaciones
  - Por ejemplo `int a; float b,c; a = b + c;`
  - El resultado de la suma se convierte a `int` (menor rango!)

Programación en C

25

## Punteros

- Un puntero es una variable que contiene la dirección de otra variable.
- Se declaran con un asterisco delante del identificador de la variable:  

```
int *px, y; /* px es un puntero a entero e y un entero */
```
- La variable `px` (el puntero) contiene la dirección de una posición de memoria donde hay un entero, `*px`

Programación en C

26

## Punteros

- Los gestión de punteros admite dos operadores básicos:
  - Si `px` es un puntero (dirección): `*px` es el contenido de la posición de memoria apuntada por el puntero (el valor almacenado en la dirección contenida en `px`).
    - `*` es el operador indirección
  - Si `x` es una variable: `&x` es la dirección de memoria donde está almacenada la variable.
    - `&` es el operador dirección

Programación en C

27

## Punteros

```
int main()
{
    int *px, y=3;

    px=&y;
    /* px apunta a y */

    *px=5;
    /* y vale 5 */
}
```

Dirección	Contenido	Gráfica								
px-> 35: y-> 39:	<table><tr><td>?</td><td>?</td><td>?</td><td>?</td></tr><tr><td>0</td><td>0</td><td>0</td><td>3</td></tr></table>	?	?	?	?	0	0	0	3	<p>px 39</p> <p>y 3</p>
?	?	?	?							
0	0	0	3							
px-> 35: y-> 39:	<table><tr><td>0</td><td>0</td><td>0</td><td>39</td></tr><tr><td>0</td><td>0</td><td>0</td><td>3</td></tr></table>	0	0	0	39	0	0	0	3	<p>px 39</p> <p>y 3</p>
0	0	0	39							
0	0	0	3							
px-> 35: y-> 39:	<table><tr><td>0</td><td>0</td><td>0</td><td>39</td></tr><tr><td>0</td><td>0</td><td>5</td><td>3</td></tr></table>	0	0	0	39	0	0	5	3	<p>px 39</p> <p>y 5</p>
0	0	0	39							
0	0	5	3							

Programación en C

28

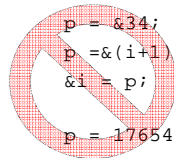
## Punteros

- En una archit. de 32 bits, un puntero ocupa 4Bytes
- Un puntero con valor 0 apunta a NULL
- Sentencias ILEGALES. Supongamos `int *p;`

```

p = &34; // las const. no tienen dir.
p = &(i+1); // las expr. tampoco
&i = p; // la dir. de una var. no se // puede cambiar
p = 17654; // Obligatorio el casting: // sería p=(int *)17654

```



Programación en C

29

## Punteros

- Un puntero puede ser de tipo `void`
  - Puede apuntar a variables de cualquier tipo
- No se permiten asig. sin casting entre punteros que apuntan a distinto tipo de variable.  

```

int *p; // Puntero a int
float *q; // Puntero a float
void *r; // Puntero a cualquier tipo de var.
p = q; // MAL!!
p = (int *) q; // BIEN
p = r = q; // BIEN

```
- Declarar una variable (que no sea un puntero) de tipo `void` no tiene sentido.
  - `void *px, v; // v MAL declarada`

Programación en C

30

## Aritmética de punteros

- Los punteros guardan información de
  - la dirección a la que apuntan
  - el tipo del dato que se almacena en esa dirección
- Se pueden hacer sumas y restas con punteros
 

```
int *p;
p = p + 1; // a p se le suma sizeof(int)
```
- Si restas dos punteros de igual tipo te devuelve
  - Distancia entre las direcciones. PERO NO EN BYTES
  - Sino en datos de ese mismo tipo

Programación en C

31

# Programación en C

## Operaciones aritméticas y lógicas

Programación en C

32

## Operaciones aritméticas

- El operador de asignación es el igual (=).
- Los operadores aritméticos son:
  - Suma (+)
  - Resta (-)
  - Multiplicación (\*)
  - División (/)
  - Módulo o resto de la división entera (%)
    - Sólo aplicable a operandos entero:  $24 \% 3$  es 3.

Programación en C

33

## Operaciones aritméticas

- División entera vs división real:
  - Depende de los operandos:

4 / 3	--> 1	entero
4.0 / 3	--> 1.333	real
4 / 3.0	--> 1.333	real
4.0 / 3.0	--> 1.333	real

Programación en C

34

## Expresiones

- Conjunto de variables/constantes/expresiones relacionadas mediante operandos
  - $5.0 + 3.0 * x - x * x / 2.0$
- Las expresiones pueden contener paréntesis
  - Agrupan términos y sub-expresiones
- Las expresiones están al lado derecho de la asignac.
  - Nunca en el lado izquierdo : MAL  $a + b = c$

Programación en C

35

## Pre/post-incrementos

Los operadores unarios (++) y (--) representan operaciones de incremento y decremento, respectivamente.

```
a++; /* similar a a=a+1 */
```

- Existen post- y pre- incremento/decremento

Ejemplos:

```
a=3; b=a++; /* Post incr. a=4, b=3 */
a=3; b=++a; /* Pre incr. a=4, b=4 */
a=3; b=a--; /* a=2, b=3 */
```

Programación en C

36

## Operaciones de asignación

El operador de asignación en C es el igual(=)

```
a=b+3;
```

Existen otras variantes de asignación:

```
a+=3; /* Equivalente a a=a+3 */
a*=c+d; /* Equivalente a a=a*(c+d) */
a/=a+1; /* Equivalente a a=a/(a+1) */
```

Para las asignaciones entre variables o expresiones de tipos diferentes se recomienda hacer *casting*.

```
a=(int)(x/2.34);
```

Programación en C

37

## Operadores de comparación

Los operadores de comparación en C son:

- Igual (==)
- Distinto (!=)
- Mayor (>) y Mayor o igual (>=)
- Menor (<) y Menor o igual (<=)

El resultado de un operador de comparación es un valor entero (0 es falso) y (distinto de 0 verdadero).

```
a=3>7 /* a vale 0 (falso) */
```

Programación en C

38

## Operadores lógicos

Sobre expresiones booleanas (enteros) se definen los siguientes operadores lógicos:

- And lógico (&&)
- Or lógico (||)
- Negación lógica (!)

Ejemplo

```
a=(3>2 || 5==4) && !1 /* Falso */
```

C tiene un modelo de evaluación perezoso.

```
a=3>2 || w==4 /* w==4 no se evalúa */
```

Programación en C

39

## Reglas de precedencia y asociatividad

- 3+4\*2 es 14?... 0 es 11?

Precedencia	Asociatividad
() [] -> .	→
++ -- ! sizeof (tipo) * (dirección) &(dirección)	←
* / %	→
+ -	→
< <= > >=	→
== !=	→
&&	→
	→
?:	←
= += -= *= /=	←
, (operador coma en bucles)	→

Programación en C

40

## Operadores de Bit

- & → Y lógico (and) a nivel de bit
- | → O lógico (or) a nivel de bit
- ^ → O exclusivo lógico (xor) a nivel de bit
- << → Desplazamiento a la izquierda
  - a = b << 2 // a es igual a b\*4 (si no hay overflow)
- >> → Desplazamiento a la derecha
  - a = b >> 2 // a es igual b/4 (hace extensión de signo)
- ~ → Complemento a 1 (not)

Programación en C

41

## Operadores de Bit

```
char a=48;      00110000 a
char b=19;      00010011 b
char x,y,z,w,t,s;

x=a & b;        00010000 x = 16
y=a | b;        00110011 y = 51
z=a ^ b;        00100011 z = 35
w=~a;           11001111 w = 207
t=a>>2;         00001100 t = 12
s=b<<3;         10011000 s = 152
```

Programación en C

42

## Uso de los Operadores de Bit

```
const char LECTURA =1;
const char ESCRITURA=2;
const char EJECUCION=3;

char permisos=LECTURA | ESCRITURA;
permisos|=EJECUCION;
permisos&=~ESCRITURA;

if(permisos & EJECUCION)
    printf("Es ejecutable");
```

Programación en C

43

## Sentencias

- Las sentencias contienen expresiones
  - Expresiones aritméticas, lógicas o generales.
- Tipos de sentencias
  - Simples: expresiones terminadas con ";"
    - Declaraciones o sentencias aritméticas (float a; a =a+1;)
  - Vacía o nula: No hace nada → ";"
  - Compuesta o bloque: sentencias agrupadas entre { }

```
{
    int i=1, j=3, k;
    k=i+j;
}
```

Programación en C

44

## Programación en C

### Sentencias de control

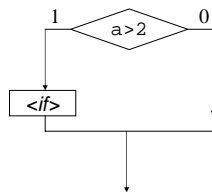
Programación en C

45

### if

```
int main()
{
    int a=3,b;

    if(a>2)
    {
        b=100+a;
        printf("parte if");
    }
}
```



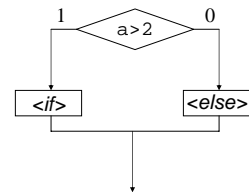
Programación en C

46

### if ... else

```
int main()
{
    int a=3,b;

    if(a>2)
    {
        b=100+a;
        printf("parte if");
    }
    else
        printf("parte else");
}
```



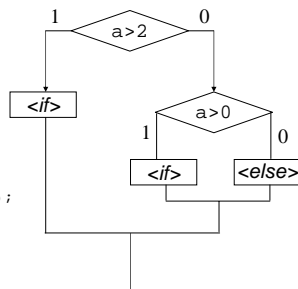
Programación en C

47

### if ... else múltiple

```
int main()
{
    int a=3,b;

    if(a>2)
    {
        printf("parte if");
    }
    else if (a>0)
        printf("parte else if");
    else
        printf("parte else");
}
```



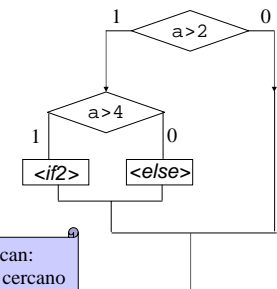
Programación en C

48

### if anidados

```
int main()
{
    int a=3,b;

    if(a>2)
        printf("parte if");
    if (a>4)
        printf("parte if2");
    else
        printf("parte else");
}
```



Si las llaves no lo especifican:  
El else pertenece al if más cercano

Programación en C

49



## Operador ?

```
int main()
{
    int a,b=4,c=5;

    a=b>0 ? c : c+1;
    /* Equivalente a
       if(b>0)
           a=c;
       else
           a=c+1; */
}
```

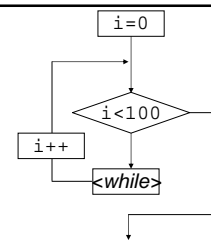
Programación en C

50

## while

```
int main()
{
    int i=0,ac=0;

    while(i<100)
    {
        printf("%d",i*i);
        ac+=i;
        i++;
    }
}
```



Puede que el cuerpo del while no llegue a ejecutarse nunca

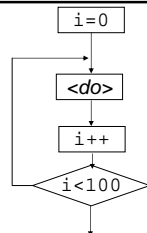
Programación en C

51

## do ... while

```
int main()
{
    int i=0,ac=0;

    do
    {
        printf("%d",i*i);
        ac+=i;
        i++;
    }
    while(i<100);
}
```



El cuerpo del while se ejecuta al menos una vez  
NOTAR el ";" al final del while

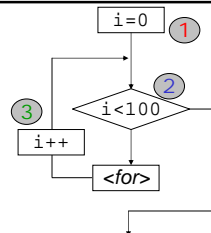
Programación en C

52

## for

```
int main()
{
    int i,ac=0;

    for(i=0;i<100;i++)
    {
        printf("%d",i*i);
        ac+=i;
    }
}
```



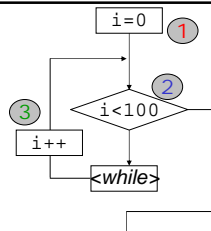
Sintaxis: **for**(inicialización;condición\_permanencia;incremento)

Programación en C

53

## for convertido a while

```
int main()
{
    int i,ac=0;
    1 i=0;
    2 while(i<100)
    {
        printf("%d",i*i);
        ac+=i;
        3 i++;
    }
}
```



Sintaxis:  
**for**(inicialización;condición\_permanencia;incremento)

Programación en C

54

## Ejemplo: producto escalar de vectores

```
int main()
{
    int i;
    float pe, a[100], b[100];

    for(pe=i=0; i<100; i++)
        pe+=a[i]*b[i];
}
```

a	b
1	5
3	8
4	3
2	9
7	1
5	2
6	3
3	1

NOTAR el uso del operador "+"  
Sentencia múltiple de asignación

Programación en C

55

## break y continue

```
int main()
{
    int i;
    for(i=0;i<100;i++)
    {
        if(i%2==0)
            continue; /*Comienza la sig. iter.*/
        if(i%17==0)
            break; /*Sale del bucle*/
        printf("%d",i);
    }
}
```

Programación en C

56

## switch

```
switch(ch)
{
    case 'A': printf("A");
              break;
    case 'B':
    case 'C': printf("B o C");
              break;
    case 'D': printf("B, C o D");
              break;
    default: printf("Otra letra");
}
```

Programación en C

57

## Programación en C

### Arrays y Strings

Programación en C

58

## Definición de Arrays

La definición de una variable de tipo array (vector) se realiza indicando la dimensión entre corchetes:

```
int a[100]; /* Un vector de 100 enteros */
float vx[4][4]; /* Matriz de 4x4 reales */
int *pt[10][10][10][10]; /* Una matriz de
    4 dimensiones de punteros a enteros */
```

Asimismo, pueden inicializarse:

```
float a[3]={2.45, -1.34, 2.11};
int vx[2][3]={{3,5,1},
              {2,1,2}};
```

Programación en C

59

## Indexación de arrays

- Un array se accede mediante un subíndice (exp. int)
- El rango de subíndices es [0, no\_elementos-1].

```
int a[6];
a[0]=23;
a[3]=a[0]+5;
for(i=0;i<6;i++) printf("%d",a[i]);
// No se puede operar con el array entero
// Por ejemplo → a=23; MAL
```

a → 

23			28		
----	--	--	----	--	--

  
a[0] a[1] a[2] a[3] a[4] a[5]

Programación en C

60

## Arrays de varias dimensiones

- Una matriz es una array de dos dimensiones
  - int a[2][4]; // Matriz de dos filas y 4 columnas
  - Las matrices se almacenan por filas
    - En una matriz de NxM, el elemento (i,j) se almacena en posición del elemento[0][0] + i\*M\*sizeof(tipo) + j\*sizeof(tipo)
- Tres dimensiones: int a[2][4][3];
  - Se puede ver como 2 matrices de 4x3

Programación en C

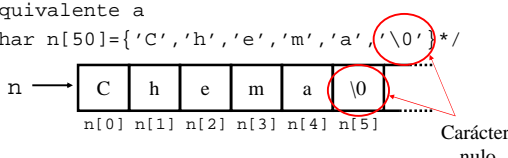
61

## Strings

Los strings son los arrays de caracteres de una dimensión. Son las cadenas de caracteres.

Definición:

```
char x[20], n[50] = "Chema";  
/*equivalente a  
char n[50] = {'C', 'h', 'e', 'm', 'a', '\0'} */
```



Carácter nulo

Programación en C

62

## Asignación de Strings

La asignación de strings por medio del operador (=) sólo es posible en la declaración.

Ejemplo:

```
char n[50] = "Chema"; /* Correcto */  
n = "Otro nombre"; /* Error: no es  
declaración */
```

Para las operaciones sobre strings se utilizan diferentes funciones de librería. Por ejemplo, **strlen()** calcula el tamaño del string (número de caracteres).

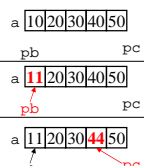
Programación en C

63

## Arrays y punteros

- El identificador de una variable array es un puntero
  - Un punt. constante (su dirección no se puede cambiar)

```
int *pb, *pc;  
int a[5] = {10, 20, 30, 40, 50};  
pb = a; // o pb = &a[0];  
*pb = 11;  
pc = &a[3];  
*pc = 44;
```



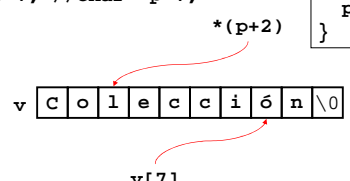
Programación en C

64

## Aritmética de Punteros y Arrays

- La aritmética de punteros permite acceder a arrays

```
char v[] = "Colección";  
char *p;  
p = v; // char *p = v;  
for(p = v; *p; p++)  
{  
    printf("%c", *p);  
}
```



Programación en C

65

## Aritmética de Punteros y Arrays

- Resumiendo
  - \*p equivale a v[0], a \*v y a p[0];
  - \*(p+1) equivale a v[1], a \*(v+1) y a p[1];
  - etc.

- Cuatro formas de sumar los elementos de a[n]

```
int a[n], suma, i, *p;  
for (i=0, suma=0; i<n; i++) suma += a[i] // 1ª  
for (i=0, suma=0; i<n; i++) suma += *(a+i) // 2ª  
for (p=a, i=0, suma=0; i<n; i++) suma += p[i] // 3ª  
for (p=a, suma=0; p<&a[n]; p++) suma += *p // 4ª
```

Programación en C

66

## Aritmética de Punteros y Arrays

Las operaciones soportadas sobre punteros son:

- Suma y resta de valores enteros (+, -, ++ y --).
- Comparación y relación (<, >, <=, >=, == y !=).
- Valor booleano (comparación con **NULL**).

```
void copiar(char* dest, const char* orig)  
{  
    if(orig && dest)  
        while(*orig)  
            *dest++ = *orig++;  
}
```

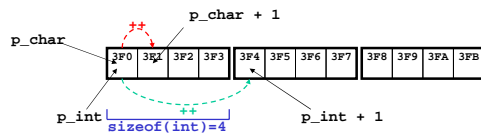
Programación en C

67

## Aritmética de Punteros

Las operaciones de suma o resta sobre punteros modifican el valor del dependiendo del tipo del puntero:

```
int* p_int; char* p_char; (p_int=p_char;)
p_int++; /* Suma sizeof(int) */
p_char++; /* Suma sizeof(char) */
```



Programación en C

68

## Programación en C

### Funciones

Programación en C

70

## Definición de una función

La definición de una función tiene la estructura:

```
tipo identificador (argumentos ...) _____ Header
{                                     (Cabecera)
    ...
    cuerpo de la función
    ...
}
```

Programación en C

71

## Uso de una función

Una función se invoca proporcionando valores a los argumentos de la llamada.

- Los argumentos se pasan **siempre por valor**.
- El valor se devuelve por medio de `return( )`.
- Los procedimientos son funciones de tipo `void`.
- El control del número y tipo de argumentos es mínimo.
- Las funciones en C admiten recursividad.

Programación en C

72

## Función de ejemplo

```
int factorial(int n)
{
    int ret=1;
    while (n>1)
        ret*=n--;
    return(ret);
}
int main()
{
    printf("%d!=%d\n",5,factorial(5));
}
```

Programación en C

73

## Declaración de funciones

Para poder hacer uso de una función es necesario que ésta esté definida o declarada con antelación.

- Definición de la función: Todo el código de la función.
- Declaración de la función: Únicamente la cabecera o prototipo de la función:

```
int factorial(int n);
int factorial(int);
int factorial();
```

Programación en C

74

## Prototipo de una función

```
int factorial(int n);    /* Prototipo */

int main()
{
    printf("%d!=%d\n",5,factorial(5));
}

int factorial(int n) /* Definición */
{
    ...
}
```

Programación en C

75

## Paso por referencia

El lenguaje C **NO** tiene paso por referencia. En su lugar se pasa por valor la dirección de la variable a modificar.

```
int reiniciar(int *a, int b)
{
    *a=0; b=0;
}

int x=2,y=2;
reiniciar(&x,y); /* x valdrá 0 e y 2 */
```

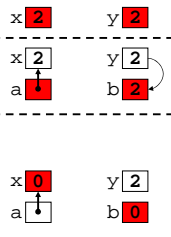
Programación en C

76

## Paso por referencia

```
int x=2,y=2;
reiniciar(&x,y);

-----
int reiniciar(int *a, int b)
{
    *a=0; b=0;
}
```



Programación en C

77

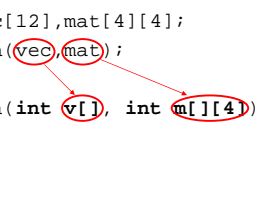
## Argumentos de tipo array

Cuando un array se pasa como argumento a una función, la primera de las dimensiones no se define.

Ejemplo:

```
int vec[12],mat[4][4];
calcula(vec,mat);

void calcula(int v[], int m[][4])
{
    ...
}
```



Programación en C

78

## Programación en C

Estructuras de datos

Programación en C

79

## Tipos de estructuras

El lenguaje C tiene tres tipos de estructuras de datos:

- Registro o estructura (struct).
- Unión de campos (union).
- Tipo enumerado (enum).

Programación en C

80

## struct

Un **struct** es un tipo de datos complejo conformado por un conjunto de campos de otros tipos (básicos o complejos) asociados a un identificador:

```
struct [etiqueta]
{
    tipo campo;
    tipo campo;
    ...
};
```

Programación en C

81

## struct

```
struct persona
{
    char nombre[20];
    int edad;
    float peso;
} yo,tu,ellos[10];

struct persona el={"Antonio López",31,80};
struct persona *ella, todos[20];
```

Programación en C

82

## struct

El acceso a los campos de una estructura se hace por medio de un punto (.) o de una flecha (->) si es un puntero a estructura.

```
struct persona el, *ella, todos[20];

printf("Su nombre %s\n",el.nombre);
todos[2].edad=20;
ella=&todos[2];
printf("La edad de ella es %d\n",ella->edad);
```

Programación en C

83

## union

Un **union** es similar a un **struct**, pero todos los campos comparten la misma memoria.

```
struct datos          union datos
{                     {
    int a,b;           int a,b;
    int x[2];          int x[2];
    char c;            char c;
} d;                  } d;

d.a [ ] [ ] [ ] [ ]  d.x [ ] [ ] [ ] [ ] [ ] [ ]
d.b [ ] [ ] [ ] [ ]  d.a [ ] [ ] [ ] [ ] [ ] [ ]
d.x[0] [ ] [ ] [ ] [ ] d.b [ ] [ ] [ ] [ ] [ ] [ ]
d.x[1] [ ] [ ] [ ] [ ] d.c [ ] [ ] [ ] [ ] [ ] [ ]
d.c [ ] [ ] [ ] [ ]
```

Programación en C

84

## Uso de union

Los **union** se usan para diferentes representaciones de los datos o para información condicionada:

```
union
{
    int integer;
    char oct[4];
} data;

struct empleado
{
    char nombre[40];
    int tipo_contrato;
    union
    {
        int nomina;
        int pts_hora;
    } sueldo;
} p;
```

Programación en C

85

## Estructuras y funciones

Las estructuras de datos son tipos complejos y (aunque ciertos compiladores lo admiten) no deben ser pasados como argumentos ni devueltos por funciones. En su lugar se usan punteros a dichas estructuras:

```
void evaluar_empleado(struct empleado* emp);
struct empleado* nuevo_empleado();
```

Programación en C

86

## enum

- Las enumeraciones con conjuntos de constantes numéricas definidas por el usuario.

```
enum color {rojo, verde, azul} fondo;
enum color letras, borde=verde;

enum tipo_empleado {contratado=1,
                    temporal=2,
                    becario=3};
```

Programación en C

87

## Definición de nuevos tipos

Las sentencias `typedef` se usan para definir nuevos tipos en base a tipos ya definidos:

```
typedef int boolean;
typedef struct persona persona_t;
typedef struct punto
{
    int coord[3];
    enum color col;
} punto_t;
persona_t p[4];
```

Programación en C

88

## Programación en C

Entrada/salida básica

Programación en C

89

## Funciones de entrada/salida

- Las funcionalidades de entrada/salida en C no pertenecen a las palabras reservadas del lenguaje. Son funciones de librería, por ejemplo:
  - Entrada: `scanf()`.
  - Salida: `printf()`.

Programación en C

90

## printf()

- El formato de llamada de `printf()` es:  
`printf(format, exp1, exp2, exp3, ..., expn);`

donde:

- `format`: Es el string de formato de salida de los datos.
- `expi`: Es la expresión a incluir dentro del formato.

Programación en C

91

## printf()

Ejemplo:

```
int a=3;
float x=23.0;
char c='A';
printf("Hola mundo!!\n");
```

```
printf("Un entero %d\n", a);
```

```
printf("Un real %f \ny un char %c\n", x, c);
```

Programación en C

92

## printf()

Formato	Expresión	Resultado
%d %i	entero	entero decimal con signo
%u	entero	entero decimal sin signo
%o	entero	entero octal sin signo
%x %X	entero	entero hexadecimal sin signo
%f	real	real en notación punto
%e %E %g %G	real	real en notación científica
%c	carácter	carácter
%p	puntero	dirección de memoria
%s	string	cadena de caracteres
%ld %lu ...	entero largo	entero largo (distintos formatos)

Programación en C

93

## printf()

- Otras opciones de formato:
  - Precisión (número de decimales).
  - Justificación (izquierda o derecha).
  - Caracteres especiales (% o \).
  - ...

Ver página del manual:

man printf

Programación en C

94

## scanf()

- El formato de llamada de `scanf()` es:

```
scanf(format, dir1, dir2, dir3, ..., dirn);
```

donde:

- *format*: Es el string de formato de entrada de los datos.
- *dir<sub>i</sub>*: Es la dirección donde se almacena el resultado.

Programación en C

95

## scanf()

### Ejemplo

```
int a,*pa;
float x;
char c;

scanf("%d",&a); /* Lee un entero y lo
                  almacena en a */
scanf("%f %c",&x,&c); /* Lee x y c */
scanf("%d",pa); /* PELIGROSO */
pa=&a; scanf("%d",pa); /* OK. Lee a */
```

Programación en C

96

## scanf() Lectura de strings

### Ejemplo:

```
char *pc;
char str[82];

scanf("%s",pc); /* PELIGROSO */

scanf("%s",str); /* Lee hasta un blanco o
                  fin de línea */

scanf("%[^\n]",str); /* Lee toda la línea */
```

Programación en C

97

## Programación en C

### Ejemplos I

Programación en C

98



## Ejemplos I-1

Se plantea el desarrollo de un programa que lea los datos (nombre, apellidos, nif y sueldo) de 10 empleados por teclado e imprima los empleados con sueldo máximo y mínimo, así como la media de los sueldos.

Programación en C

99

## Ejemplos I-1: Solución

En primer lugar definimos la estructura de datos a utilizar.

```
struct empleado_st
{
    char nombre[40];
    char apellidos[40];
    char nif[10];
    long sueldo;
};

typedef struct empleado_st empleado_t;
```

Programación en C

100

## Ejemplos I-1: Solución

Seguidamente programamos el cuerpo del programa principal.

```
int main()
{
    empleado_t emp[10];
    int i,min,max;
    float med;

    for(i=0;i<10;i++)
        leer_empleado(&emp[i]);

    min=buscar_min(emp,10);
    max=buscar_max(emp,10);
    med=buscar_med(emp,10);

    printf("Mínimo:\n");
    imprimir_empleado(&emp[min]);
    printf("Máximo:\n");
    imprimir_empleado(&emp[max]);
    printf("Media:%9.2f\n",
           med);
    return(0);
}
```

Programación en C

101

## Ejemplos I-1: Solución

Los prototipos de las funciones a implementar son:

```
void leer_empleado    (empleado_t *pe);
int  buscar_min       (empleado_t es[],
                       int        tam);
int  buscar_max       (empleado_t es[],
                       int        tam);
float buscar_med       (empleado_t es[],
                       int        tam);
void imprimir_empleado(empleado_t *pe);
```

Programación en C

102

## Ejemplos I-1: Solución

```
void leer_empleado    (empleado_t *pe)
{
    printf("Nombre y Apellidos: ");
    scanf("%s %[^\\n]",
          pe->nombre,pe->apellidos);
    printf("NIF: ");
    scanf("%s",pe->nif);
    printf("Sueldo: ");
    scanf("%ld",&pe->suelo);
    printf("-----\\n");
}
```

Programación en C

103

## Ejemplos I-1: Solución

```
int buscar_min        (empleado_t es[],
                       int        tam)
{
    int candidato=0,i;

    for(i=1;i<tam;i++)
        if(es[i].suelo<es[candidato].suelo)
            candidato=i;
    return(candidato);
}
```

Programación en C

104

### Ejemplos I-1: Solución

```
int  buscar_max      (empleado_t es[],
                      int      tam)
{
    int candidato=0,i;

    for(i=1;i<tam;i++)
        if(es[i].sueldo>es[candidato].sueldo)
            candidato=i;
    return(candidato);
}
```

Programación en C

105

### Ejemplos I-1: Solución

```
float buscar_med      (empleado_t es[],
                       int      tam)
{
    int i;
    float acc=0;
    for(i=0;i<tam;i++)
        acc+=(float)es[i].sueldo;
    return(acc/(float)tam);
}
```

Programación en C

106

### Ejemplos I-1: Solución

```
void imprimir_empleado(empleado_t *pe)
{
    printf("\tNombre:      %s\n",pe->nombre);
    printf("\tApellidos:   %s\n",pe->apellidos);
    printf("\tNIF:          %s\n",pe->nif);
    printf("\tSueldo:       %ld\n",pe->sueldo);
}
```

Programación en C

107

## Programación en C

### Modificadores de Ámbito

Programación en C

108

### Modificadores de Variables

- La declaración de variables acepta los siguientes modificadores:
  - **static** (Local): El valor de la variable se conserve entre llamadas. Comportamiento similar a una variable global.
  - **register** : La variable es almacenada siempre (si es posible) en un registro de la CPU (no en memoria).
  - **volatile** : Un proceso exterior puede modificar la variable.
  - **const** : La variable no puede ser modificada.

Programación en C

109

### Modificadores de Variables

```
int una_funcion(int a, int b)
{
    static  char last;
    register int i;
    const   int  max=12;
    volatile long acc;

    ....
}
```

Programación en C

110

## Modificadores de Variables (**static**)

```
void cuenta()
{
    static int cnt=0;
    printf("%d\n",cnt++)
}

int main()
{
    cuenta();cuenta();cuenta();cuenta();
    return 0;
}
```

Salida:

```
0
1
2
3
```

Programación en C

111

## Modificadores de Variables (**const**)

```
const int max=10;
int letra(const char* text, char l)
{
    int i,acc=0;
    for(i=0;i<max && text[i];i++)
        if(text[i]==l)
            acc++;
    return acc;
}
```

Programación en C

112

## Modificadores de Funciones

- Las funciones también pueden ser declaradas con ciertos modificadores:
  - static** : Restricción de enlace. Sólo se puede usar dentro del mismo fichero (también variables globales).
  - extern** : La función o variable se encuentra declarada pero no definida. Usada con variables globales.
  - inline** : La función es expandida íntegramente al ser invocada. No hay un salto a la función. Incrementa la eficiencia y aumenta el tamaño del código.

Programación en C

113

## Modificadores de Funciones

**Fichero1.c:**

```
static void func()
{
    ...
}

void aux()
{
    func();
}
```

**Fichero2.c:**

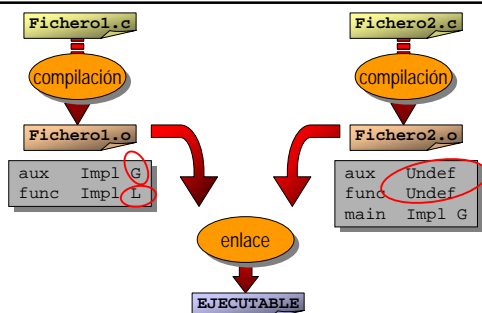
```
extern void aux();

int main()
{
    aux();
    func(); /* NO
            *   VISIBLE */
}
```

Programación en C

114

## Modificadores de Funciones



Programación en C

115

## Modificadores de Funciones

```
inline int max(int a, int b)
{
    if(a>b)
        return a;
    else
        return b;
}

...
x=max(x+1,y);
```

```
{
    if(x+1>y)
        x=x+1;
    else
        x=y;
}
```

Programación en C

116

# Programación en C

## Punteros y Memoria Dinámica

Programación en C

117

## Punteros a Funciones

Mecanismo para pasar funciones como argumento:

```
char (*f)(int,int);
```

**f** es un puntero a una función que devuelve un **char** y recibe dos enteros como argumento.

A un puntero a función se le puede asignar como valor cualquier identificador de función que tenga los mismos argumentos y resultado.

Programación en C

118

## Punteros a Funciones

```
char* menor (char** text,
             int tam,
             int (*compara)(char*,char*))
{
    int i;
    char* min=text[0];
    for(i=1;i<tam;i++)
        if(*compara(minor,text[i]))
            min=text[i];
    return min;
}
```

Programación en C

119

## Punteros a Funciones

```
int main()
{
    char *palabras[]={ "hola", "casa", "perro",
                       "coche", "rio" };
    printf("Menor:%s",
           menor(palabras,5,alfabetico));
    return 0;
}

int alfabetico(char* a,char* b)
{ .... }
```

Programación en C

120

## Memoria Dinámica

Además de la reserva de espacio estática (cuando se declara una variable), es posible reservar memoria de forma dinámica.

Funciones de gestión de memoria dinámica:

- **void\* malloc(size\_t)**: Reserva memoria dinámica.
- **free(void\*)**: Libera memoria dinámica.
- **void\* realloc(void\*,size\_t)**: Ajusta el espacio de memoria dinámica.

Programación en C

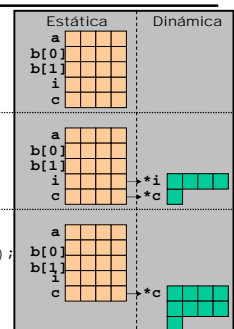
121

## Memoria Dinámica

```
int a,b[2];
int* i;
char* c;

i=(int*)malloc(sizeof(int));
c=(char*)malloc(sizeof(char));

free(i);
c=(char*)realloc(c,sizeof(char)*9);
```



Programación en C

122

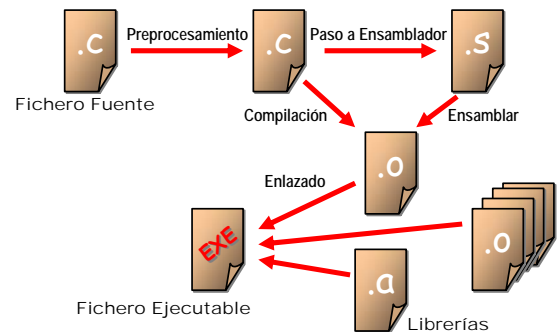
# Programación en C

## Preprocesador y Compilación

Programación en C

123

## Fase de Compilación



Programación en C

124

## Directrices del Preprocesador

Son expandidas en la fase de preprocesado:

- **#define** : Define una nueva constante o macro del preprocesador.
- **#include** : Incluye el contenido de otro fichero.
- **#ifdef** **#ifndef** : Preprocesamiento condicionado.
- **#endif** : Fin de bloque condicional.
- **#error** : Muestra un mensaje de error

Programación en C

125

## Constantes y Macros

Permite asociar valores constantes a ciertos identificadores expandidos en fase de preprocesamiento:

**#define** *variable* *valor*

Define funciones que son expandidas en fase de preprocesamiento:

**#define** *macro(args,...)* *función*

Programación en C

126

## Constantes y Macros

```
#define PI 3.14
#define NUM_ELEM 5
#define AREA(rad) PI*rad*rad
#define MAX(a,b) (a>b ? a : b)
int main()
{
    int i;
    float vec[NUM_ELEM];
    for(i=0;i<NUM_ELEM;i++)
        vec[i]=MAX((float)i*5.2,AREA(i));
}
```

Programación en C

127

## Constantes y Macros

Tras la fase de preprocesamiento

```
int main()
{
    int i;
    float vec[5];
    for(i=0;i<5;i++)
        vec[i]=((float)i*5.2>3.14*i*i ?
                (float)i*5.2 :
                3.14*i*i);
}
```

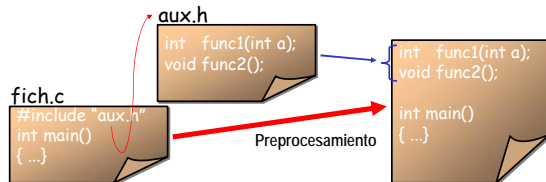
Programación en C

128

## Inclusión de Ficheros

Los prototipos de las funciones usadas por varios ficheros fuente se suelen definir en fichero de cabecera.

`#include <stdio.h>` Cabeceras del sistema.  
`#include "mis_func.h"` Ficheros de cabecera locales.



Programación en C

130

## Inclusión de Ficheros

La inclusión de ficheros esta sujeta a las siguientes recomendaciones:

- Por lo general los ficheros de cabecera tienen como **extensión .h**
- En los ficheros de cabecera **no se incluyen implementación** de funciones
- Las variables en un fichero de cabecera son **declaradas extern** y se encuentran declaradas en algún otro fichero **.c**

Programación en C

131

## Sentencias Condicionales

Para incluir código cuya compilación es dependiente de ciertas opciones, se usan los bloques:

```
#ifdef variable
<bloque de sentencias>
...
#endif
#ifdef variable
<bloque de sentencias>
...
#endif
```

Programación en C

132

## Ejemplo: Depuración

```
#define DEBUG
int main()
{
    int i,acc;
    for(i=0;i<10;i++)
        acc=i*i-1;
    #ifdef DEBUG
        printf("Fin bucle acumulador: %d",acc);
    #endif
    return 0;
}
```

Programación en C

133

## Ejemplo: Fichero de Cabecera

**aux.h**

```
#ifndef _AUX_H
#define _AUX_H
<definiciones>
#endif
```

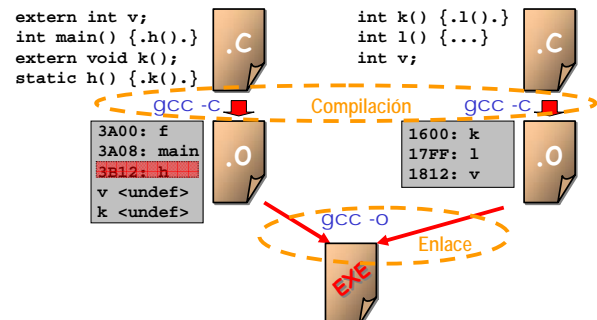
Evita la redefinición de funciones y variables

```
#include "aux.h"
int main()
{
    ...
}
```

Programación en C

134

## Enlace de Ficheros



Programación en C

135

# Programación en C

## Librerías Estándar

Programación en C

136

## Manejo de Cadenas

- `char* strcat(char*,char*)`: Concatena cadenas.
- `char* strchr(char*,char)`: Busca un carácter.
- `int strcmp(char*,char*)`: Comparación de cadenas.
- `char* strcpy(char*,char*)`: Copia cadenas.
- `char* strdup(char*)`: Duplica una cadena.
- `int strlen(char*)`: Longitud de una cadena.
- `char* strncpy(int,char*,char*)`: Copia cadenas.
- `char* strncat(int,char*,char*)`: Concatena.
- ...

Programación en C

137

## Manejo de Buffers

- `void* memcpy(void*,void*,int)`: Copia memoria.
- `void* memmove(void*,void*,int)`: Copia memoria.
- `int memcmp(void*,void*,int)`: Compara memoria.
- `void* memset(void*,int,int)`: Rellena memoria.
- `void bzero(void*,int)`: Pone a cero la memoria.
- `void bcopy(void*,void*,int)`: Copia memoria.
- `void* memccpy(void*,void*,int,int)`: Copia memoria hasta que encuentra un *byte*.
- ...

Programación en C

138

## Entrada Salida

- `int fprintf(FILE*,...)`: Salida sobre fichero.
- `int fscanf(FILE*,...)`: Entrada desde fichero.
- `int sprintf(char*,...)`: Salida sobre un buffer.
- `int sscanf(char*,...)`: Entrada desde un buffer.
- `int fgetc(FILE*)`: Lee un carácter desde fichero.
- `char* fgets(char*,int,FILE*)`: Lee una línea.
- `FILE* fopen(char*,char*)`: Abre un fichero.
- `int fclose(FILE*)`: Cierra un fichero.
- `int fflush(FILE*)`: Descarga un buffer de fichero.
- `int feof(FILE*)`: Indica si ha finalizado un fichero.

Programación en C

139

## Ficheros Especiales

Existen tres variables de tipo `FILE*` asociados a tres ficheros estándar:

- `stdin`: Entrada estándar.
- `stdout`: Salida estándar.
- `stderr`: Salida de error estándar.

```
fprintf(stdout,"Usuario: ");  
fscanf(stdin,"%s",usr);  
fprintf(stderr,"Error:No válido");
```

Programación en C

140

## Ordenación y Búsqueda

- `void* bsearch(void*,void*,int,int,int (*)(void*,void*))`:  
Búsqueda binaria sobre lista ordenada.
  - `void qsort(void*,int,int,int (*)(void*,void*))`:  
Ordenación mediante el algoritmo *quicksort*.
- ```
char *vec[10]={\"casa\",.....};  
  
qsort((void*)vec,10,sizeof(char*),strcmp);
```

Programación en C

141

## Conversión de Tipo

- `int atoi(char*)`: Traduce de string a entero.
- `long atol(char*)`: Traduce de string a un entero largo.
- `double atof(char*)`: Traduce de string a real.
- `long strtol(char*,char**,int)`: Traduce de array a entero (en diferentes bases).
- `double strtod(char*,char**)`: Traduce de array a real.
- `char* itoa(char*,int)`: Traduce un entero a array.

Como alternativa se pueden usar las funciones:  
`sscanf` y `sprintf`.

Programación en C

142

## Funciones Matemáticas

- `double exp(double)`: Calcula  $e^x$ .
- `double log(double)`: Calcula el logaritmo natural.
- `double log10(double)`: Calcula el logaritmo en base 10.
- `double pow(double,double)`: Calcula  $x^y$ .
- `double sqrt(double)`: Calcula la raíz cuadrada.
- `double sin(double)`: Calcula el seno (en radianes).
- `double cos(double)`: Calcula el coseno (en radianes).
- `double sinh(double)`: Calcula el seno hiperbólico.
- `double atan(double)`: Calcula el arco tangente.
- ...

Programación en C

143

## Uso de Funciones de Librería

El uso de las funciones de librería estándar esta sujeto a las siguientes recomendaciones:

- Estudiar la página del manual (`man 3 sprintf`).
- Incluir en el fichero fuente el fichero de cabecera adecuado (`#include <stdio.h>`).
- Enlazar la librería si es necesario (`gcc .... -lm`).

Programación en C

144

## Programación en C

Argumentos del Programa

Programación en C

145

## Argumentos de main

La función principal `main` puede recibir argumentos que permiten acceder a los parámetros con los que es llamado el ejecutable.

- ```
int main(int argc, char* argv[])
```
- `int argc` : Número de parámetros.
  - `char* argv[]` : Parámetros del ejecutable.

Programación en C

146

## Argumentos de main

```
$ gcc prog.c -o prog
$ prog uno dos tres cuatro
```

```
int main(int argc, char* argv[])
    argc=5
    argv[0] → prog
    argv[1] → uno
    argv[2] → dos
    argv[3] → tres
    argv[4] → cuatro
```

Programación en C

147



## Argumentos de `main`

---

```
int main(int argc, char* argv[])
{
    int i=0;
    printf("Ejecutable: %s\n",argv[0]);

    for(i=0;i<argc;i++)
        printf("Argumento[%d]: %s\n",i,argv[i]);

    return 0;
}
```