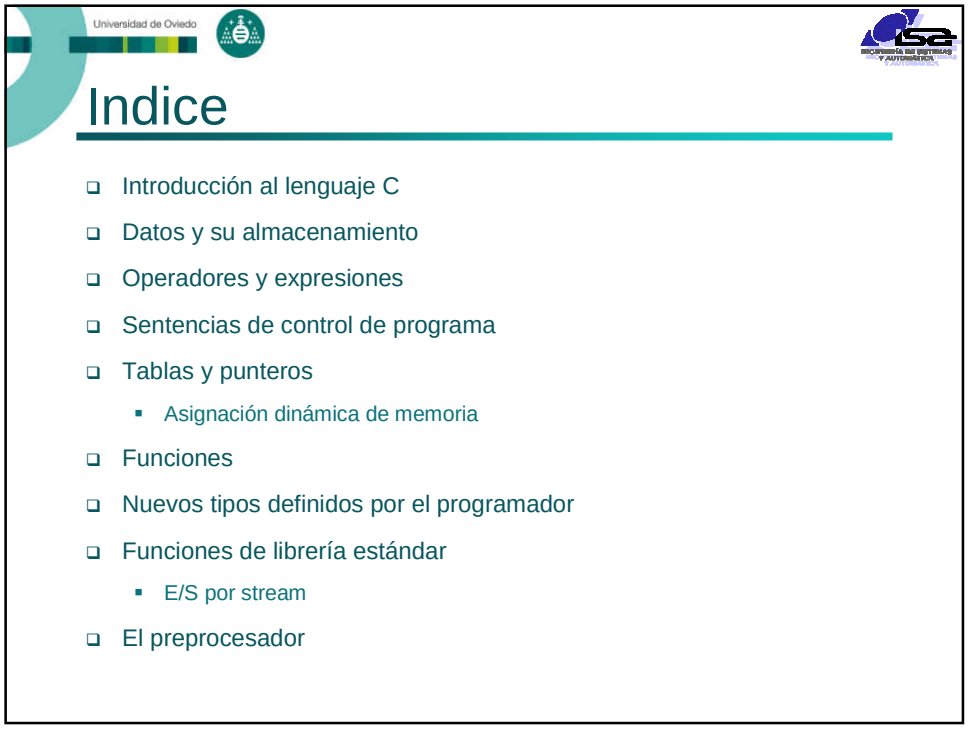

Universidad de Oviedo
Lenguaje C
Ignacio Alvarez García
Octubre - 2006


Universidad de Oviedo
Indice
Introducción al lenguaje C
Datos y su almacenamiento
Operadores y expresiones
Sentencias de control de programa
Tablas y punteros
Asignación dinámica de memoria
Funciones
Nuevos tipos definidos por el programador
Funciones de librería estándar
E/S por stream
El preprocesador

Introducción al lenguaje C

- Lenguaje de nivel medio
  - Características de alto nivel: tipado, estructurado (no totalmente), altamente portable (estándar ANSI-C).
  - Características de bajo nivel: acceso a bits, bytes y direcciones; lenguaje no seguro.
- Diseñado para programadores de sistemas
  - Lenguaje de programación adaptable a múltiples escenarios

Un repaso rápido al C

- Forma típica de un programa en C

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

#define TAM_TABLA       5

int Factorial(int n)
{
    int i,result;
    for (i=1,result=1;i<=n;i++)
        result=result*i;
    return result;
}

main()
{
    int i;
    int x[TAM_TABLA],fact[TAM_TABLA];

    for (i=0;i<TAM_TABLA;i++)
    {
        printf("Introduzca x[%d] = ",i);
        scanf("%d",&x[i]);
        fact[i]=Factorial(x[i]);
        printf("El factorial de %d es %d\n",x[i],fact[i]);
    }
}
```

Directivas de preprocesador

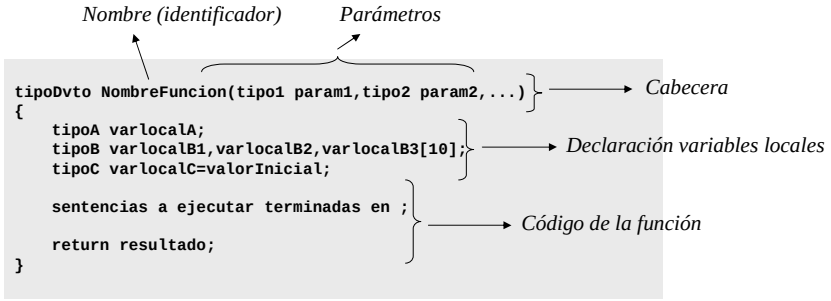
Declaración variables globales (evitar si es posible)

Función

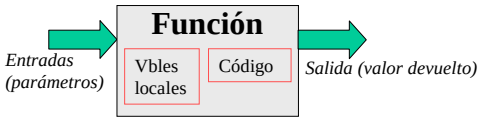
Función principal

Un repaso rápido al C

- Contenidos de una función



- Una función es una caja negra



Un repaso rápido al C

- Tipos básicos de variables:
  - **int** (entero): 3, -117, 0
  - **char** (carácter): ‘a’ ‘b’ ‘c’
  - **float** (real): 3.17, -0.000035, 2.0

- Declaración de variables (terminan en **;**):
  - `int cuenta;  /* Variable tipo entero */`
  - `int num_alumnos=23; /* Id. con valor inicial 23 */`
  - `int valorx,valory,valorz; /* Tres variables enteras */`
  - `int datos[5];  /* Cinco variables enteras consecutivas (tabla o array) */`

Un repaso rápido al C

- Sentencias y expresiones:
  - Todas las sentencias terminan en **;**
  - **Expresión:** todo o parte de una sentencia en la que intervienen operandos y operadores para dar un resultado.
  - **Operandos**: valores de variables, constantes y resultados de función
  - **Operadores:** aritméticos (+ - * / …), relacionales (>= <= == != …), de asignación (=), paréntesis, etc.
  - **Asignación:** operador que modifica el contenido de una variable
  - Ejemplos de expresiones:
    - `y = 2*x + tabla[4]*Factorial(x-1);`
    - `x-5;`
    - `Condicion=(z>=3 && z<=5);`

Un repaso rápido al C

- Sentencias de control de programa
  - Bloque:

```
{
  sentencia1;
  sentencia2;
  …
}
```

  - Alternativa:

```
if (condicion)
  sentenciaSiTRUE;
else
  sentenciaSiFALSE;
```

```
condicion = valor entero :
        0    ≡  Falso
        No 0 ≡  Verdadero
```

  - Bucle while:

```
while (condicion)
  sentenciaARepetir;
```

```
Sentencia se puede
sustituir por un bloque
```

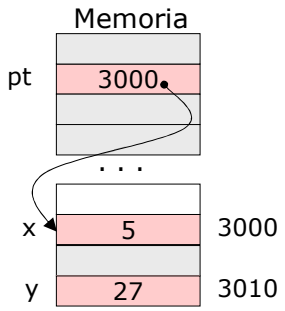
  - Bucle for:

```
for (inicializacion ; condicion_continuidad ; paso )
  sentenciaARepetir;
```

Un repaso rápido al C

- Punteros:
  - Puntero: contiene la dirección de memoria donde se aloja una variable, y el tipo de la misma. Permite acceder a esa posición de forma indirecta.
  - Declaración:
    - tipo *puntero;
  - Acceso a dirección apuntada:
    - *puntero
  - Obtención de dirección de vble:
    - &vble
  - Ejemplo:

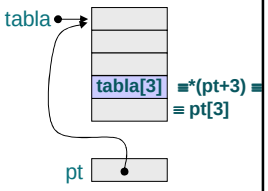
```
int x=5,y=27;
int *pt;
pt = &x;
*pt = 50 ;   // x pasa a valer 50
pt = &y;
*pt = 50 ;   // y pasa a valer 50
```



Un repaso rápido al C

- Tablas:
  - Conjunto de variables almacenadas en memoria de forma consecutiva con un identificador único.
  - Declaración:
    - tipo identif[TAM];
  - TAM debe ser constante
  - Acceso al elemento i-simo:
    - identif[i]

```
int tabla[5] ;
int* pt;

pt=tabla;
```

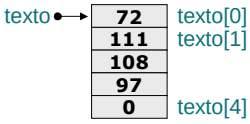


La tabla y el puntero, tras la declaración, son equivalentes: el identificador de tabla es un puntero (constante), o un puntero se puede utilizar como una tabla

Un repaso rápido al C

- Cadenas de caracteres:
  - Son tablas de char terminadas en el valor 0 (carácter nulo).
  - Almacenan secuencias de caracteres de texto codificadas en ASCII (ejemplos: ‘H’ ≡ 72, ‘o’ ≡ 111, ‘\n’ ≡ 10 ≡ retorno de carro).
  - Constantes tipo cadena: “texto” (incluye carácter nulo)

char texto[5] = “Hola”;



Un repaso rápido al C

- Funciones de Entrada / Salida por consola:
  - putchar(char c); → Escribe carácter cuyo código ASCII es c
  - char getchar(); → Espera pulsación de carácter (+INTRO) y devuelve su código ASCII
  - puts(char* cad); → Escribe cadena (todos los caracteres hasta el nulo)
  - gets(char* cad); → Espera pulsación de secuencia de caracteres (+INTRO) y almacena sus códigos ASCII en la tabla apuntada por cad
  - printf(char* fmt,…); → Escribe cadena con formato
  - scanf(char* fmt,…); → Espera pulsación de datos con formato

Un repaso rápido al C

- Funciones de Entrada / Salida por consola:
  - printf(char* fmt,valor1,valor2,…);
    - Escribe en pantalla los caracteres de la cadena fmt, excepto cuando aparece %ALGO. El %ALGO lo sustituye por el valor correspondiente (por orden):
      - %d: entero
      - %f: real (float)
      - %c: carácter
      - %s: cadena de caracteres
  - Ejemplo:

```
int alumnos=12;
char estudios[13]="Programación";
printf("Hay %d alumnos estudiando %s\n",alumnos,estudios);
```

```
Hay 12 alumnos estudiando programación
_
```

Un repaso rápido al C

- Funciones de Entrada / Salida por consola:
  - scanf(char* fmt,ptero_a_valor1,ptero_a_valor2,…);
    - Espera por teclado la pulsación de una secuencia de caracteres terminada en INTRO. Una vez obtiene la secuencia, va extrayendo valores en función de los %ALGO de la cadena de formato y almacenando dichos valores en las direcciones indicadas por los punteros:
  - Ejemplo:

```
int alumnos,grupos;
printf("Introduzca nº de alumnos: ");
scanf("%d",&alumnos);
grupos=alumnos/2;
printf("Puedo hacer %d grupos de 2 alumnos",grupos);
```

```
Introduzca nº de alumnos: 12↵
Puedo hacer 6 grupos de 2 alumnos_
```

Tipos de datos en lenguaje C

- Enteros: Permiten almacenar valores numéricos enteros (o conjuntos de datos representables por enteros). *char*, *short*, *int*, *long,* todos elllos *signed* o *unsigned*.
- Reales: Permiten almacenar valores numéricos fraccionarios. *float, double, long double*.
- Punteros: Permiten almacenar direcciones donde se encuentran datos de otros tipos (incluidos punteros). Se indican como *tipo**.
- Ningún tipo: No almacena nada. *void*.

Tipos de datos enteros

- Enteros sin signo (sólo positivos): codificación ponderada en base 2 con el nº de bits indicado en la tabla de la diapositivia siguiente.
- Enteros con signo (positivos y negativos): positivos como los anteriores, negativos con el bit más significativo (MSB) a 1, indican valor en complemento a 2.

decimal sin signo	**código binario**	**decimal con signo**
0	000 …….. 000	0
1	000 …….. 001	1
2	000 …….. 010	2
3	000 …….. 011	3
. . .	. . .	. . .
$2^{n-1}-2$	011 …….. 110	$2^{n-1}-2$
$2^{n-1}-1$	011 …….. 111	$2^{n-1}-1$
2^{n-1}	100 …….. 000	-2^{n-1}
$2^{n-1}+1$	100 …….. 000	$-2^{n-1}+1$
. . .	. . .	. . .
$2^{n}-3$	111 …….. 101	-3
$2^{n}-2$	111 …….. 110	-2
$2^{n}-1$	111 …….. 111	-1

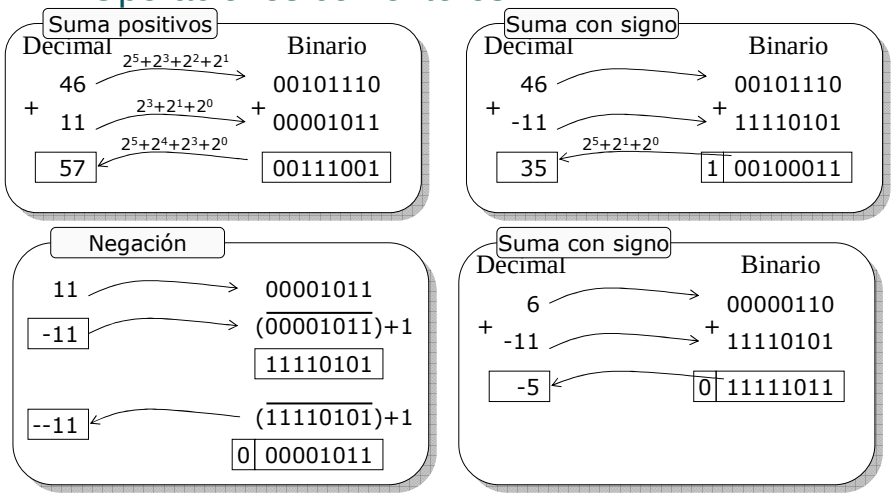
Tipos de datos enteros

- Rango de los enteros

	Nº bits mín	Nº bits VC++	Rango sin signo		Rango con signo	
			Mín (0)	Máx (2^n-1)	Mín (-2^{n-1})	Máx (2^{n-1}-1)
char	8	8	0	255	-128	127
short	>=char	16	0	65535	-32768	32767
int	>char >=short	32	0	4.294.967.295	-2.147.483.648	2.147.483.647
long	>short >=int	32	0	4.294.967.295	-2.147.483.648	2.147.483.647

Tipos de datos enteros

- Operaciones con enteros



Tipos de datos enteros

- Cambio de tamaño de un dato entero

	Operación realizada	Ejemplos
sin signo, nº bits creciente	Ampliar con 0s a la izquierda	12 (8 bits) = 00001100 12 (16 bits) = **00000000**00001100
con signo, nº bits creciente	Ampliar con bit de signo a la izquierda	-11 (8 bits) = 11110101 -11 (16 bits) = **11111111**11110101
ambos, nº bits decreciente	Truncamiento de la parte superior (puede perderse el valor)	258 (16 bits) = 0000000100000010 2 (8 bits) = 00000010

Tipos de datos enteros

- ALU para suma de enteros

Sumador de 1 bit

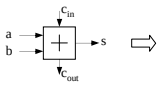
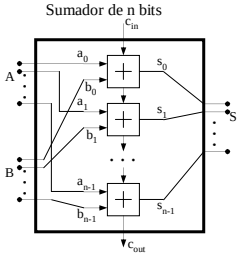


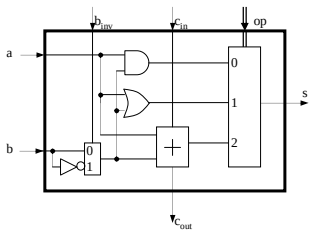
Tabla de verdad

a	b	c_{in}	c_{out}	s
0	0	0	0	0
0	1	0	0	1
1	0	0	0	1
1	1	0	1	0
0	0	1	0	1
0	1	1	1	0
1	0	1	1	0
1	1	1	1	1

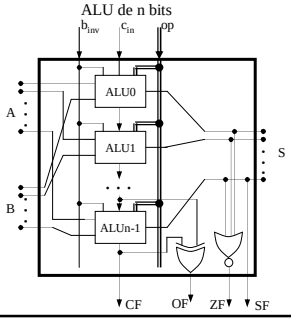
Sumador de n bits



ALU de 1 bit (+, -, AND, OR)



ALU de n bits



Tipos de datos enteros

- Importante:
  - Toda operación con enteros da como resultado otro entero: si hay salida de rango, se produce un resultado válido aunque no correcto.
  - Ej: `char a=100,b=30,c;`
    `c=a+b;` →c pasa a valer -126

- Importante:
  - Aunque todos los ordenadores generan información y disponen de instrucciones de máquina para comprobar salida de rango , el lenguaje C no dispone de mecanismos de comprobación de salida de rango

Tipos de datos enteros

- Definición de constantes de tipo entero:
  - Formato decimal: 27, -2
  - Formato hexadecimal: 0x1B,0xFFFE
  - Sufijos: U=unsigned, L=long
  - Formato código ASCII: ‘a’, ‘;’, ‘2’
    - Secuencias de escape:

Escape Sequence	Represents
\a	Bell (alert)
\b	Backspace
\f	Formfeed
\n	New line
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\'	Single quotation mark
\"	Double quotation mark
\\	Backslash
\?	Literal question mark
\ooo	ASCII character in octal notation
\xhhh	ASCII character in hexadecimal notation

Universidad de Oviedo

Tabla de caracteres ASCII

- Códigos estándar
- Códigos extendidos

Ctrl	Dec	Hex	Char	Code
^@	0	00		NUL
^A	1	01	☺	SOH
^B	2	02	☻	STX
^C	3	03	♥	ETX
^D	4	04	♦	EOT
^E	5	05	♣	ENQ
^F	6	06	♠	ACK
^G	7	07	•	BEL
^H	8	08	◘	BS
^I	9	09	○	HT
^J	10	0A	◙	LF
^K	11	0B	♂	VT
^L	12	0C	♀	FF
^M	13	0D	♪	CR
^N	14	0E	♫	SO
^O	15	0F	☼	SI
^P	16	10	►	SLE
^Q	17	11	◄	CS1
^R	18	12	↕	DC2
^S	19	13	‼	DC3
^T	20	14	¶	DC4
^U	21	15	§	NAK
^V	22	16	▬	SYN
^W	23	17	↨	ETB
^X	24	18	↑	CAN
^Y	25	19	↓	EM
^Z	26	1A	→	SIB
^[	27	1B	←	ESC
^\	28	1C	∟	FS
^]	29	1D	↔	GS
^^	30	1E	▲	RS
^_	31	1F	▼	US

Dec	Hex	Char
32	20	sp
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+
44	2C	,
45	2D	-
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?

Dec	Hex	Char
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T
85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_

Dec	Hex	Char
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}
126	7E	~
127	7F	⌂†

† ASCII code 127 has the code DEL. Under MS-DOS, this code has the same effect as ASCII 8 (BS). The DEL code can be generated by the CTRL+BKSP key.

Dec	Hex	Char
128	80	Ç
129	81	ü
130	82	é
131	83	â
132	84	ä
133	85	à
134	86	å
135	87	ç
136	88	ê
137	89	ë
138	8A	è
139	8B	ï
140	8C	î
141	8D	ì
142	8E	Ä
143	8F	Å
144	90	É
145	91	æ
146	92	Æ
147	93	ô
148	94	ö
149	95	ò
150	96	û
151	97	ù
152	98	ÿ
153	99	Ö
154	9A	Ü
155	9B	¢
156	9C	£
157	9D	¥
158	9E	₧
159	9F	ƒ

Dec	Hex	Char
160	A0	á
161	A1	í
162	A2	ó
163	A3	ú
164	A4	ñ
165	A5	Ñ
166	A6	ª
167	A7	º
168	A8	¿
169	A9	⌐
170	AA	¬
171	AB	½
172	AC	¼
173	AD	¡
174	AE	«
175	AF	»
176	B0	▒
177	B1	▓
178	B2	│
179	B3	┤
180	B4	╡
181	B5	╢
182	B6	╖
183	B7	╕
184	B8	╣
185	B9	║
186	BA	╗
187	BB	╝
188	BC	╜
189	BD	╛
190	BE	┐
191	BF	└

Dec	Hex	Char
192	C0	┴
193	C1	┬
194	C2	├
195	C3	─
196	C4	┼
197	C5	╞
198	C6	╟
199	C7	╚
200	C8	╔
201	C9	╩
202	CA	╦
203	CB	╠
204	CC	═
205	CD	╬
206	CE	╧
207	CF	╨
208	D0	╤
209	D1	╥
210	D2	╙
211	D3	╘
212	D4	╒
213	D5	╓
214	D6	╫
215	D7	╪
216	D8	┘
217	D9	┌
218	DA	█
219	DB	▄
220	DC	▌
221	DD	▐
222	DE	▀
223	DF	α

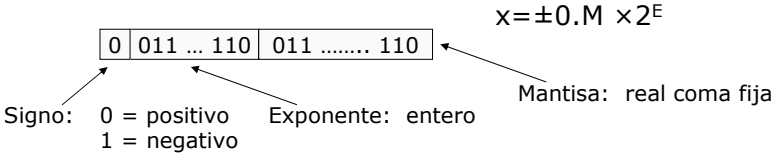
Dec	Hex	Char
224	E0	ß
225	E1	Γ
226	E2	π
227	E3	Σ
228	E4	σ
229	E5	µ
230	E6	τ
231	E7	Φ
232	E8	Θ
233	E9	Ω
234	EA	δ
235	EB	∞
236	EC	φ
237	ED	ε
238	EE	∩
239	EF	≡
240	F0	±
241	F1	≥
242	F2	≤
243	F3	⌠
244	F4	⌡
245	F5	÷
246	F6	≈
247	F7	°
248	F8	∙
249	F9	·
250	FA	√
251	FB	ⁿ
252	FC	²
253	FD	■
254	FE	
255	FF	

Universidad de Oviedo

Tipos de datos reales (coma flotante)

- Formato de los reales: signo / exponente / mantisa

$$x = \pm 0.M \times 2^{E}$$



	Nº bits mín	Nº bits VC++	Codificación partes		Capacidad de representación	
			Exp.	Mant.	Máx	Resolución
float	32	32	8 bits exceso a 127	23 bits 1.Mant	3.4×10^{38}	>=6 dígitos decimales
double	>=float	64	11 bits exceso a 1023	52 bits 1.Mant	1.8×10^{308}	>=15 dígitos decimales
long double	>float >=double	id. a double				

Tipos de datos reales (coma flotante)

- Constantes reales: siempre formato decimal
  - Constantes tipo double: 2.5, 1.0, -4.5, 3.7e-8
  - Constantes tipo float: id. con sufijo F

Tipos de datos reales (coma flotante)

- Operaciones con reales en coma flotante

$$a = \pm 0.Ma \times 2^{Ea}$$
$$b = \pm 0.Mb \times 2^{Eb}$$

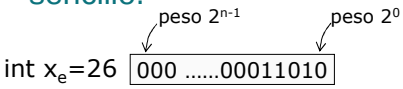
Si Ea>=Eb:

$$a+b = (\pm 0.Ma + \pm 0.\overbrace{00\ldots0}^{Ea-Eb}Mb) \times 2Ea$$

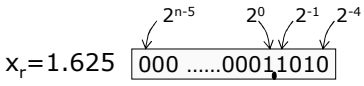
- Importante, las operaciones con reales son más complejas que con enteros: necesitan una ALU más compleja (FPALU) o lógica mediante programa (programas más lentos)

Tipos de datos reales en coma fija

- Reales en coma fija mediante enteros
  - Utilizando tipos de datos enteros se pueden 'simular' reales en coma fija, haciendo operaciones más rápidas o con hardware más sencillo.



  - Si se supone que el punto decimal está entre el bit de peso 3 y 4:



$x_r = x_e \cdot 2^{-4}$

Tipos de datos reales en coma fija

- Operaciones con reales en coma fija mediante enteros (ej. con 'q' bits de parte fraccionaria)
  - Suma/resta: se suman/restan directamente los enteros representativos, y el resultado queda en el mismo formato

$$c_r = a_r + b_r = a_e.2^{-q} + b_e.2^{-q} = \underbrace{(a_e + b_e)}_{c_e}.2^{-q}$$

  - Producto: se multiplican los enteros representativos y el resultado se desplaza 'q' bits a derecha

$$c_r = a_r.b_r = a_e.2^{-q}.b_e.2^{-q} = (a_e.b_e).2^{-2q} = \underbrace{(a_e.b_e.2^{-q})}_{c_e}.2^{-q}$$

  - División: se desplaza el entero representativo del dividendo 'q' bits a izquierda y se divide por el entero rep. del divisor

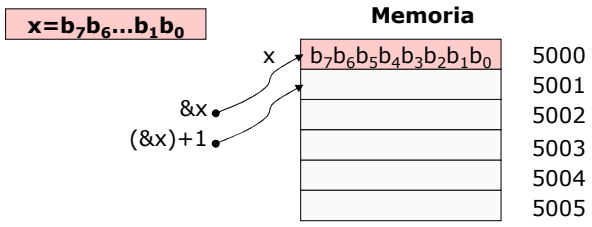
$$c_r = a_r/b_r = a_e.2^{-q}/(b_e.2^{-q}) = (a_e/b_e) = \underbrace{(a_e.2^q/b_e)}_{c_e}.2^{-q}$$

Tipos de datos reales en coma fija

- Rango y resolución:
  - Rango: -2^{n-1-q} a $2^{n-1-q}-1$
  - Resolución: 2^{-q}
- Importante:
  - Se pueden mezclar reales en coma fija codificados con distintos valores de 'q' si se tienen en cuenta los desplazamientos necesarios de manera similar a los casos anteriores.

Almacenamiento de datos y punteros

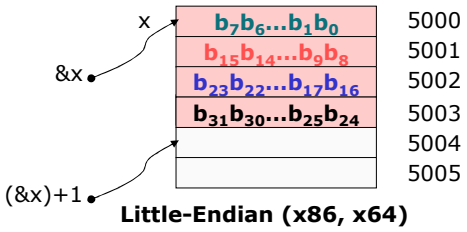
- Si el tipo de dato ocupa 1 byte (1 dirección de memoria):
  - Pesos 7..0 en 1 direc. de memoria
  - La dirección del dato es esa dirección de memoria
  - Los punteros al tipo de dato se incrementan de byte en byte.



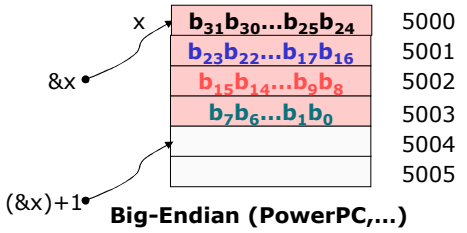
Almacenamiento de datos y punteros

- Si el tipo de dato ocupa más de 1 byte
  - Bytes en direc. de memoria consecutivas
  - La dirección del dato es la más baja
  - El orden de bytes puede ser:
    - Little-endian: byte bajo en dirección baja
    - Big-endian: byte alto en dirección baja
  - Para cada byte, pesos 7..0 como anteriormente

$x=b_{31}b_{30}...b_{24}b_{23}...b_{16}b_{15}...b_8b_7...b_1b_0$



Little-Endian (x86, x64)



Big-Endian (PowerPC,...)

Variables

- Lugares de almacenamiento
  - **Global**: declarada fuera de todos los bloques de código.
    - Tiempo de vida: todo el programa
    - Acceso: accesible desde todas las funciones
  - **Local**: declarada al principio de un bloque de código
    - Tiempo de vida: se 'crea' temporalmente al empezar la ejecución del bloque de código, se 'destruye' cuando se termina su ejecución.
    - Acceso: accesible sólo desde el bloque de código en que se ha declarado
  - **Parámetro de función**: declarada en la cabecera de una función, funcionamiento equivalente a una local
    - Tiempo de vida: se 'crea' temporalmente al llamar a la función, se 'destruye' al finalizar la ejecución de la función.
    - Acceso: sólo es accesible desde la función, aunque su valor inicial se da en la llamada a la misma.

(ver transparencias de funcionamiento del computador para comprender el mecanismo de creación/destrucción de variables locales y parámetros)

Variables

- Modos de almacenamiento
  - *auto*: automático (modo por defecto) → crear en la pila al entrar en el bloque de código.
  - *register*: registro → almacenar en un registro de la CPU (si hay disponibles): acceso más rápido, pero no se puede obtener punteros a la variable.
  - *static*: estática →
    - Si es local, tiene almacenamiento permanente (como una global) aunque sólo es accesible para el bloque de código en que se declaró.
    - Si es global, sólo es accesible para el módulo (archivo .C) en que se declaró.
  - *extern*: externa→ sólo para variables globales, indica que no se reserve espacio de almacenamiento, sino que se encontrará el mismo en otro módulo cuando se enlacen.

Variables

- Especificadores de clase de almacenamiento:
  - *const*: la variable no puede ser modificada (salvo en la inicialización). Utilizado sobre todo en parámetros de función de tipo puntero (ver más adelante)
  - *volatile*: la variable puede ser modificada por causas ajenas al hilo de ejecución → el compilador no la incluirá en ciertas optimizaciones.

(muy importante para Sistemas de Tiempo Real)

Variables

- Declaración:
  - tipo nombre;
  - tipo nombre1,nombre2,nombre3;
  - tipo nombre[NUM ELS];
  - tipo nombre[NUM1][NUM2];
- Inicialización de variables
  - tipo nombre1=valor,nombre2=valor;

Operadores

- Realizan operaciones sencillas con operandos (con tipo) y dan un resultado (con tipo).
- Los operandos pueden ser: variables, constantes o resultado de funciones.
- Tipos de operadores según el nº de operandos: monarios, binarios, ternarios.
- Los operadores se enlazan en expresiones, terminadas en ;
- Tipos de operadores: aritméticos, relacionales, lógicos, de bit, de puntero, de asignación, otros.

Operadores aritméticos

- Binarios (2 operandos):
  - + (suma): num**+**num→num ptero**+**ent→ptero
  - - (resta): num**-**num→num ptero**-**ent→ptero ptero**-**ptero→ent
  - * (producto): num*****num→num
  - / (división): num**/**num→num
  - % (resto de división entera): ent**%**ent→ent
- Monarios:
  - - (negación): **-**num→num

Operadores relacionales

- Binarios (2 operandos):
  - > (mayor que)
  - < (menor que)
  - >= (mayor o igual que)
  - <= (menor o igual que)
  - == (igual que)
  - != (distinto a)
- Todos ellos:
  - num **OPREL** num → ent ent = 0 si falso, ent ≠ 0 si verdadero
  - ptero **OPREL** ptero → ent

Operadores lógicos de palabra

- Binarios (2 operandos):
  - && (y): entA **&&** entB → entR
    - entR ≠ 0 si entA≠0 Y entB≠0
    - entR = 0 si entA==0 O entB==0
  - || (o): entA **||** entB → entR
    - entR ≠ 0 si entA≠0 O entB≠0
    - entR = 0 si entA==0 Y entB==0
- Monarios:
  - ! (negación lógica):

    ! entA → entR
    - entR ≠ 0 si entA==0
    - entR = 0 si entA≠0

Operadores lógicos de bit

- Binarios (2 operandos):
  - & (y): entA **&** entB → entR — $\forall i=0..n-1$: $BIT_i(entR) = BIT_i(entA)$ AND $BIT_i(entB)$
  - | (o): entA **|** entB → entR — $BIT_i(entR) = BIT_i(entA)$ OR $BIT_i(entB)$
  - ^ (o excl.): entA **^** entB → entR — $BIT_i(entR) = BIT_i(entA)$ XOR $BIT_i(entB)$
- Monarios:
  - ~ (negación lógica):

    ~ entA → entR — $\forall i=0..n-1$: $BIT_i(entR) = \overline{BIT_i(entA)}$

Operadores de desplazamiento de bits

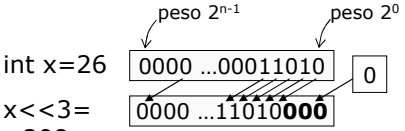
- Binarios (2 operandos):
  - << (desplazamiento a izquierda): entA **<<** entB → entR

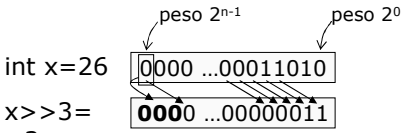
    $\forall$ i=entB..n-1 $BIT_i(entR) = BIT_{i-entB}(entA)$

    $\forall$ i=0..entB-1 $BIT_i(entR) = 0$
  - >> (desplazamiento a derecha): entA **>>** entB → entR

    $\forall$ i=0..entB-1 $BIT_i(entR) = BIT_{i+entB}(entA)$

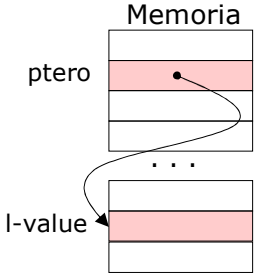
    $\forall$ i=entB..n-1 $BIT_i(entR) =$ { 0: si entA es unsigned ; Bit signo entA: si entA es signed }





Operadores de puntero

- Monarios:
  - & (dirección de): **&**l-value→ptero
  - * (contenido de): ***ptero→l-value**
- Binarios (2 operandos):
  - [] (corchete): ptero**[**ent**]**→l-value ptero**[**ent**]**≡*(ptero+ent)
- Tipo de datos del l-value: tipo
- Tipo de datos del ptero: tipo*



Operadores de asignación

- Asignación simple (binario): l-value **=** valor → result (asigna el nuevo valor al l-value y devuelve ese valor)
- Abreviaturas (binario): l-value **OP=** valor → result (opera l-value con valor, asigna el resultado al l-value y devuelve ese resultado)
  - ***= /= %= += –= <<= >>= &= ^= |=**
- Autoincremento y autodecremento (monarios):
  - Preincremento: **++**l-value → result (l-value=l-value+1, devuelve l-value)
  - Predecremento: **--**l-value → result (l-value=l-value-1, devuelve l-value)
  - Postincremento: l-value**++** → result (devuelve l-value, después: l-value=l-value+1)
  - Postdecremento: l-value**--** → result (devuelve l-value, después: l-value=l-value-1)

Otros operadores

- Paréntesis (monario): **(**exp**)** → valor — prioriza la ejecución
- Molde (monario): **(tipo)** valorA → valorB — devuelve valorB lo más parecido posible a valorA pero del tipo de datos indicado
- Tamaño de (monario): **sizeof(**vt**)** → ent — devuelve nº de bytes necesarios para almacenar vt (vble. o tipo de datos)
- Acceso a estructuras (binarios): **.** y **->**
- Coma (binario): valorA**,**valorB → valorB — evalúa ambos, dve. el 2º
- Condicional (ternario): ent **?** vB **:** vC → valor
  devuelve vB si ent≠0 (true)
  devuelve vC si ent==0 (false)

Expresiones

- Una expresión es una secuencia ordenada de operandos y operadores, terminada en ;
- Una expresión es dividida por el compilador en una secuencia de operaciones individuales, cada una de ellas con sus operandos y su tipo. El orden de esta secuencia se establece según la tabla de precedencia de operadores.
- Si en una operación individual los operandos son de distinto tipo, el de menor capacidad de representación se 'promociona' al de mayor capacidad de representación, y la operación se realiza en ese tipo.

Conversión de tipos en expresiones

- Para toda la expresión:
  char y ***short*** se convierten automáticamente a ***int*** (con o sin signo según el original)

- Operación por operación:

SI un operando es ***long double*** ENTONCES Se convierte el otro a ***long double***
SINO
  SI un operando es ***double*** ENTONCES Se convierte el otro a ***double***
  SINO
    SI un operando es ***float*** ENTONCES Se convierte el otro a ***float***
    SINO
      SI un operando es ***unsigned long*** ENTONCES Se convierte el otro a ***unsigned long***
      SINO
        SI un operando es ***long*** ENTONCES Se convierte el otro a ***long***
          (salvo que el otro sea ***unsigned int*** y no se pueda representar como ***long***;
          en este caso, ambos se convierten a ***unsigned long***)
        SINO
          SI un operando es ***unsigned int*** ENTONCES Se convierte el otro a ***unsigned int***

- Asignación:
  Se convierte el valor de la derecha al tipo del 'lvalue' a asignar

Precedencia de operadores

- Orden de evaluación:

Precedencia	Tipo	Operadores	Asociación
Mayor		`() [] -> .`	izda. a dcha.
	Monario	`! ~ ++ -- - (molde) * & sizeof`	dcha. a izda.
	Binario	`* / %`	izda. a dcha.
	Binario	`+ -`	izda. a dcha.
	Binario	`<< >>`	izda. a dcha.
	Binario	`< <= > >=`	izda. a dcha.
	Binario	`== !=`	izda. a dcha.
	Binario	`&`	izda. a dcha.
	Binario	`^`	izda. a dcha.
	Binario	`|`	izda. a dcha.
	Binario	`&&`	izda. a dcha.
	Binario	`||`	izda. a dcha.
	Ternario	`?:`	dcha. a izda.
	Binario	`= += -= *= /= &= etc`	dcha. a izda.
Menor		`,`	izda. a dcha.

Sentencias de control de programa

- Bloque de código: hace ver a un grupo de sentencias como una sola. Las variables locales sólo se pueden declarar al comienzo de un bloque de código:

```
{
  decl. vbles. locales;
  sentencias;
}
```

En las siguientes páginas, donde se indica "Sentencia" puede ser sustituido por un bloque de código con múltiples sentencias encerradas entre llaves

Sentencias de control de programa

- Condicionales: permite ejecutar unas sentencias u otras (selección en tiempo de ejecución)

```
if (exp. ent.)
    Sentencia si verdadero;
else
    Sentencia si falso;
```

Opcional

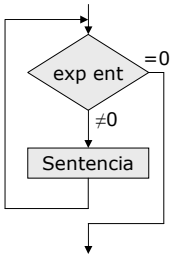
```
switch (exp)
{
    case cte1:
      Sentencias;
      break;
    case cte2:
      Sentencias;
      break;

    default:
      Sentencias;
      break;
}
```

Opcional

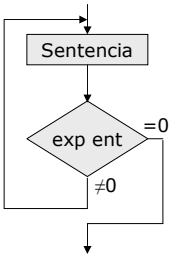
Sentencias de control de programa

- Bucles: permiten ejecutar repetidamente una(s) sentencia(s) mientras se cumpla una condición:

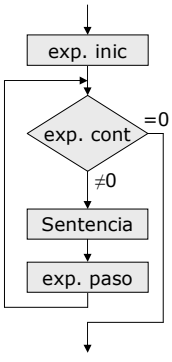
```
while (exp. ent.)
    Sentencia;
```




```
do
    Sentencia;
while (exp. ent.);
```




```
for (exp. inic ; exp. cont ; exp. paso)
    Sentencia;
```



Sentencias de control no estructuradas

- Sentencia **break**: termina bruscamente la ejecución de un bucle o un switch

```
for (exp. inic ; exp. cont ; exp. paso)
{
    Sentencias;
    if (condicion)
        break;
    Sentencias;
}
```

- Sentencia **goto**: salto no estructurado

```
goto aquí:
. . . ;
aquí:
. . . ;
```

NO USAR

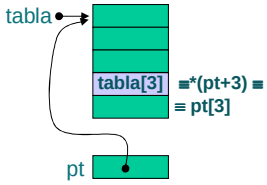
- Sentencia **continue**: evita la ejecución de sentencias de un bucle

```
for (exp. inic ; exp. cont ; exp. paso)
{
    Sentencias;
    if (condicion)
        continue;
    Sentencias;
}
```

Tablas y punteros

- Tabla (o array):
  - Conjunto de variables del mismo tipo, consecutivas en memoria, que se acceden mediante un único identificador y un índice.
  - El identificador es la dirección de inicio de la tabla.
  - El tamaño de la tabla se define al declararla (constante de tiempo de compilación).
  - El índice indica la posición del elemento a partir del comienzo de la tabla: desde 0 hasta n-1.
  - El lenguaje C no comprueba la salida de rango en el índice.
  - Una vez declarada una tabla, su dimensión no es utilizada por el compilador (sí por el programador).
  - Una vez declarada una tabla, su identificador equivale a un puntero.
- Puntero (pointer):
  - Variable que indica una dirección de memoria, y permite acceder a ella de forma indirecta.
- Dualidad tabla puntero:

```
int tabla[5] ;
int* pt;

pt=tabla;
```

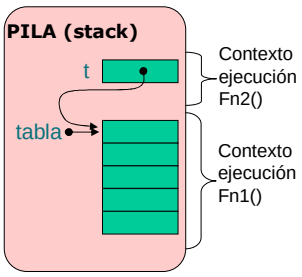


La tabla y el puntero, tras la declaración, son equivalentes: el identificador de tabla es un puntero (constante), o un puntero se puede utilizar como una tabla

Tablas y punteros

- El tamaño de la tabla (valor constante) sólo es usado en el momento de declarar la variable (local o global, no parámetro de función).
- A partir de la declaración, la tabla sólo es un puntero (constante).
- Por tanto, una función no recibe o devuelve tablas completas, sino punteros a su comienzo.

```
Fn2(int t[5])
{
  t[i]=…
}

Fn1()
{
   int tabla[5] ;

   Fn2(tabla);
}
```

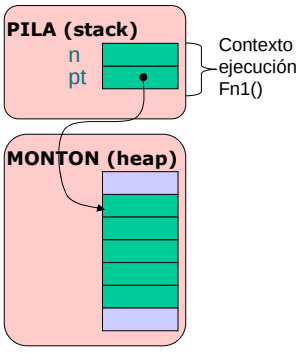


Fn2(int t[5]) ≡ Fn2(int t[ ]) ≡ Fn2(int* t)

Asignación dinámica de memoria

- Si se desea una tabla de ‘n’ datos, con ‘n’ variable en tiempo de ejecución: **malloc() / free()**
- **void* malloc(int nbytes)** ⇒ busca espacio en el montón (heap) para nbytes consecutivos: si se encuentra, reserva y devuelve dirección (puntero) de comienzo
- **free(void* pt)** ⇒ comprueba si hay memoria asignada en el heap que comience en la dirección indicada: si existe, la libera para nuevo uso

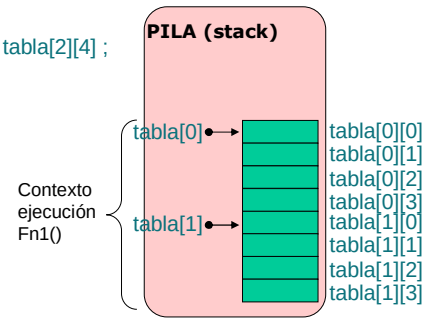
```
Fn1()
{
   int *pt;
   int n;

   n=5;
   pt=(int*) malloc(n*sizeof(int));
   if (pt!=NULL)
      Usar  pt[i]…
   free(pt);
}
```



Tablas bidimensionales estáticas

- Declaración:
  tipo vble[DIM1][DIM2];
- DIM1 y DIM2 constantes
- La tabla se organiza en memoria por filas (ver figura).
- Una vez declarada, el compilador necesita la dirección de comienzo y DIM2 para calcular el acceso al elemento i,j
- **vble[i][j]** es un 'tipo'
- **vble[i]** es un puntero a 'tipo' (tipo*) que apunta al comienzo de la fila i-sima
- **vble** es un puntero a puntero a 'tipo' (tipo**)

```
Fn1()
{
   int tabla[2][4] ;

}
```



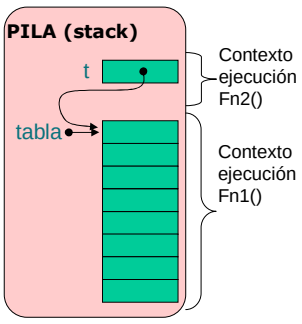
El compilador calcula la posición de tabla[i][j] como:

tabla[0]+i*DIM2+j

Tablas bidimensionales estáticas

- Si se pasa una tabla bidimensional estática a una función, sólo se pasa el puntero a principio (como para tablas unidimensionales) …
- Pero es necesario que la función conozca la DIM2 para que el compilador calcule correctamente el acceso al elemento i,j

```
Fn2(int t[2][4])
{
   t[ i ][ j ]=…
}

Fn1()
{
    int tabla[2][4] ;

    Fn2(tabla);
}
```



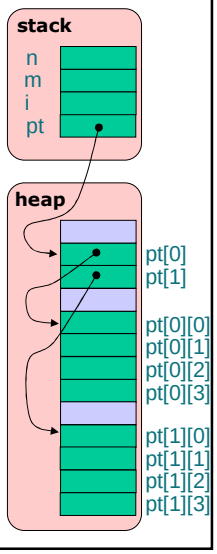
Fn2(int t[2][4]) ≡ Fn2(int t[ ][4]) ≢ Fn2(int t[ ] [ ])

Tablas bidimensionales dinámicas

- Asignar memoria dinámica para una tabla de punteros a cada fila
- Asignar memoria dinámica para cada que cada puntero a fila apunte a una fila completa
- Acceder a los elementos con [i][j] (pero ahora la posición i,j no depende del nº de filas o de columnas)

```
Fn1()
{
   int **pt;
   int n,m,i;

   n=2; m=4;
   pt=(int**) malloc(n*sizeof(int*));
   for (i=0;i<n;i++)
     pt[ i ]=(int*) malloc(m*sizeof(int));

   pt[ i ][ j ] …

   for (i=0;i<n;i++)
      free(pt[ i ]);
   free(pt);
}
```

Comprobar !=NULL



Asignación estática vs dinámica

- Estática:
  - Tamaño conocido en tiempo de compilación
  - Puntero constante
  - Espacio ocupado en el stack (vbles locales)
  - Tablas bidimensionales requieren tamaño de 2ª dimensión
  - Tablas bidimensionales compactas
- Dinámica
  - Tamaño asignable en tiempo de ejecución
  - Puntero variable
  - Espacio ocupado en el heap
  - Tablas bidimensionales con acceso independiente de su tamaño (más genérico)
  - Tablas bidimensionales no compactas
  - Tablas bidimensionales necesitan más memoria
  - Tablas bidimensionales no precisan el mismo tamaño para todas las filas

Funciones

- Formato:

```
tipodvto NombreFuncion(tipo1 par1,tipo2 par2,…)
{
    tipoA vbleA1,vbleA2,vbleA3;
    tipoB vbleB1,vbleB2;
    …
    sentencias;
    return valor;
}
```

- Llamada:

  NombreFuncion(valorP1,valorP2,…) puede formar parte como operando de cualquier expresión que admita el tipo 'tipodvto'

- Prototipo:

```
tipodvto NombreFuncion(tipo1 par1,tipo2 par2,…);
```

- Múltiples return:

  Una función puede utilizar varias sentencias **return**. La función termina y regresa cuando se alcanza la primera de ellas.

Funciones y tablas

- Tablas como parámetros y valores devueltos:
  - Una función nunca recibe una tabla como parámetro, sólo un puntero al principio.
  - Una función nunca puede devolver una tabla, sólo un puntero a principio.
  - Para pasar / devolver tablas: parámetro puntero a principio.
  - Pasar tablas que no se modifican en la función: puntero const

¡¡¡¡INCORRECTO!!!!

```
int* FuncionDevuelveTabla(...)
{
   int tabla[5];

   ... Acceso a tabla[i]...
   return tabla;
}
```

CORRECTO

```
void FuncionDevuelveTabla(int t[])
{
    ... Acceso a t[i]...
}
void FuncionUsaTabla(const int x[])
{
    ... Acceso a x[i] pero sin
        asignar a x[i] ...
}
```

Funciones recursivas

- Una función puede llamarse a sí misma
- La solución de un algoritmo reduciéndolo a soluciones más simples de sí mismo se llama recursividad.
- Ejemplo Factorial:
  - Solución iterativa: `n! = n*(n-1)*(n-2)*…*1`
  - Solución recursiva: $n! = \begin{cases} n*(n-1)! & \text{si } n>1 \\ 1 & \text{si } n<=1 \end{cases}$

```
int Factorial(int n)
{
    if (n>1)
        return n*Factorial(n-1);
    else
        return 1;
}
```

Funciones: la función main()

- La función main():
  - Es la 1ª que se ejecuta, y cuando finaliza se termina el programa. Por tanto, las variables locales de main() existen durante todo el programa.
  - Es llamada por una función de la librería standard, que realmente es la 1ª que se ejecuta.
  - Puede tener parámetros y resultado, que permiten intercambiar datos entre nuestro programa y el programa cargador (loader):
    - Prototipo de main():
      int main(int argc,char* argv[]);
    - Parámetros de main:
      - int argc: Cuenta de argumentos de línea de comandos
      - char* argv[]: Tabla de cadenas de caracteres que contienen dichos argumentos
    - Resultado devuelto: entero que puede ser utilizado por el programa cargador para conocer el resultado

Funciones: la función main()

- **Ejemplo de uso: llamada a MIPROG.EXE con argumentos en línea de comandos.**

```
C:\Cptr\Debug> MIPROG.EXE texto1 "este es el texto 2" 47
```

- **Código para recuperar los argumentos y devolver valor según su número:**

```
int main(int argc,char* argv[])
{
     int i;
     for (i=0;i<argc;i++)
          printf("Argumento %d=%s\n",i,argv[i]);
     if (argc<3)
     {
          printf("Error en nº de argumentos");
          return -1;
     }
     return 0;
}
```

Desde exlorador de carpetas de Windows: Crear acceso directo al ".exe", seleccionar propiedades del acceso directo, y en la casilla "Destino" de la pestaña "Acceso directo" escribir el comando completo con argumentos
Desde el depurador de Visual Studio: Seleccionar propiedades del proyecto, propiedades de configuración, depuración, argumentos del comando

```
C:\Cptr\Debug> MIPROG.EXE texto1 "este es el texto 2" 47
Argumento 0=MIPROG.EXE
Argumento 1=texto1
Argumento 2=este es el texto 2
Argumento 3=47
```

Punteros a función

- Permiten llamar a una función de manera indirecta: la función llamada es decidida en tiempo de ejecución y no de compilación.
- Declaración:

```
tipoDvto (*ptero)(tipo1,tipo2,…);
```

- Asignación:

```
ptero = NombreDeFuncionSinParentesis ;
```

- Uso:

```
ptero(valorp1,valorp2,…) -> Llama a la función apuntada con
                            los parámetros deseados y devuelve
                            valor del tipoDvto
```

- Ejemplo de uso:

```
int Suma(int a,int b)
{
     return a+b;
}

int Producto(int a,int b)
{
     return a*b;
}
```

```
FnX()
{
     int (*ptero)(int,int);
     int x=3,y=7,z;

     ptero=Suma;
     z=ptero(x,y);        // z=10

     ptero=Producto;
     z=ptero(x,y);        // z=21
}
```

Punteros a función

- **Uso típico:**
  - Parámetros en llamadas a funciones de librería que tienen que utilizar una función provista por el programador.
- **Ejemplo:**
  - qsort(): ordena una tabla según el criterio indicado por una función de usuario.

```
void qsort (void *base, size_t num, size_t width,
        int (*fn_cmp)(const void *,const void *));

int FnCompara(const void* el1,const void* el2)
{
     const float *ptEl1=(const float*) el1;
     const float *ptEl2=(const float*) el2;
     float fracEl1=*ptEl1-floor(*ptEl1);
     float fracEl2=*ptEl2-floor(*ptEl2);

     if (fracEl1>fracEl2)
           return 1;
     if (fracEl1<fracEl2)
           return -1;
     return 0;
}

FnX()
{
     float datos[9]={1,4,2,9,3,5,8,7,6};
     qsort(datos,9,sizeof(float),FnCompara);
}
```

Definición de nuevos tipos de datos

- El programador puede definir “nuevos” tipos de datos:
  - Estructuras: para agrupar varias variables bajo un solo tipo.
  - Uniones: para ver la misma zona de memoria de varias formas.
  - Enumeraciones: para variables que sólo pueden tomar un conjunto de valores.
  - Redefinición de tipos existentes: para dar nuevos nombres a tipos existentes.
- Estos tipos de datos no son nuevos en cuanto a las operaciones que permiten realizar (salvo operadores . y ->), sino diferentes formas de organizar los tipos existentes.

Nuevos tipos de datos: estructuras

- Estructuras: permiten agrupar varias variables (campos) en una sola, de forma que pueden accederse como conjunto o individualmente.
- Declaración:

```
struct nombre
{
   tipoA campo1A,campo2A;
   tipoB campo1B;
   …
} ;
```

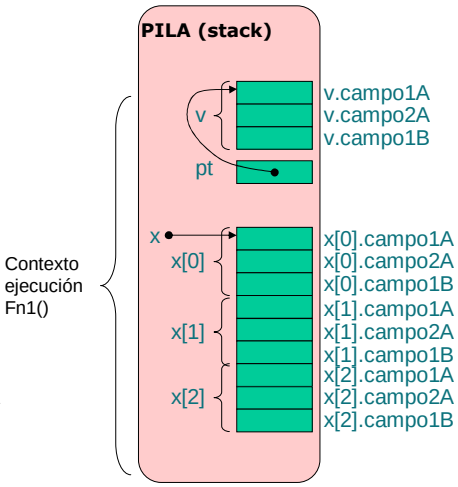
Atención: aquí sí va un **;**

- Las sentencias anteriores no son código ejecutable: sólamente declaran el nuevo tipo de datos “struct nombre”.
- Los campos de la estructura se acceden mediante los operadores . y ->:
  - Para una variable de tipo struct nombre: `vble.campo1A`
  - Para un puntero al tipo struct nombre: `ptero->campo1A`

Nuevos tipos de datos: estructuras

- Declaración y uso de variables tipo estructura:

```
…  Fn1(…)
{
   struct nombre v;
   struct nombre x[3];
   struct nombre *pt;
   …
   pt= &v;
}
```



`v.campo1A`
`x[i].campo1A`
`pt->campo1A`

} Son a todos los efectos variables de tipo tipo1A

Ejemplo de uso de estructuras

- Números complejos:

```
struct complex
{
 float re,im;
} ;

struct complex ProdComplex(struct complex a,struct
complex b)
{
 struct complex result;

 result.re=a.re*b.re-a.im*b.im;
 result.im=a.im*b.re+a.re*b.im;
 return result;
}

…
…  Fn1(…)
{
 struct complejo c1,j={0.0F,1.0F};
 …
 c1=ProdComplex(c1,j);
}
```

Recordar este ;

Inicialización de estructura

Ejemplo de uso de estructuras

- Datos de alumnos:

```
struct datosAlumno
{
 char nombre[20];
 char apellidos[40];
 int edad;
 float calificacion;
} ;

…  Fn1(…)
{
 int i;
 struct datosAlumno alumnos[12];
 ...
 for (i=0;i<12;i++)
 {
      printf("Nombre alumno %d: ",i);
      gets(alumnos[i].nombre);
      ...
 }
}
```

Estructuras para listas enlazadas

- Tipo de datos para una lista enlazada:

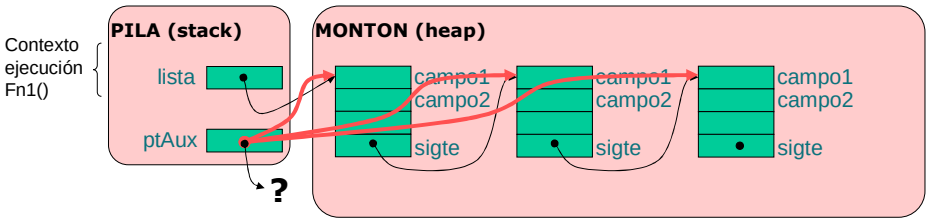
```
struct elemLista
{
   tipo1 campo1;
   tipo2 campo2;
   . . .
   struct elemLista *sigte;
} ;
```

- Acceso a la lista: puntero a principio

```
Fn1()
{
   struct elemLista *lista;
   … Creación de la lista …
   … Acceso a los elementos de la lista …
   … Destrucción de la lista …
}
```

Acceso a una lista enlazada existente

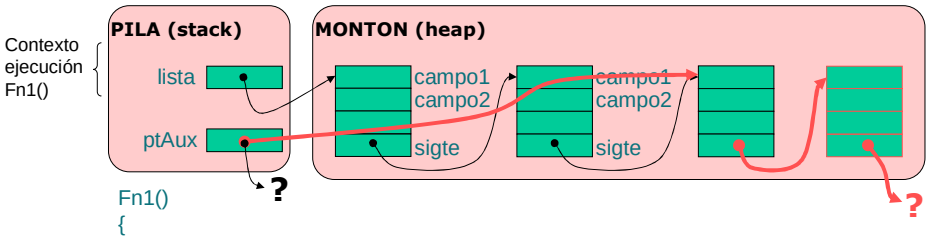
- Se accede a cualquier elemento a través del puntero al 1er elemento de la lista




```
Fn1()
{
   struct elemLista *lista,*ptAux;
   … Creación de la lista …
   for (ptAux=lista;ptAux!=NULL;ptAux=ptAux->sigte)
      … Acceso a ptAux->campo1, ptAux->campo2, … ;
   … Destrucción de la lista …
}
```

Añadir al final de una lista enlazada existente

- Se busca el último elemento de la lista, su puntero se asigna a un nuevo espacio de memoria (malloc), y se rellenan sus datos




```
Fn1()
{
   struct elemLista *lista,*ptAux;
   . . .
   for (ptAux=lista;ptAux->sigte!=NULL;ptAux=ptAux->sigte) ;
   ptAux->sigte=(struct elemLista*) malloc(sizeof(struct elemLista));
   ptAux->sigte->sigte=NULL;
   ptAux->sigte->campo1 = ..... ;
   . . .
}
```

Nuevos tipos de datos: uniones

- Uniones: permiten agrupar varias variables (campos) que ocupan una misma zona de memoria.
- Declaración:

```
union nombre
{
   tipoA campo1A,campo2A;
   tipoB campo1B;
   ...
} ;
```

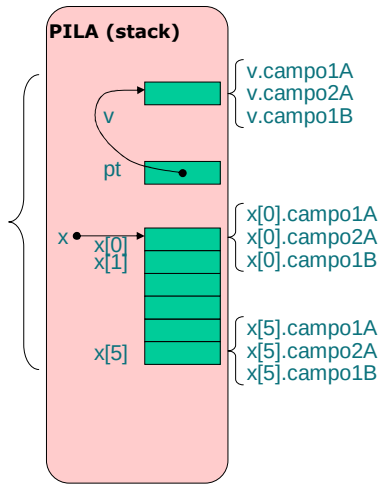
Atención: aquí sí va un **;**

- Las sentencias anteriores no son código ejecutable: sólamente declaran el nuevo tipo de datos “struct nombre”.
- Los campos de la unión se acceden mediante los operadores . y ->:
  - Para una variable de tipo union nombre: `vble.campo1A`
  - Para un puntero al tipo union nombre: `ptero->campo1A`

Nuevos tipos de datos: uniones

- Declaración y uso de variables tipo unión:

```
…  Fn1(…)
{
   union nombre v;
   union nombre x[6];
   union nombre *pt;
   …
   pt= &v;
}
```



`v.campo1A`
`x[i].campo1A`
`pt->campo1A`
} Son a todos los efectos variables de tipo tipo1A

Ejemplo de uso de uniones

- Acceso a una zona de datos en varios formatos:

```
union numReal
{
 unsigned int entero;
 float real;
} ;

…  Fn1(…)
{
 union numReal r;
 unsigned int i,mask;
 r.real=-5.25;
```

r | 11000000 10101000 00000000 00000000 | r.real= -5.25

r.entero=$2^{31}+2^{30}+2^{23}+2^{21}+2^{19}$ = 3232235520

```
 printf("Codificación binaria de %f: ",r.real);
 for (i=0,mask=1<<31;i<32;i++,mask>>=1)
      putchar((r.entero & mask) ? '1' : '0');
}
```

Ejemplo de uso de uniones

- Agrupación de datos de diferentes tipos:

```
struct coche
{
 int plazas;
 ...
} ;
struct camion
{
 float tara,pma;
 ...
} ;
struct moto
{
 ...
} ;
union vehiculos
{
   struct coche co;
   struct camion ca;
   struct moto mo;
} ;
```

```
struct vehiculo
{
   char matricula[8];
   int cilindrada;
   int tipo;
   union vehiculos datos;
} ;
```

```
#define ES_COCHE     0
#define ES_CAMION    1
#define ES_MOTO 2

… Fn1(…)
{
 struct vehiculo p_m[12];
 ...
 for (i=0;i<12;i++)
 {
   ...
   if (p_m[i].tipo==ES_COCHE)
   {
      p_m[i].datos.co.plazas=5;
   }
   if (p_m[i].tipo==ES_CAMION)
   {
      p_m[i].datos.ca.pma=5100;
   }
  }
}
```

Nuevos tipos de datos: enumeraciones

- Enumeraciones: permiten definir constantes para variables que sólo pueden tomar un conjunto de valores.
- Declaración:

```
enum nombre {idcte0,idcte1,… };
```

- Uso:

```
enum nombre v1,v2;
v1=idcte0;
v2=v1+1;
if (v2==idcte1)
   …
```

- El compilador asigna constantes 0, 1, … a los identificadores idcte0,idcte1, etc., y los trata como enteros.
- El compilador no comprueba salida de rango ni tiene tratamiento de lista circular (“wrap around”).

Ejemplo de uso de enumeraciones

- Tipo de datos días de la semana:

```
enum diasSem {lunes,martes,miercoles,jueves,viernes,sabado, domingo
} ;

enum diasSem Incrementa(enum diasSem d)
{
 d++;
 if (d>domingo)
      d=lunes;
}

…  Fn1(…)
{
 enum diasSem hoy,manyana,pasado;

 hoy=sabado;
 manyana=Incrementa(hoy);
 pasado=Incrementa(manyana);
}
```

Nueva denominación de tipos

- typedef permite dar un nuevo nombre a un tipo existente:

```
typedef tipoExistente tipoNuevo;
```

- Ejemplos de uso:
  - Acortar nombres de tipos:

```
typedef struct datosAlumnos strDatos;
strDatos alumnos[12];
```

  - Usar nombres genéricos para cambiar fácilmente en compilación:

```
typedef float real;
real ProductoEscalar(real a[],real b[]);
```

Funciones de librería estándar

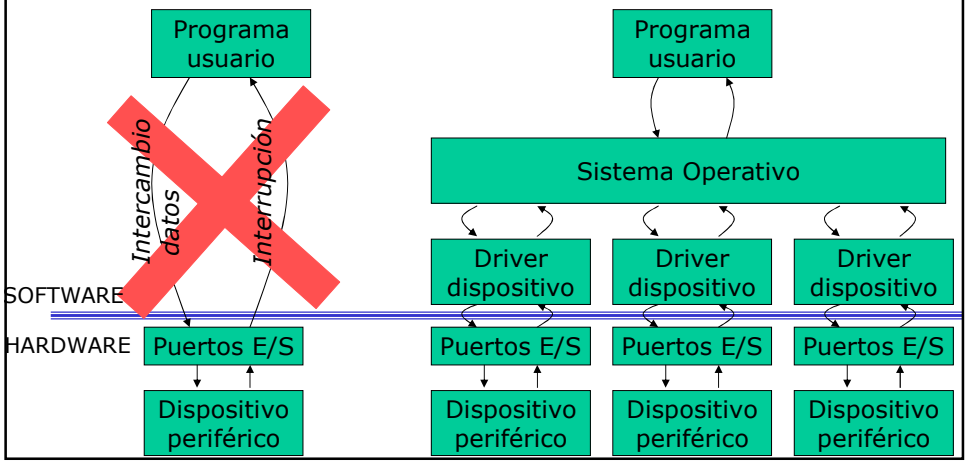
- Las funciones de librería acompañan siempre a los compiladores de lenguaje C aunque estrictamente no forman parte del lenguaje.
- Incluir archivos de cabecera para utilizarlas.
- Clasificación:
  - Funciones matemáticas en coma flotante
  - Funciones de asignación dinámica de memoria
  - Funciones de E/S por stream
  - Funciones de cadena de caracteres
  - Funciones de conversión de datos
  - Funciones de caracteres
  - Funciones de manipulación de memoria
  - Funciones de sistema y entorno
  - Funciones de fecha y hora
  - Funciones de búsqueda y ordenación
  - Generación de números aleatorios

Funciones matemáticas

- Trabajan siempre con tipo de datos double
- Utilizar moldes para asignar a float
- Incluir <math.h>
- Funciones más utilizadas:
  - fabs
  - sin,cos, tan, asin,acos,atan,atan2
  - sqrt, exp, log, log10, pow
  - floor,ceil

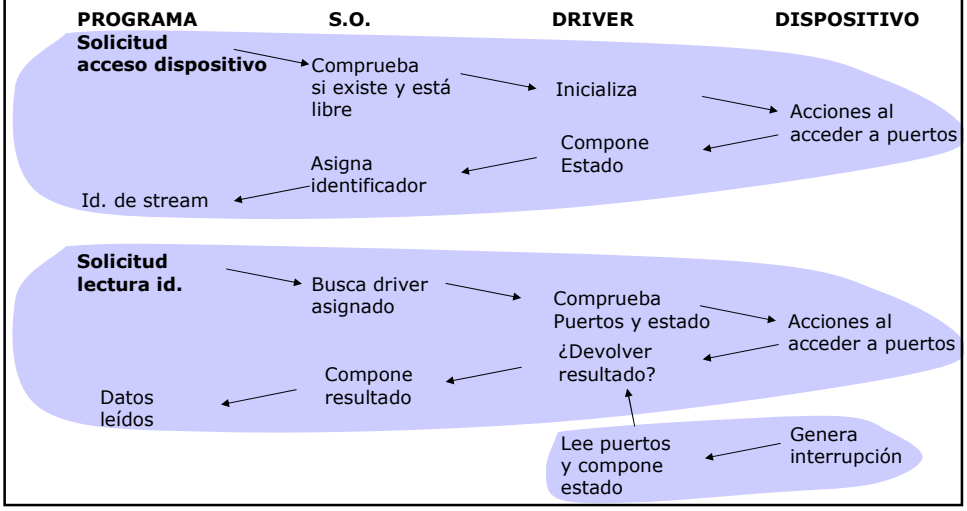
Funciones de E/S por stream

- Permiten intercambiar información con los dispositivos de E/S mediante streams



E/S por stream

- Funcionamiento de la E/S por stream



E/S por stream

- Stream: corriente de datos → se envían o reciben un conjunto ordenado de bytes al/del dispositivo
- Todos los dispositivos son tratados de igual forma: se leen o escriben corrientes de bytes
- Tipos de streams:
  - De entrada / salida / entrada y salida
  - Con memoria (archivos en dispositivos de almacenamiento) / sin memoria (resto de dispositivos)
  - Orientados a texto / binarios
- Denominación de dispositivos: cadena de caracteres (ejs "COM1", "/dev/ttyS0", "C:\usuario\datos.txt")
- El S.O. se encarga de la organización lógica de los dispositivos (nombres, drivers, derechos de acceso, libre/ocupado,…).
- Los drivers se encargan del acceso físico a los dispositivos (leer/escribir puertos, gestión de interrupciones).

Funciones para manejo de streams

- Permiten asignar identificador a stream, leer/escribir sobre identificador, liberar stream asociado.
- Incluir <stdio.h>
- Identificador de stream: variable tipo FILE*.
- Funciones de apertura/cierre de stream:
  - FILE* fopen(const char* name,const char* mode);
    - name: nombre del dispositivo. Si no existe, supone que es un archivo en un dispositivo de almacenamiento.
    - mode: modo de apertura: lectura/escritura/añadir, texto/binario (ver ayuda en VC++).
  - int fclose(FILE* fid);

Funciones para manejo de streams

- Funciones de E/S en streams de texto:
  - E/S de caracteres:
    - int fgetc(FILE* fid); // Lee 1 carácter
    - int fputc(int car,FILE* fid); // Escribe 1 carácter
  - E/S de cadenas:
    - char *fgets(char *str,int n,FILE *fid); // Lee 1 línea
    - int fputs(const char* str,FILE* fid); // Escribe 1 línea
  - E/S con formato:
    - int fscanf(FILE* fid,char *fmt,…); // Lee con formato
    - int fprintf(FILE *fid,char *fmt, …); // Escribe con formato

Streams por defecto para consola

- Streams por defecto: permiten realizar E/S a consola con printf(), scanf(),gets() ,puts(), getchar(),putchar():
  - stdin: entrada estándar
  - stdout: salida estándar
  - stderr: salida de error
- Ej: printf(…) ≡ fprintf(stdout,…)
- Estos streams están conectados por defecto a la consola, en modo texto (stdin a lectura, stdout y stderr a escritura), pero pueden redirigirse a otros dispositivos (ej. a archivo):
  - Miprograma.exe < entrada.txt > salida.txt

Funciones para manejo de streams

- Funciones de E/S en streams binarios:
  - Lectura de datos:
    - int fread(void* buffer, size_t size, size_t count, FILE *fid);
      // Lee un conjunto de count*size bytes del stream y los
      // almacena en el mismo formato en el buffer
  - Escritura de datos:
    - int fwrite(const void* buffer, size_t size, size_t count, FILE *fid);
      // Envía al stream un conjunto de count*size bytes indicados
      // a partir de la dirección de memoria buffer
  - En ambos casos:
    - Las funciones devuelven el nº de elementos (bytes/size) realmente escritos o leídos.

Funciones para manejo de streams

- Funciones auxiliares para streams con almacenamiento (archivos):
  - int feof(FILE* fid); // ¿Alcanzado fin de archivo?
  - void rewind(FILE* fid); // Vuelve al principio del archivo
  - long ftell(FILE *fid); // Devuelve posición (en bytes)
    // desde el principio del stream
  - int fseek(FILE *fid, long offset, int origin);
    // Coloca en posición (bytes) deseada
    // desde:
    // el principio: origin = SEEK_SET
    // la posición actual: origin = SEEK_CUR
    // el final: origin = SEEK_END

Funciones para manejo de streams

- Funciones auxiliares para streams:
  - int ferror(FILE* fid); // Dve. código del último error (0 = ok)
  - void clearerr(FILE* fid); // Borra código del último error
  - int fflush(FILE *fid); // Vacía buffers intermedios con los que el
    // S.O. gestiona la E/S física del dispositivo
    // asociado al stream:
    // Si estaba abierto para salida, asegura
    // que dicha salida es escrita físicamente.
    // Si estaba abierto para entrada, elimina
    // los datos de los buffer intermedios.

Funciones de E/S sin buffer

- Se hace la E/S directamente al driver, sin conversiones ni buffers intermedios.
- Identificador de dispositivo: tipo de datos int
- Dispositivos por defecto: stdin ≡ 0, stdout ≡ 1, stderr ≡ 2
- Funciones:
  - open(), creat(), close(), read(), write(), tell(), …
- No forman parte del estándar ANSI, aunque sí están presentes en la mayoría de implementaciones.
- Dan acceso al uso de funciones más especializadas en entornos estilo Unix:
  - ioctl(), fcntl(), select()

Funciones de cadena de caracteres

- Trabajan con tablas de char terminadas en el caracter nulo '\0'.
- Incluir <string.h>
- Funciones más utilizadas
  - int strlen(const char* cad); *// Devuelve longitud de cadena*
  - char* strcpy(char* dst,const char* src); *// Copia src en dst*
  - char* strncpy(char* dst,const char* src, int n); *// Copia máx n caracteres de src en dst (no incl. nulo)*
  - char* strcat(char* dst,const char* src); *// Añade src al final de dst*
  - char* strncat(char* dst,const char* src,int n); *// Añade máx n caracteres de src al final de dst (incl. nulo)*
  - …

Funciones de cadena de caracteres

- …Funciones más utilizadas
  - int strcmp(const char* str1,const char* str2); *// Compara a nivel de códigos ASCII. Devuelve: 0 ⇒ iguales, <0 ⇒ str1 es anterior, >0 ⇒ str1 es posterior.*
  - int strncmp(const char* str1,const char* src,int n); *// Id. con máximo n caracteres*
  - char* strerror(int errnum); *// Obtiene cadena de caracteres de error*
  - char* strchr(const char* str,char c); *// Busca caracter c en cadena cad*
  - char* strstr(const char* str,const char* substr); *// Busca subcadena substr en cadena str*
  - Otras funciones de búsqueda (ver ayuda VC++): strcspn(), strpbrk(), strtok(), strspn(), …

Funciones de conversión de datos

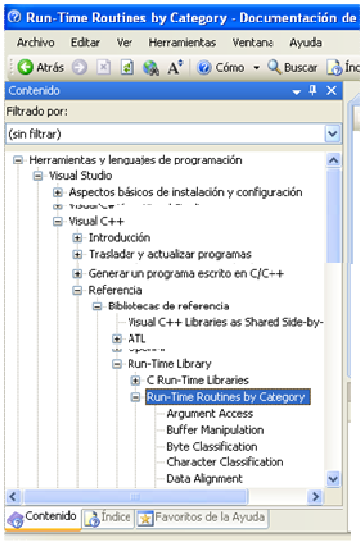
- Convierten cadenas de caracteres de/hacia tipos de datos numéricos.
- Funciones más utilizadas
  - int atoi(const char* cad); *// Obtiene entero decimal*
  - double atof(const char* cad); *// Obtiene valor real*
  - double strtod(const char* cad,char** ptEnd); *// Obtiene valor real*
    // y puntero al final de conversión
  - int sscanf(const char* cad,const char* fmt,…); *// Id. a scanf() pero*
    // obteniendo datos de una cadena
  - char* itoa(int num); *// Obtiene cadena decimal para entero*
  - char* ftoa(double num); *// Obtiene cadena decimal para real*
  - int sprintf(char* cad,const char* fmt,…); *// Id. a printf() pero*
    // almacenando en una cadena

Funciones de manipulación de memoria

- Permiten manejar buffers (zonas) de memoria.
- Incluir <string.h>
- Funciones más utilizadas
  - void* memcpy(void* dest,const void* fte,int n);
    // Copia n bytes desde direc. fte en direc. dest
  - void* memmove(void* dest,const void* fte,int n);
    // Id. a memcpy() pero comprobando solapamiento
  - void* memset(void* dest,char c,int n);
    // Pone n bytes a partir de la dirección dest al valor c
  - int memcmp(const void* b1,const void* b2,int n);
    // Compara n bytes a partir de la dirección b1 con n bytes
    // a partir de la dirección b2. Devuelve 0 si todos son iguales,
    // valor < 0 si el primero diferente es menor en b1,
    // valor > 0 si el primero diferente es mayor en b1

Otras funciones de librería

- Ver ayuda en:



El preprocesador

- Se puede indicar ciertas acciones previas al compilado mediante directivas de preprocesador
- Las directivas de preprocesador deben ocupar una línea comenzando por #
- Directivas más usuales:
  - # include <archivo.h> ó “archivo.h”
  - # define IDENTIF texto
  - # pragma …
  - # ifxxx …
  - # else
  - # endif
  - # error

El preprocesador

- Uso de macros
  - Sin parámetros:

    ```
    #define   N   50
    ```
  - Con parámetros:

    ```
    #define   MIN(a,b) (a>b) ? a : b
    ```

- Macros con parámetros:
  - Aunque su invocación es similar a la de una función, su funcionamiento es diferente. Usar únicamente para código sencillo y repetitivo (ej. MIN)
  - Evitar problemas en expansión de macros poniendo paréntesis a los parámetros y a la expresión completa:

    ```
    #define   MIN(a,b) ( ((a)>(b)) ? (a) : (b) )
    ```

El preprocesador

- Compilación condicional:

  ```
  #if   condición_evaluable_en_tiempo_de_compilación
  …           // Código a compilar si condición != 0
  #else       // Alternativa opcional
  …           // Código a compilar si condición == 0
  #endif
  ```

- Alternativas a #if:

  ```
  #ifdef      identif_constante
  #ifndef     identif_constante
  ```

- Diferenciar entre
  - #if : Se produce la comprobación en tiempo de compilación. El código que no cumple no forma parte del ejecutable.
  - if () : Se produce la coprobación en tiempo de ejecución. Todo el código forma parte del ejecutable

El preprocesador

- Ejemplos de compilación condicional:

```
#define DEPURANDO    0        // cambiar por 1 ó 2

…
#if   DEPURANDO > 0
      printf("Resultado parcial 1 = ",…);
#endif

#if   DEPURANDO > 1
      printf("Resultado parcial 2 = ",…);
#endif
```

El preprocesador

- Compilación condicional para evitar inclusiones múltiples: