INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

Universidad de Oviedo

Programación en lenguaje C

Ignacio Alvarez García

Julio - 2013

Indice

- **Introducción al computador y el lenguaje C**
- Datos, expresiones y algoritmos
- Funciones
- Tablas y punteros
- Cadenas de caracteres
- Operaciones con valores binarios
- Preprocesador y compilación separada
- E/S en archivos y dispositivos
- Criterios de buena programación

El computador

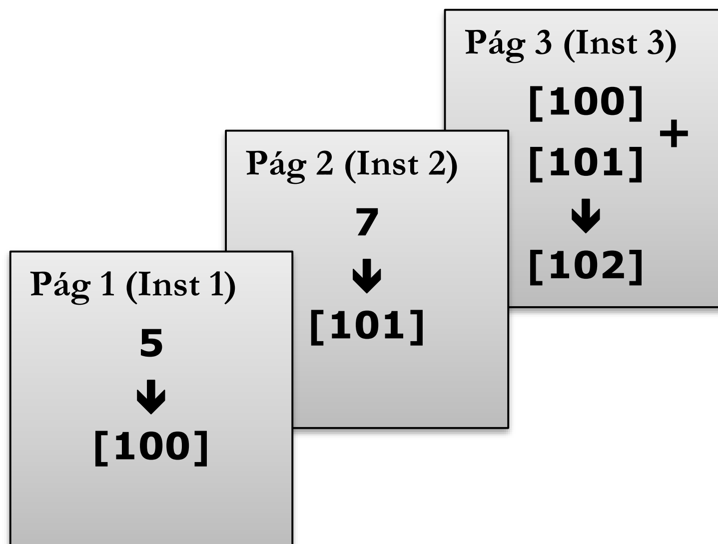
- Un computador es una máquina de ejecución secuencial de instrucciones:
  - Cada **instrucción** es una operación muy sencilla, que utiliza **datos** fuente y almacena el resultado en **datos** destino:
    - MOV dest,src ; equivale a dest = src
    - ADD dest,src ; equivale a dest = dest + src
    - NEG dest ; equivale a dest = -dest
    - DIV dest,src ; equivale a dest = dest / src
    - …
  - Los **datos** (o **variables**) son lugares de almacenamiento de información: guardan valores que pueden ser requeridos o modificados posteriormente.
  - Un **Programa** es una secuencia de **instrucciones** sencillas.
    - Ej: calcular de la media de 3 datos x1,x2,x3 y guardarla en el dato m.

```
MOV  m,x1    ; m = x1
ADD  m,x2    ; m = m+x2 = x1+x2
ADD  m,x3    ; m = m+x3 = x1+x2+x3
DIV  m,3     ; m = m/3 = (x1+x2+x3)/3
```

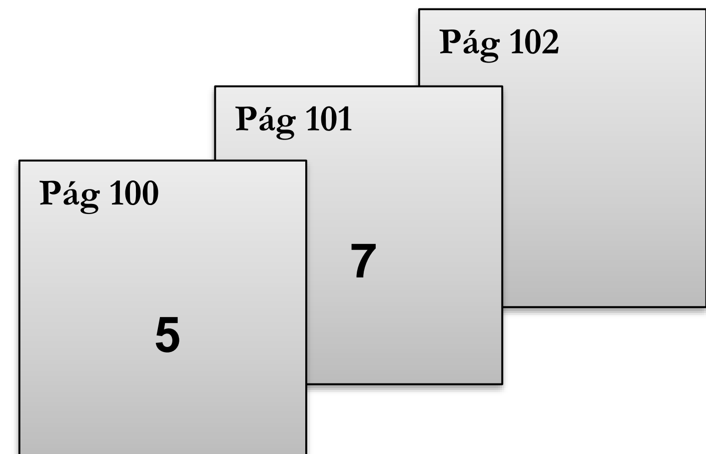
El computador: una analogía

- Se puede asimilar el funcionamiento de un computador al de un niño que sabe entender instrucciones básicas y realizar operaciones aritméticas:
  - En cada hoja de una libreta se le indica una instrucción básica
  - La instrucción se refiere a una operación a realizar con los datos apuntados en otra(s) hoja(s) de la libreta
  - Todas las hojas están numeradas

Páginas de instrucciones

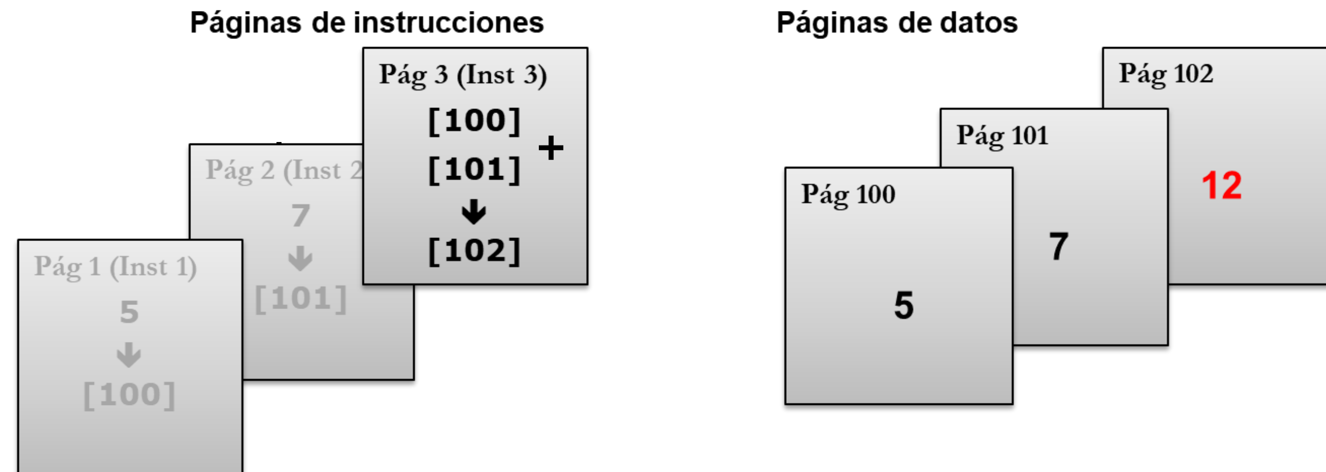


Páginas de datos



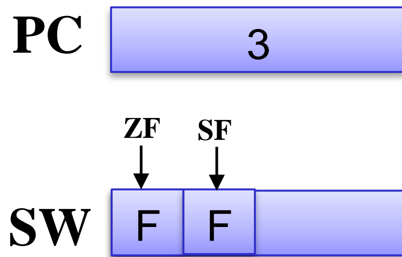
El computador: analogía niño

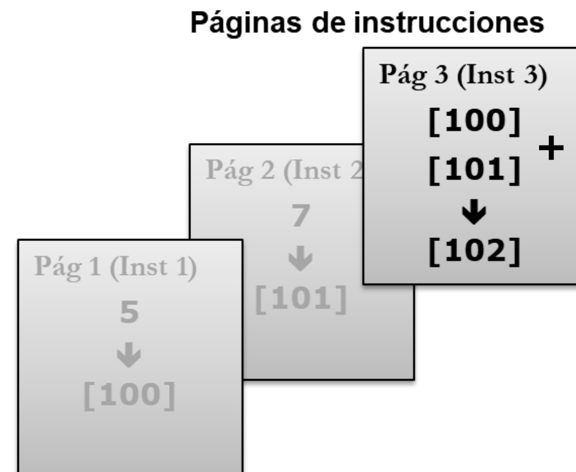
- Para cada página de instrucción, el niño:
  - Decodifica la instrucción indicada en la página:
    - Pág 3: sumar Pág [100] + Pág [101] y almacenar el resultado en Pág [102]
  - Obtiene los operandos fuente de las páginas de datos
    - El contenido de la pág. [100] es 5, el contenido de la pág. [101] es 7
  - Realiza la operación pedida con los operandos
    - En el ejemplo: 7 + 5 → 12
  - Guarda el resultado en su página de datos destino
    - Anota 12 en la pág. [102] (si había otro valor, es reemplazado por el nuevo)
  - Pasa a la siguiente página de instrucción y repite el mismo ciclo

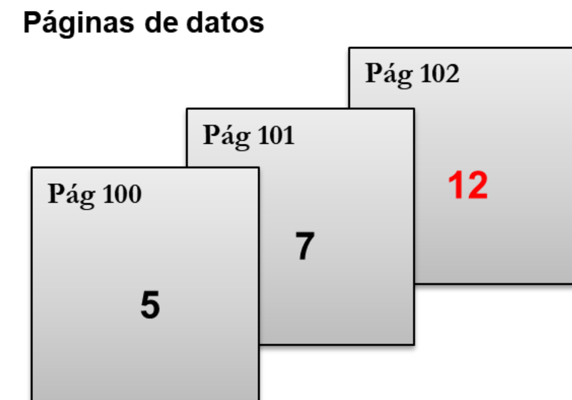


El computador: analogía niño

- Además de las páginas de la libreta, el niño dispone de "post-it" para anotaciones rápidas:
  - En un post-it (PC) anota el número de página de instrucción que debe ejecutar. Va incrementando este valor en 1 cada vez que ejecuta una instrucción.
  - En un post-it (SW) anota un resumen del resultado de la última instrucción aritmética:
    - **ZF** = **T**rue (el resultado ha sido cero) / **F**alse (el resultado ha sido distinto de cero)
    - **SF** = **T**rue (el resultado ha sido negativo) / **F**alse (el resultado no ha sido negativo)

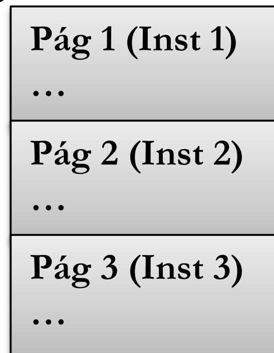


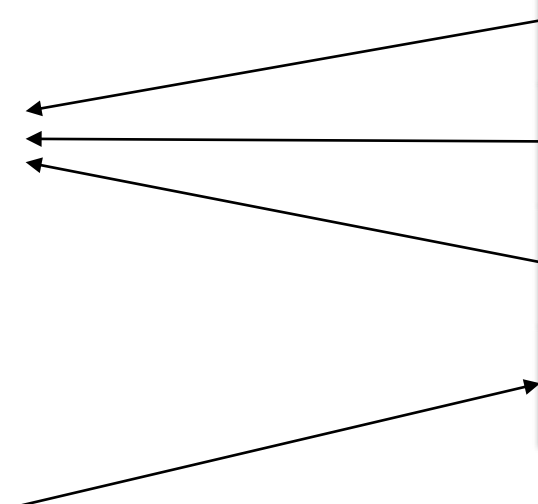


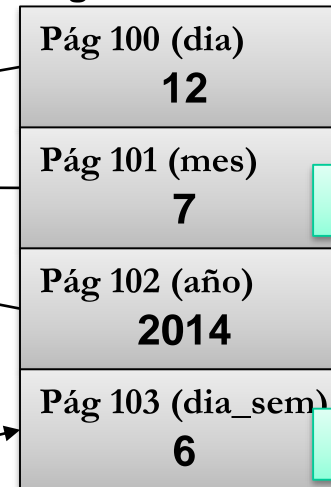


El computador: analogía niño

- Un cálculo complejo estará basado en una secuencia de instrucciones (algoritmo), tras la cual se obtiene el resultado
- Ejemplo:
  - ¿Qué día de la semana ha sido el 12 de Julio de 2014?

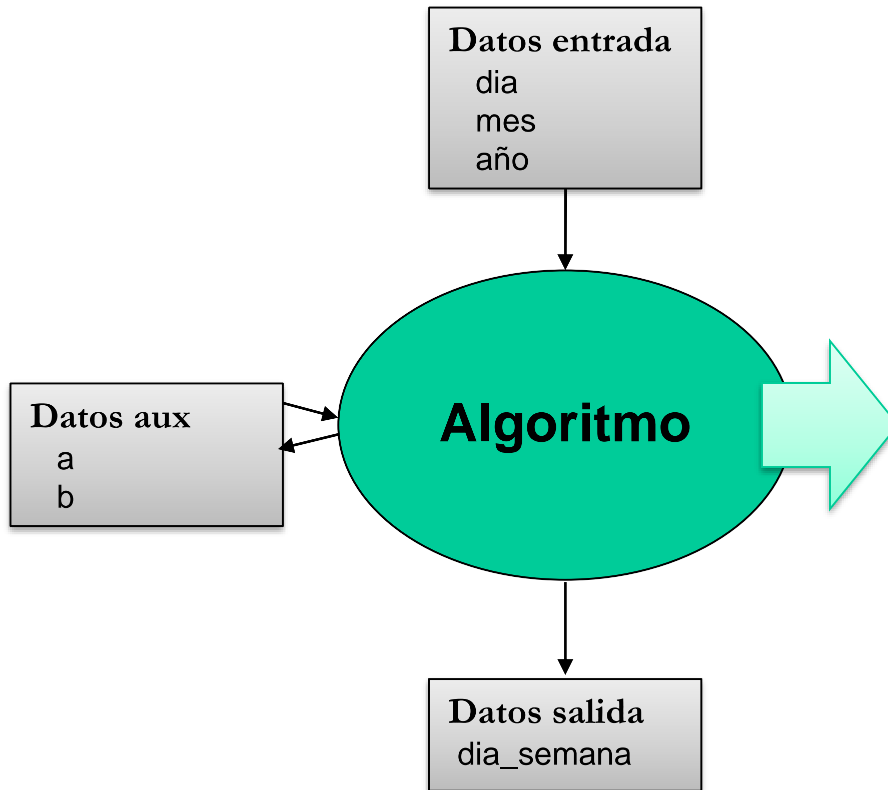






El computador: analogía niño

- Para realizar un algoritmo puede ser necesario utilizar:
  - Datos auxiliares
  - Instrucciones de salto (no ejecutar la instrucción siguiente)



https://www.matesymas.es/algoritmo-de-zeller/

(Todos los / representan cociente de división entera)
Si mes<=2
  mes=mes+12
  año=año-1
Si No
  mes=mes-2
a=año mod 100   *(mod = resto división entera)*
b=año / 100
dia_sem=(700 + (mes*26-2)/10 + dia +   a +
  a / 4 + b / 4 – b * 2) mod 7

(dia_sem es un entero que representa al dia de la semana tal que domingo es 0, lunes es 1… sábado es 6)

(Todos los / representan cociente de división entera)
Si mes<=2
 mes=mes+12
 año=año-1
Si No
 mes=mes-2
a=año mod 100 *(mod = resto división entera)*
b=año / 100
dia_sem=(700 + (mes*26-2)/10 + dia + a + a / 4 + b / 4 – b * 2) mod 7 (*domingo es 0, lunes es 1… sábado es 6)*

animacion.mp4

que describir el algoritmo y sus datos con todo detalle.

Páginas de instrucciones

Pág 1 (Inst 1)
mes - 2

Pág 2 (Inst 2)
Si SF es F Salta a Pag 6

Pág 3 (Inst 3)
mes + 12 → mes

Pág 4 (Inst 4)
año + 1 → año

Pág 5 (Inst 5)
Salta a Pag 7

Pág 6 (Inst 6)
mes-2 → mes

Pág 7 (Inst 7)
resto de año/100 → a

Pág 8 (Inst 8)
cociente de año/100 → b

PC

Pág 9 (Inst 9)
700 → tmp1

Pág 10 (Inst 10)
mes * 26 → tmp2

Pág 11 (Inst 11)
tmp2 – 2 → tmp2

Pág 12 (Inst 12)
cocte de tmp2/10 → tmp2

Pág 13 (Inst 13)
tmp1 + tmp2 → tmp1

Pág 14 (Inst 14)
tmp1 + dia_mes → tmp1

Pág 15 (Inst 15)
tmp1 + a → tmp1

Pág 16 (Inst 16)
cocte de a/4 → tmp2

SW

Pág 17 (Inst17)
tmp1 + tmp2 → tmp1

Pág 18 (Inst 18)
cocte de b/4 → tmp2

Pág 19 (Inst 19)
tmp1 + tmp2 → tmp1

Pág 20 (Inst 20)
b * 2 → tmp2

Pág 21 (Inst 21)
tmp1 - tmp2 → tmp1

Pág 22 (Inst 22)
resto de tmp1/7 → dia_sem

Páginas de datos

Pág 100 (dia_mes)
12

Pág 101 (mes)
7

Pág 102 (año)
2014

Pág 103 (dia_sem)

Pág 104 (a)

Pág 105 (b)

Pág 106 (tmp1)

Pág 107 (tmp2)

El computador: analogía niño

- Codificación de datos
  - Para interpretar instrucciones y datos, el niño utiliza códigos
  - Para instrucciones:
    - Interpreta las secuencias de **símbolos** dibujados en las páginas de datos como instrucciones
    - Los **símbolos** ([100], 5, →, + , …) surgen de un código comúnmente adoptado: lenguaje alfanumérico y matemático
  - Para datos enteros:
    - Los dígitos son representados por **símbolos** (0 , 1 , 2 , … 9), que expresan una cantidad (numeración arábiga)
    - Un número se compone de dígitos ordenados, donde el orden representa el peso (implícito) del dígito
    - Codificación ponderada en base 10 (decimal):

10^3	10^2	10^1	10^0
↓	↓	↓	↓
2	**0**	**1**	**4**

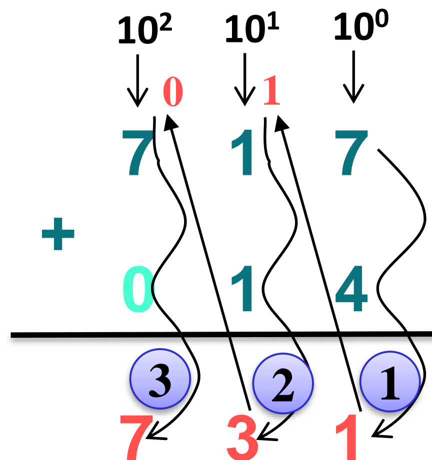
Universidad de Oviedo

INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

El computador: analogía niño

Operaciones con datos

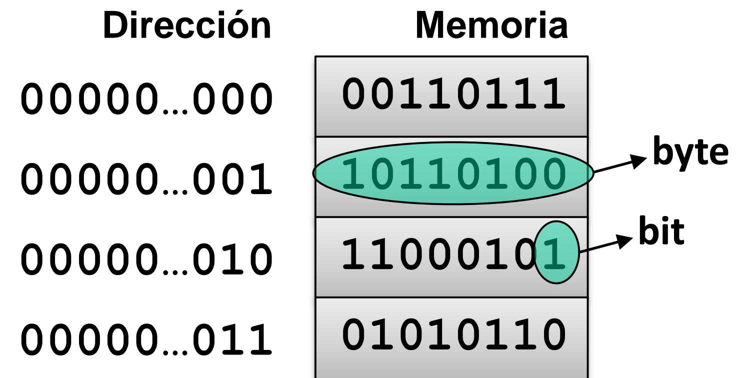
- Para operar con los datos, el niño utiliza automatismos
- Los automatismos están basados en el sistema de numeración decimal



1. Suma dígitos de menor peso, generando como resultado el dígito del mismo peso y un acarreo para sumar al peso siguiente si se excede la base de numeración
2. Suma dígitos del peso siguiente junto al acarreo del peso anterior, generando como resultado el dígito del mismo peso y un acarreo para sumar al peso siguiente si se excede la base de numeración
3. Repite lo anterior para todos los dígitos, de menor a mayor peso. Si uno de los sumandos no tiene dígitos en un determinado peso, se asigna cero

El computador: analogía niño

- Un computador hace lo mismo que un niño, pero:
  - Sus únicos **símbolos** son 0 y 1 (tanto para instrucciones como para datos) ya que usa electrónica digital (off/on)
  - Su **libreta** se denomina **memoria principal**
  - Las **páginas** de su libreta se denominan **direcciones**, y cada una sólo puede almacenar 8 símbolos (bits) = 1 byte
  - Sus **post-it** se denominan **registros**
  - Sus automatismos para el cálculo se basan en lógica de compuertas (AND, OR, …)



El computador: codificación binaria

Codificación de datos en binario

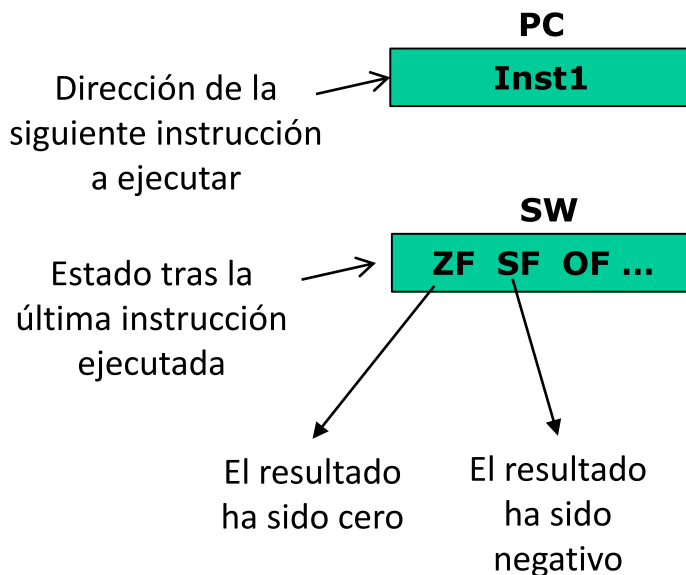
- Una misma cantidad (ej. 23) se puede expresar en múltiples sistemas de numeración
- Cada sistema de numeración utiliza unos determinados símbolos y describe su interpretación
  - Codificación ponderada en base 10 (decimal): **1 4** (pesos 10^1 10^0)
  - Números romanos: **X I V** (valores 10, -1, -5)
  - Codificación ponderada en base 2 (binario): **1 1 1 0** (pesos 2^3 2^2 2^1 2^0)
- Algunos sistemas de numeración (ponderados) favorecen la realización simple de operaciones:

$$\begin{array}{r} 14 \\ +\ 27 \\ \hline 41 \end{array} \qquad \begin{array}{r} 01110 \\ +\ 11011 \\ \hline 101001 \end{array}$$

Mismas cantidades expresadas en sistemas de numeración diferentes

El computador: codificación binaria

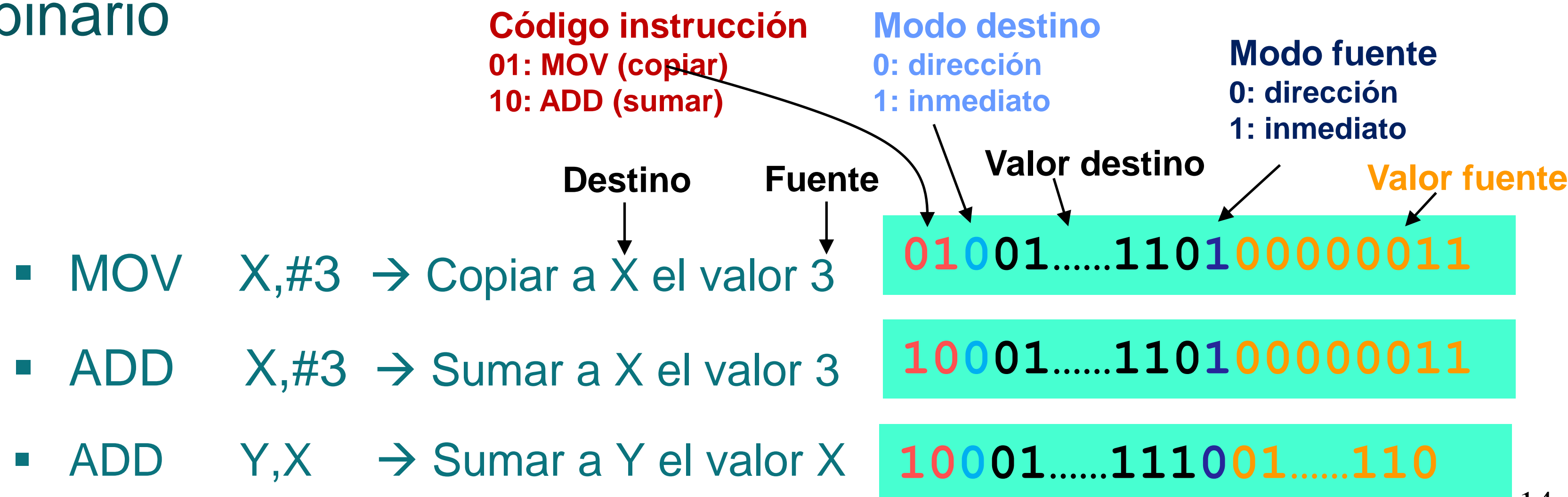
- Datos e instrucciones se codifican en binario.
- Datos e instrucciones se almacenan en memoria, organizada en direcciones (0, 1, … , 2^{n-1}).
- Cada dirección almacena 1 byte (8 bits)
- Si un valor a codificar “no cabe” en una sola dirección, ocupará varias consecutivas
- Dispone de lugares de almacenamiento específicos (registros) para datos internos (PC, SW, otros)



Dirección	Contenido	
000..00	01101100	
000..01	00101110	
000..10	11111100	
	. . .	
Inst1:	MOV X,#3	Instrucciones codificadas en binario
Inst2:	MOV Z,X	
Inst3:	ADD Z,Y	
	. . .	
X:	0	Datos codificados en binario
Y:	7	
Z:	2	
	. . .	

El computador: ejecución de programas

- Un programa es una secuencia estática de instrucciones, que se ejecutan secuencialmente.
- Cada instrucción es muy básica: usa 1, 2 ó 3 operandos y realiza una copia/suma/resta/multiplicación o división
- Los operandos se expresan como contenidos de direcciones de memoria o valores inmediatos
- Tanto instrucciones como operandos se codifican en binario



Universidad de Oviedo

INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

El computador: ejecución de programas

- Para poder alterar la ejecución puramente secuencial del programa:
  - Tras cada instrucción aritmética se modifica de forma automática el registro de estado (SW), indicando si el resultado fue 0 o no (ZF), negativo o no (SF), …
  - Instrucciones específicas (de salto) tienen como destino el contador de programa (PC)

- CMP X,#3 → Comparar X con el valor 3
  (Modifica ZF, SF, según resultado)

Código instrucción
11: Comparar
Modo destino
0: dirección
1: inmediato
Valor destino
Modo fuente
0: dirección
1: inmediato
Valor fuente

`11001……110100000011`

- JRELIFZ #2 → Si ZF activo, PC = PC+2

Código instrucción
00: Saltar Si Flag
Si ZF
Sumar a PC
Modo fuente
1: inmediato
Valor fuente

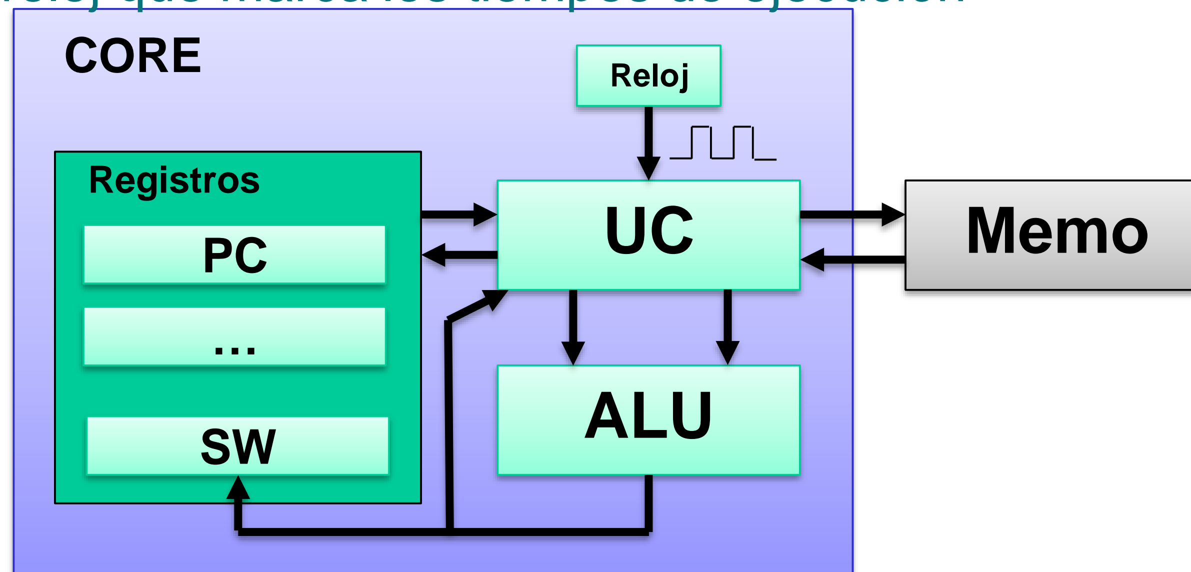
`001000000110000010`

Universidad de Oviedo

INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

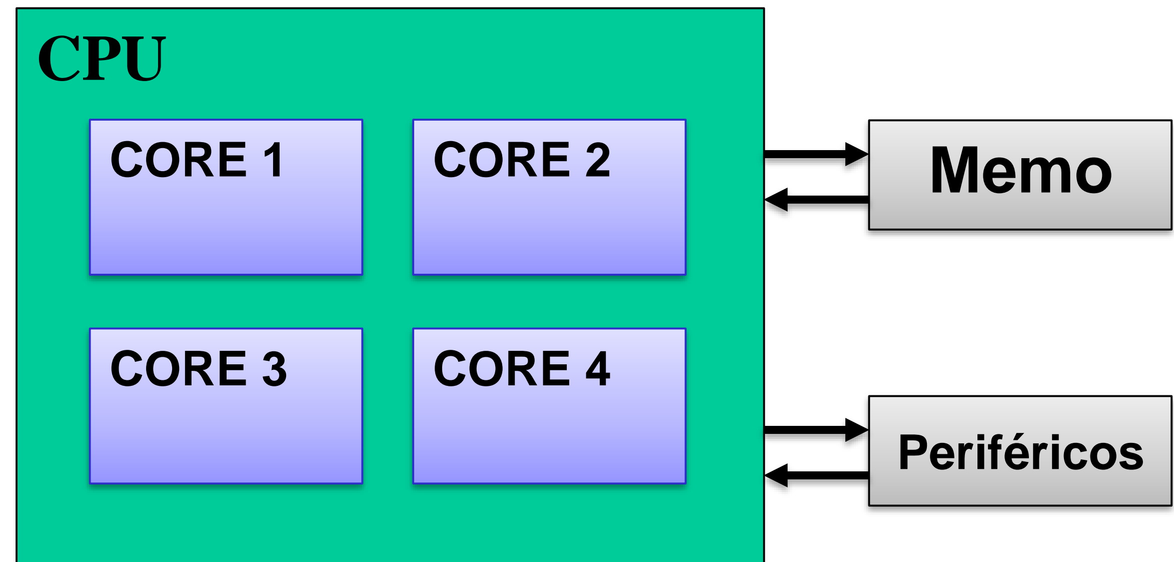
El computador: partes

- El corazón (core) de un computador está formado por:
  - Unidad de control (CU) encargada de la obtención, decodificación y ejecución de las instrucciones, y los accesos a la memoria principal
  - Unidad Aritmético-Lógica (ALU) encargada de las operaciones
  - Registros que contienen valores temporales (PC, SW, otros)
  - Un reloj que marca los tiempos de ejecución



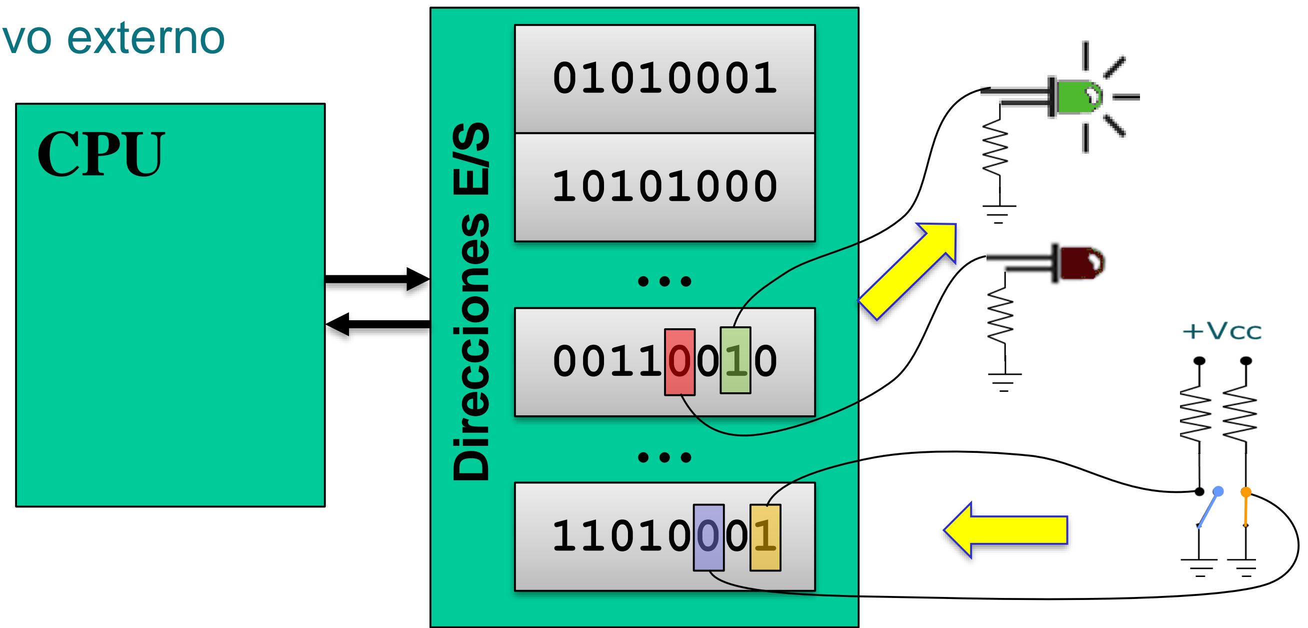
El computador: partes

- La Unidad Central de Proceso (CPU) está formada por uno o varios Cores, conectados a la misma memoria
  - Cada Core puede ejecutar secuencias de instrucciones de forma simultánea al resto, utilizando diferentes zonas de memoria
  - Además de a la memoria, la CPU se conecta a otros dispositivos externos llamados periféricos (o dispositivos de Entrada/Salida)



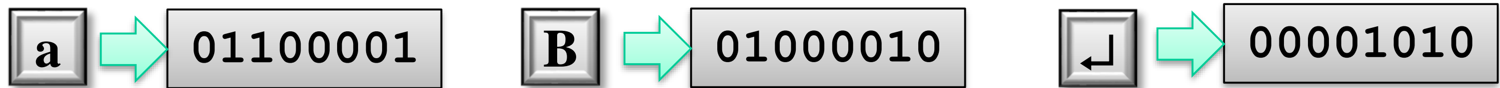
El computador: los periféricos

- Los periféricos (o dispositivos de Entrada/Salida) sirven para que la CPU intercambie información con el "exterior"
- De cara a la CPU, la E/S es similar a la memoria:
  - Está organizada en direcciones (mapa de E/S)
  - Se intercambia con ella datos binarios (instrucciones IN , OUT)
- La diferencia:
  - Cada bit de cada dirección de E/S está físicamente conectado a un dispositivo externo

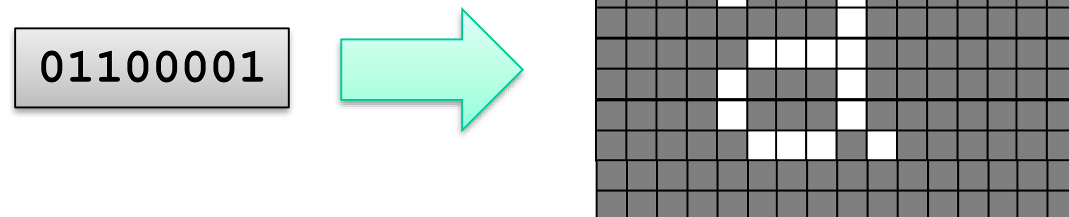


El computador: los periféricos

- La mayoría de periféricos realizan operaciones más sofisticadas con los valores binarios de las direcciones de E/S a las que están conectados:
  - **Teclado**: cuando se pulsa una tecla, escribe en la dirección de E/S correspondiente un código (ASCII) según la tecla pulsada



  - **Pantalla**: cuando la CPU escribe un código (ASCII) en la dirección de E/S correspondiente, genera puntos luminosos en los lugares adecuados para que el observador visualice el carácter equivalente



El computador

- ¿Cómo se cargan instrucciones y datos en memoria?
  - Se utiliza un programa “cargador” que vuelque los contenidos de un archivo (programa ejecutable) en memoria: instrucciones y datos
- ¿Cómo se ordena la ejecución del programa?
  - Se indica al programa “cargador” la dirección de la 1ª instrucción.
  - Una vez se ejecuta la 1ª instrucción, las siguientes se ejecutan secuencialmente (salvo los casos de salto).
- ¿Es necesario conocer y escribir programas en código binario?
  - No, existen programas más sofisticados (**compiladores** e intérpretes) que nos permiten traducir a código máquina desde **lenguajes de “alto nivel”** (similares al lenguaje matemático).

El computador

- Más información:
  - Presentación más detallada (con animaciones) del funcionamiento interno del computador:

  http://isa.uniovi.es/~ialvarez/Curso/descargas/Funcionamiento%20Computador.pps

- A continuación:
  - Programación en lenguaje de alto nivel

El lenguaje de alto nivel (C)

- Un lenguaje de alto nivel permite escribir programas en una notación similar a la utilizada habitualmente en el cálculo matemático.
  - Ej. media de 4 valores:

Leng máquina

```
MOV   m,x1
ADD   m,x2
ADD   m,x3
ADD   m,x4
DIV   m,4
```

Lenguaje C

```
int x1,x2,x3,x4;
float m;
m=(x1+x2+x3+x4)/4;
```

- La tarea de traducir el texto (código fuente) a la secuencia de instrucciones de máquina la realiza un programa compilador.
- El código de alto nivel es más fácil de escribir, entender, y portar (compiladores distintos generan, a partir del mismo código de alto nivel, instrucciones binarias para máquinas distintas).

El lenguaje de alto nivel (C)

- Para escribir programas en lenguaje de alto nivel, el programador debe:
  - Conocer y utilizar de forma precisa la notación que es capaz de entender el compilador.
  - Utilizar los tipos de datos adecuados para las variables y constantes. Los más básicos son: **int**, **float**, **char**.
  - Saber describir un algoritmo complejo en una secuencia de instrucciones entendibles por el compilador.
  - Utilizar funciones para E/S de datos.
  - Ejemplo: calcular media de 4 valores.
    - ¿De qué naturaleza son los datos a manejar?
      - Enteros: ej., los 4 valores representan el nº de alumnos de 4 clases.
      - Reales: ej., los 4 valores representan las temperaturas de 4 zonas del aula.
    - ¿Qué secuencia de operaciones permite obtener la media?
      - La media se obtiene sumando todos los valores y dividiendo por el nº de ellos.

Enteros y reales utilizan distintas formas de codificación y automatismos para el cálculo : necesitan instrucciones de máquina diferentes

El lenguaje de alto nivel (C)

- El compilador de C es un programa con reglas muy estrictas. Las más importantes:
  - Se deben declarar las variables antes de utilizarlas, indicando su tipo mediante una palabra clave (int, float, char) y utilizando un identificador (formado por caracteres alfanuméricos y _) .
  - Se distingue mayúsc de minúsc en palabras claves e identificadores
  - Todo el código ejecutable (instrucciones) debe estar dentro de una función (la principal es **main**, pero hay otras…)
  - Todas las sentencias deben terminar en punto y coma.
  - Las sentencias se ejecutarán en el orden en que están escritas.
  - Todo texto entre /* y */ será ignorado. También el resto de una línea tras //
  - Se puede operar con variables, constantes y resultados de funciones. Hay funciones estándar para las operaciones más habituales
  - Se invoca a una función con su nombre y, a continuación, entre paréntesis, los valores de sus argumentos.

Entrada/salida por consola en C

- Salida por consola (pantalla):
  - printf(“texto a escribir”,valor1,valor2,…);
    - Escribe en pantalla los caracteres de la cadena de texto, 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
    - \n en el texto significa retorno de carro.
  - Ejemplo:

```
int alumnos;
float nota_media;
alumnos=12;
nota_media=7.25;
printf("Hay %d alumnos, nota media %f\n",alumnos,nota_media);
```

Resultado en pantalla:

```
Hay 12 alumnos, nota media 7.25
_
```

Entrada/salida por consola en C

- Entrada de consola (teclado):
  - scanf(“texto”,direccion_valor1,direccion_valor2,…);
    - Espera por teclado la pulsación de una secuencia de caracteres terminada en INTRO.
    - El texto **sólo** debe contener cadenas de formato (%ALGO).
    - Las direcciones se indican anteponiendo a cada variable el carácter **&**
    - El usuario debe separar cada entrada con espacio, tabulador o Return, y siempre terminar con Return.
  - Ejemplo:

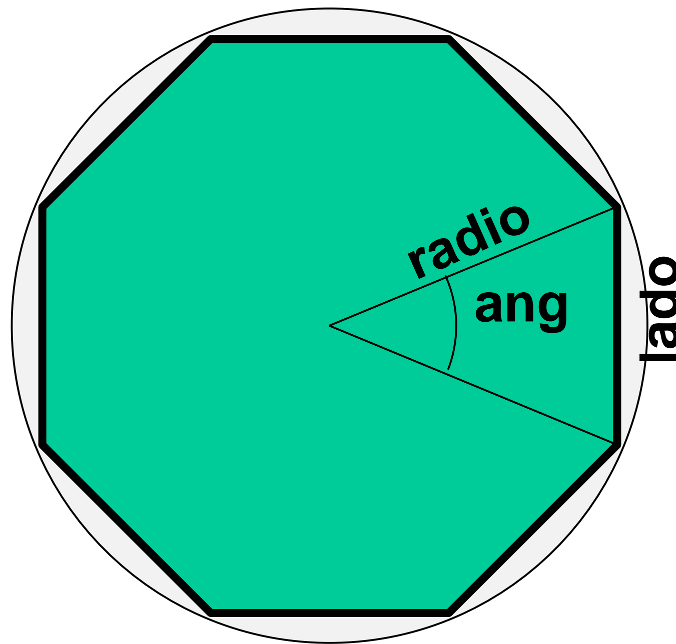
```
int alumnos;
float nota_media;
printf("Introduzca alumnos y nota media: ");
scanf("%d%f",&alumnos,&nota_media);
printf("Hay %d alumnos, nota media %f\n",alumnos,nota_media);
```

Resultado en pantalla:

```
Introduzca alumnos y nota media: 12   7.25↵
Hay 12 alumnos, nota media 7.25
_
```

El lenguaje de alto nivel (C)

- Ejemplo (ejercicio propuesto 3): dados el radio y el número de lados de un polígono regular, calcular su perímetro.



Datos:

n_lados (debe ser entero)

radio (puede ser real)

Resultado:

perim (puede ser real)

Cálculo:

$perim = lado * n_lados$

donde *lado* se obtiene de:

$$\sin(ang/2) = \frac{lado/2}{radio}$$

y *ang* se obtiene de:

$$ang = 2\pi/n_lados$$

En *rojo/cursiva*: variables necesarias
(cambian su valor en función de los datos iniciales y/o de la evolución de los cálculos)

El lenguaje de alto nivel (C)

- El programador debe utilizar de forma precisa la notación que es capaz de entender el compilador:
  - Declaraciones previas:
    - #include, #define, etc.
  - Declaración de programa principal:
    - main() { …. }
  - Dentro de main() { …}:
    - Declaración de las variables con sus tipos.
    - Código para obtener datos de partida del exterior (teclado, archivo, sensor, …).
    - Código para procesar los datos, **en orden**.
    - Código para enviar datos resultado al exterior (pantalla, archivo, accionador, …)
  - Comentarios (el compilador no los tiene en cuenta):
    - Resto de línea: // texto informativo
    - Multi-línea: /* texto informativo */

cada declaración o expresión se termina en ;

```
#include <stdio.h>
#include <stdlib.h>
#define PI 3.1416 // Constante

main()
{
   int n_lados; // Dato de usuario
   float radio; // Dato de usuario
   float perim;  /* Resultado */
   float ang,lado; // Valores aux

   printf("Introduzca n lados: ");
   scanf("%d",&n_lados);
   printf("Introduzca radio: ");
   scanf("%f",&radio);

   ang=2*PI/n_lados;
   lado=2*radio*sin(ang/2);
   perim=n_lados*lado;

   printf("El perim es %f\n",perim);
}
```

El lenguaje de alto nivel (C)

- Ejecución, paso por paso:

Código fuente

```
#include <stdio.h>
#include <stdlib.h>
#define PI 3.1416 // Constante

main()
{
    int n_lados; // Dato de usuario
    float radio; // Dato de usuario
    float perim;  /* Resultado */
    float ang,lado; // Valores aux

    printf("Introduzca n lados: ");
    scanf("%d",&n_lados);
    printf("Introduzca radio: ");
    scanf("%f",&radio);

    ang=2*PI/n_lados;
    lado=2*radio*sin(ang/2);
    perim=n_lados*lado;

    printf("El perim es %f\n",perim);
}
```

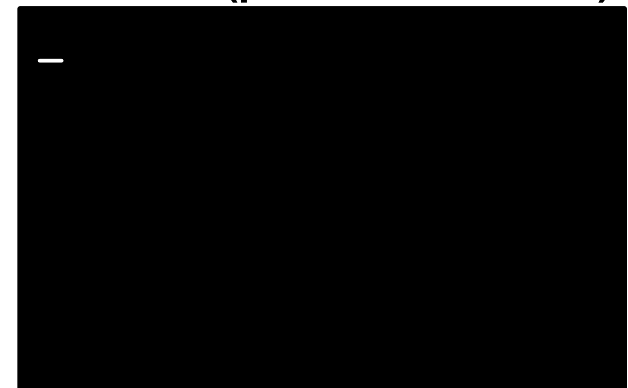
Acción

Reserva lugares en memoria para las variables declaradas y “recuerda” la posición y tipo de cada una

Memoria

Dirección	Valor	Variable
8000 a 8003	?	n_lados
8004 a 8007	?	radio
8008 a 8011	?	perim
8012 a 8015	?	ang
8016 a 8019	?	lado

Consola (pantalla + teclado)



El lenguaje de alto nivel (C)

- Ejecución, paso por paso:

Código fuente

```
#include <stdio.h>
#include <stdlib.h>
#define PI 3.1416 // Constante

main()
{
    int n_lados; // Dato de usuario
    float radio; // Dato de usuario
    float perim;  /* Resultado */
    float ang,lado; // Valores aux

    printf("Introduzca n lados: ");
    scanf("%d",&n_lados);
    printf("Introduzca radio: ");
    scanf("%f",&radio);

    ang=2*PI/n_lados;
    lado=2*radio*sin(ang/2);
    perim=n_lados*lado;

    printf("El perim es %f\n",perim);
}
```

Acción

Invoca a función printf() que escribe el texto indicado en pantalla

Memoria

Dirección	Valor	Variable
8000 a 8003	?	n_lados
8004 a 8007	?	radio
8008 a 8011	?	perim
8012 a 8015	?	ang
8016 a 8019	?	lado

Consola (pantalla + teclado)

```
Introduzca n lados:_
```

El lenguaje de alto nivel (C)

- Ejecución, paso por paso:

Código fuente

```
#include <stdio.h>
#include <stdlib.h>
#define PI 3.1416 // Constante

main()
{
    int n_lados; // Dato de usuario
    float radio; // Dato de usuario
    float perim;  /* Resultado */
    float ang,lado; // Valores aux

    printf("Introduzca n lados: ");
    scanf("%d",&n_lados);
    printf("Introduzca radio: ");
    scanf("%f",&radio);

    ang=2*PI/n_lados;
    lado=2*radio*sin(ang/2);
    perim=n_lados*lado;

    printf("El perim es %f\n",perim);
}
```

Acción

Invoca a función scanf() que espera pulsaciones por teclado terminadas en INTRO, y asigna valor a n_lados (%d indica que es un entero) (&n_lados = 8000)

Memoria

Dirección	Valor	Variable
8000 a 8003	8	n_lados
8004 a 8007	?	radio
8008 a 8011	?	perim
8012 a 8015	?	ang
8016 a 8019	?	lado

Consola (pantalla + teclado)

```
Introduzca n lados: 8 ↵
_
```

El lenguaje de alto nivel (C)

- Ejecución, paso por paso:

Código fuente

```
#include <stdio.h>
#include <stdlib.h>
#define PI 3.1416 // Constante

main()
{
    int n_lados; // Dato de usuario
    float radio; // Dato de usuario
    float perim;  /* Resultado */
    float ang,lado; // Valores aux

    printf("Introduzca n lados: ");
    scanf("%d",&n_lados);
    printf("Introduzca radio: ");
    scanf("%f",&radio);

    ang=2*PI/n_lados;
    lado=2*radio*sin(ang/2);
    perim=n_lados*lado;

    printf("El perim es %f\n",perim);
}
```

Acción

Id. para radio (%f para un float) (&radio = 8004)

Memoria

Dirección	Valor	Variable
8000 a 8003	8	n_lados
8004 a 8007	12.5	radio
8008 a 8011		perim
8012 a 8015	?	ang
8016 a 8019	?	lado

Consola (pantalla + teclado)

```
Introduzca n lados: 8 ↲
Introduzca radio: 12.5 ↲
_
```

El lenguaje de alto nivel (C)

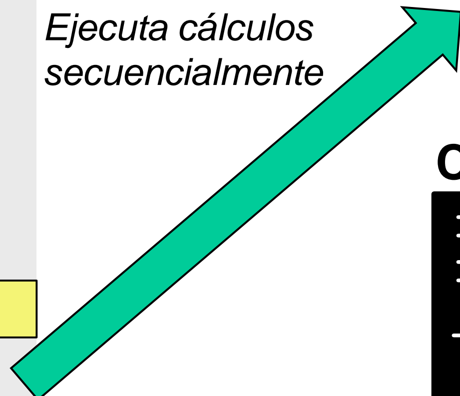
- Ejecución, paso por paso:

Código fuente

```
#include <stdio.h>
#include <stdlib.h>
#define PI 3.1416 // Constante

main()
{
    int n_lados; // Dato de usuario
    float radio; // Dato usuario
    float perim;  /* Resultado */
    float ang,lado; // Valores aux

    printf("Introduzca n lados: ");
    scanf("%d",&n_lados);
    printf("Introduzca radio: ");
    scanf("%f",&radio);

    ang=2*PI/n_lados;
    lado=2*radio*sin(ang/2);
    perim=n_lados*lado;

    printf("El perim es %f\n",perim);
}
```

Acción

Ejecuta cálculos secuencialmente



Memoria

8000 a 8003	8	n_lados
8004 a 8007	12.5	radio
8008 a 8011	?	perim
8012 a 8015	0.78540	ang
8016 a 8019	?	lado

Consola (pantalla + teclado)

```
Introduzca n lados: 8 ↵
Introduzca radio: 12.5 ↵
_
```

El lenguaje de alto nivel (C)

- Ejecución, paso por paso:

Código fuente

```
#include <stdio.h>
#include <stdlib.h>
#define PI 3.1416 // Constante

main()
{
    int n_lados; // Dato de usuario
    float radio; // Dato de usuario
    float perim;  /* Resultado */
    float ang,lado; // Valores aux

    printf("Introduzca n lados: ");
    scanf("%d",&n_lados);
    printf("Introduzca radio: ");
    scanf("%f",&radio);

    ang=2*PI/n_lados;
    lado=2*radio*sin(ang/2);
    perim=n_lados*lado;

    printf("El perim es %f\n",perim);
}
```

Acción

Ejecuta cálculos secuencialmente

Memoria

Dirección	Valor	Variable
8000 a 8003	8	n_lados
8004 a 8007	12.5	radio
8008 a 8011	?	perim
8012 a 8015	0.78540	ang
8016 a 8019	9.5918	lado

Consola (pantalla + teclado)

```
Introduzca n lados: 8 ↲
Introduzca radio: 12.5 ↲
_
```

Universidad de Oviedo

INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

El lenguaje de alto nivel (C)

- Ejecución, paso por paso:

Código fuente

```
#include <stdio.h>
#include <stdlib.h>
#define PI 3.1416 // Constante

main()
{
    int n_lados; // Dato de usuario
    float radio; // Dato de usuario
    float perim;  /* Resultado */
    float ang,lado; // Valores aux

    printf("Introduzca n lados: ");
    scanf("%d",&n_lados);
    printf("Introduzca radio: ");
    scanf("%f",&radio);

    ang=2*PI/n_lados;
    lado=2*radio*sin(ang/2);
    perim=n_lados*lado;

    printf("El perim es %f\n",perim);
}
```

Acción

Ejecuta cálculos secuencialmente

Memoria

8000 a 8003	8	n_lados
8004 a 8007	12.5	radio
8008 a 8011	57.551	perim
8012 a 8015	0.78540	ang
8016 a 8019	9.5918	lado

Consola (pantalla + teclado)

```
Introduzca n lados: 8 ↵
Introduzca radio: 12.5 ↵
_
```

El lenguaje de alto nivel (C)

- Ejecución, paso por paso:

Código fuente

```
#include <stdio.h>
#include <stdlib.h>
#define PI 3.1416 // Constante

main()
{
    int n_lados; // Dato de usuario
    float radio; // Dato de usuario
    float perim;  /* Resultado */
    float ang,lado; // Valores aux

    printf("Introduzca n lados: ");
    scanf("%d",&n_lados);
    printf("Introduzca radio: ");
    scanf("%f",&radio);

    ang=2*PI/n_lados;
    lado=2*radio*sin(ang/2);
    perim=n_lados*lado;

    printf("El perim es %f\n",perim);
}
```

Acción

Invoca a función printf() que escribe el texto indicado en pantalla (sustituye %f por el valor de perim que es un float)

Memoria

Dirección	Valor	Variable
8000 a 8003	8	n_lados
8004 a 8007	12.5	radio
8008 a 8011	57.551	perim
8012 a 8015	0.785[illegible]	ang
8016 a 8019	9.59[illegible]	lado

Consola (pantalla + teclado)

```
Introduzca n lados: 8 ↲
Introduzca radio: 12.5 ↲
El perim es 57.551

_
```

Ejercicios propuestos

1) Realizar un programa que pida los lados de un rectángulo, y calcule y escriba su diagonal.
2) Realizar un programa que pida los lados de un paralelepípedo rectangular, y calcule y escriba su diagonal.
3) Pedir el radio y el nº de lados de un polígono regular, y calcular su perímetro (comprobar que el perímetro se acerca a 2π al incrementar el nº de lados).
4) Pedir dos puntos en el espacio bidimensional, y calcular el ángulo con el eje *x* de la recta que los une.
5) Pedir dos puntos de una recta en el espacio 2D y escribir el valor de la ordenada *y* para cualquier valor de la abscisa *x*.
6) Realizar un programa que calcule el día de la semana de cualquier fecha (http://es.wikipedia.org/wiki/Congruencia_de_Zeller).

NOTAS: Algunas operaciones matemáticas en C (incluir <math.h>)

x^2 = `x*x` $\sqrt{x}$ = `sqrt(x)`

Operaciones trigonométricas: `sin(x)`, `cos(x)`, `tan(x)`, `atan(x)`

División entera. Cociente = `a/b` ; Resto = `a%b`

Indice

- Introducción al computador y el lenguaje C
- **Datos, expresiones y algoritmos**
- Funciones
- Tablas y punteros
- Cadenas de caracteres
- Operaciones con valores binarios
- Preprocesador y compilación separada
- E/S en archivos y dispositivos
- Criterios de buena programación

Tipos de datos

- Los datos pueden ser de varios tipos. Los más básicos:
  - **int**: para almacenar y realizar operaciones con valores enteros.
  - **float**: para almacenar y realizar operaciones con valores reales.
  - **char**: para almacenar y realizar operaciones con texto.
- El código programa puede utilizar datos de dos procedencias diferentes:
  - **Variables**: valores que pueden cambiar a lo largo del programa.
    - Ejs:

```
int n_alumnos;              // Variable tipo entero, sin inicializar
float nota_media=7.25;      // Variable tipo real, valor inicial 7.25
```

  - **Constantes**: valores que no cambian a lo largo del programa.
    - Ejs: **7** (días de la semana), **3.1416** (π), **"Introduzca x1:"** (texto), etc.
    - Preferible declararlas con #define al principio del código:

```
#define PI                  3.1416
```

y usar después mediante el nombre:

```
z = 2 * cos(PI/4);
```

Tipos de datos

- ¿Y si hay datos que no son de los tipos básicos?
  - Ejemplos: meses del año, frutas, colores, num. complejos, etc.
- Hay que buscar la mejor conversión de nuestros tipos de datos a los manejables por el lenguaje:
  - **Meses del año**: se puede utilizar un **int** que valga 1 para Enero, 2 para Febrero, …, 12 para Diciembre.
  - **Frutas**: un **int** para cada tipo de fruta: 1 para Manzana, 2 para Pera, ….
  - En los casos anteriores, es convienente asociar cada valor a una constante con #define:

```
#define ENERO       1
#define FEBRERO     2
…
int mes;
mes=FEBRERO;
```

  - **Colores**: un color es la combinación de 3 valores (RGB); se pueden utilizar 3 **float** (uno para cada componente) con valores de 0 a 1.
  - **Complejos**: utilizar 2 **float** (parte real, parte imaginaria).

Universidad de Oviedo

INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

Sentencias y expresiones

- Sentencia básica: la asignación.

  variable = expresión **;**

- Expresión: resultado de una operación matemática, utilizando los operadores típicos.
  - Aritméticos: **+ - * /**
  - Paréntesis: **(…)**
  - etc.
- Los operandos pueden ser:
  - Variables
  - Constantes
  - Resultados de funciones (sin,cos,tan,atan,exp,sqrt,pow,log,log10,…)
- Ejemplo de sentencia:

```
z = 3*y + (2*cos(PI/4));  // Incluir <math.h> para usar cos()
```

Expresiones

- A tener en cuenta al realizar expresiones:
  - El compilador dividirá la expresión en una secuencia de operaciones, por orden de prioridad (similar a nuestro lenguaje matemático):

```
z = 3*y + (2*cos(PI/4)); // Incluir <math.h> para usar cos()
```

  - Los cálculos internos (secuencia de instrucciones de máquina generada por el compilador) se ejecutarán por este orden:
    - 3.1416 / 4 → r1
    - cos(r1) → r2
    - 2*r2 → r3
    - 3*y → r4
    - r4+r3 → r5
    - z=r5 → r6

Los valores r1, r2, … se almacenan temporalmente en los "registros" de la CPU

(z pasa a valer r5)

Expresiones

- A tener en cuenta al realizar expresiones:
  - ¡¡ ATENCION !! Cada operación parcial utiliza los tipos de datos que tiene como operandos, y realiza la operación y devuelve el resultado según esos tipos:
    - op1 *(int)* OP op2 *(int)* → r *(int)*
    - op1 *(float)* OP op2 *(float)* → r *(float)*
    - op1 *(int)* OP op2 *(float)* → r *(float)*
    - op1 *(float)* OP op2 *(int)* → r *(float)*
  - Ejemplo:

```
int x=4,y=8;
float z;
z=x/y;          // Resultado esperado: z=0.5
```

  - Ejecución:
    - x *(=4 int)* / y *(=8 int)* → r1 *(=0, int)*
    - z = r1 *(=0,int)* → r2 *(=0,float)* **(z pasa a valer 0)**

Universidad de Oviedo

INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

Expresiones

- A tener en cuenta al realizar expresiones:
  - No hay que cambiar el tipo de un dato por culpa de las expresiones en que va a estar involucrado. Se puede forzar un cambio de tipo de datos en una expresión mediante el operador molde (cast):
    - (tipo) op → r *(tipo)*
      r es lo más parecido posible a op en el nuevo tipo
  - Ejemplo:

```
int profes=4,alumnos=8;   // El nº de profesores y alumnos es entero
float ratio;
ratio=(float) profes/(float) alumnos;
```

  - Ejecución:
    - (float) profes *(=4 int)* → r1 *(=4, float)*
    - (float) alumnos *(=8 int)* → r2 *(=8, float)*
    - r1 *(=4 float)* / r2 *(=8 float)* → r3 *(=0.5, float)*
    - ratio = r3 *(=0.5,float)* → r4 *(=0.5,float)* **(ratio pasa a valer 0.5)**

Sentencias de control

- La ejecución de sentencias es secuencial, salvo si se usan sentencias de control:
  - Bloque:

```
{
   sentencia1;           // Las sentencias incluidas en un bloque
   sentencia2;           // son tratadas como una sola por otras
   …                     // sentencias de control
}
```

  - Alternativa:

```
if (condicion)
   sentenciaSiTRUE;      // Sólo se ejecuta si se cumple la condición
else                     // La cláusula else es opcional
   sentenciaSiFALSE;     // Sólo se ejecuta si no se cumple la condición
```

  - Bucle while:

```
while (condicion)
   sentenciaARepetir;    // Repite sentencia mientras se cumpla condición
```

  - Bucle for:

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

Sentencias de control

- Condición de una sentencia de control:
  - Es un valor **entero** que se interpreta como FALSO si vale 0, y como VERDADERO si es distinto de 0.
  - Suele ser el resultado de una expresión con operadores relacionales:
    - (op1 **==** op2) Verdadero (≠0) si ambos son iguales, Falso (0) si no.
    - (op1 **!=** op2) Verdadero si ambos son distintos
    - (op1 **>** op2) Verdadero si op1 es mayor estricto que op2
    - (op1 **>=** op2) …
    - (op1 **<** op2) …
    - (op1 **<=** op2) …
  - Ejemplo: y = | x |

```
if (x>=0)
    y = x;
else
    y = -x;
```

La sentencia que depende del if, else, while o for, se indenta una tabulación a la derecha por claridad.

Sentencias de control

- Condición de una sentencia de control:
  - Los operadores relacionales se pueden encadenar con operadores lógicos a nivel de palabra:
    - (r1 **&&** r2)   Verdadero si r1 **Y** r2 son verdaderos, falso en caso contrario
    - (r1 **||** r2)   Verdadero si r1 **O** r2 son verdaderos, falso en caso contrario
    - (**!** r1)   Verdadero si r1 es falso, falso si r1 es verdadero
  - Ejemplo: y = min (| x | , 4)

```
if ((x>4) || (x<-4))
   y = 4;
else
{
   if (x>=0)
      y = x;
   else
      y = -x;
}
```

Usar { … } cuando una if, else, for ó while debe afectar a un conjunto de sentencias.

Indentar correctamente el texto facilita su lectura.

Algoritmos

- Usando variables, constantes, expresiones y sentencias de control, el programador debe trasladar su pensamiento lógico (algoritmo) a una secuencia de instrucciones que lo desarrollen.
- Aunque las expresiones de alto nivel son más complejas, siguen estando lejos del pensamiento humano.
  - Ej: fact = n!
    - En nuestra forma de pensar: fact = n * (n-1)* (n-2) * … * 1
    - La expresión anterior no es realizable directamente en lenguaje C: hay que pensar un algoritmo que pueda ser desarrollado en una secuencia de instrucciones:

```
fact=1
Repetir desde i=1 hasta n
        fact=fact*i
Al llegar aquí: fact=n!
```

Universidad de Oviedo

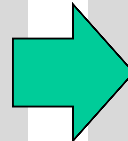
INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

Algoritmos

- Una vez pensado el algoritmo, hay que traducirlo al lenguaje C:
  - Detectar las variables necesarias y sus tipos.
  - Dar valores iniciales a las variables que lo requieran.
  - Convertir el algoritmo en expresiones y sentencias de control.
  - Usar los resultados en el modo deseado.
  - Comprobar el funcionamiento con casos conocidos.

Algoritmo:

```
fact=1
Repetir desde i=1 hasta n
     fact=fact*i
Al final, fact=n!
```



Lenguaje C:

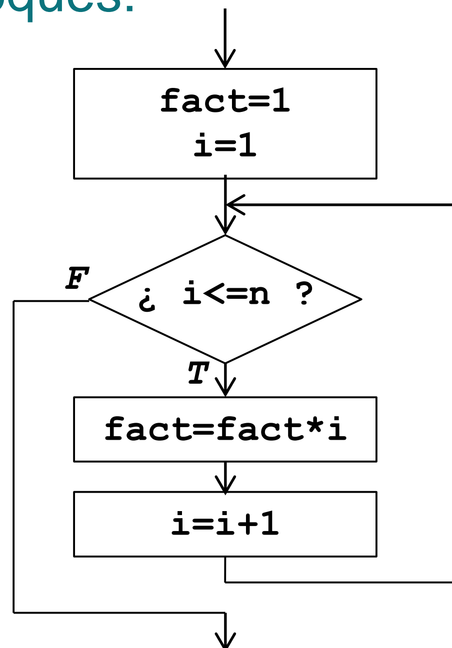
```
int n,i,fact;

printf("Introd n: ");
scanf("%d",&n);

fact=1;
for (i=1;i<=n;i++)
         fact=fact*i;
printf("%d! = %d\n",n,fact);
```

Algoritmos

- Dos formas básicas de “pensar” algoritmos:
  - Pseudo-código:
    - Similar al lenguaje de programación, pero sin sus restricciones de formato

```
fact=1
Repetir desde i=1 hasta n
    fact=fact*i
```

  - Diagrama de bloques:



Algoritmos

- Cuando “pensamos” algoritmos:
  - Desarrollo top-down: diseccionar el algoritmo total en grandes bloques, luego ir resolviendo cada bloque.
  - Al final, tenemos que llegar a instrucciones sencillas realizables mediante el lenguaje.
  - Ej: calcular e^x en desarrollo de serie de Taylor.

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

No se puede llegar hasta ∞ (no acabaría nunca)

```
// Algoritmo completo
exp_x=0
Repetir desde n=0 hasta nmax
    exp_x=exp_x + xⁿ / n!
```

```
// x_n=xⁿ
x_n=1
Repetir desde i=1 hasta n
    x_n=x_n*x;
```

```
// fact_n=n!
fact_n=1
Repetir desde i=1 hasta n
    fact_n=fact_n*i
```

Algoritmos

- Al traducir los algoritmos al lenguaje C:
  - Detectar las "nuevas" variables necesarias y sus tipos.
  - Intentar buscar "optimizaciones".
  - Ej: calcular e^x en desarrollo de serie de Taylor.

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

```
// Algoritmo completo
exp_x=1
x_n=1
fact_n=1
Repetir desde n=1 hasta nmax
   x_n = x_n * x
   fact_n = fact_n * n
   exp_x=exp_x + x_n / fact_n
```

La iteración n=0 da $x^0/0! = 1$

Aprovechamos que:
$x^n=x*x^{n-1}$
$n!=n*(n-1)!$

  - ¡OJO! Las optimizaciones hacen que nuestro código se ejecute más rápido (menos operaciones), pero dificultan su legibilidad.

Algoritmos

- Una vez escrito el código, ¡¡ hay que probarlo !!
- Un programa sintácticamente correcto (compila sin errores) no es equivalente a un programa correcto:
  - Pueden producirse numerosos fallos en el proceso:
    - Algoritmo mal pensado: no hace lo que se espera.
    - Mala traducción: no se convierte correctamente al lenguaje.
    - Mal uso de variables: modificaciones inesperadas o en lugares inapropiados, errores en tipos de datos, etc.
    - Fallos de "cálculo": no se estima correctamente lo que el lenguaje va a hacer.
    - Errores ocasionales: no se han tenido en cuenta todas las opciones que pueden suceder.
  - La prueba de los programas y corrección de errores en su código se llama depuración.
    - La depuración es un **paso clave** en el desarrollo de programas.

Más sobre tipos de datos

- **char,float, int** son los tipos de datos más utilizados de los disponibles en C.
- En ocasiones hay que seleccionar otros tipos para ajustarse al rango/resolución de los valores a manejar:
  - Tipos enteros, según el nº de bits utilizados:
    - char, short, int, long (de menor a mayor nº de bits).
    - A mayor nº de bits, mayor rango (2^n valores diferentes).
    - Todos ellos admiten unsigned (sólo positivos) o signed (por defecto).

	Nº bits mín	Nº bits VC++	Rango unsigned		Rango signed	
			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

Más sobre tipos de datos

- **char,float, int** son los tipos de datos más utilizados de los disponibles en C.
- En ocasiones hay que seleccionar otros tipos para ajustarse al rango/resolución de los valores a manejar:
  - Tipos reales, según el nº de bits utilizados:
    - float, double, long double (de menor a mayor nº de bits).
    - A mayor nº de bits, mayor rango y resolución (precisión).
    - Se codifican en notación exponencial: 1.mant x 2^{exp}
      https://www.h-schmidt.net/FloatConverter/IEEE754.html

	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	$3.4x10^{38}$	>=6 dígitos decimales
double	>=float	64	11 bits exceso a 1023	52 bits 1.Mant	$1.8x10^{308}$	>=15 dígitos decimales
long double	>float >=double	id. a double				

Más sobre tipos de datos

- El programador puede definir nuevos tipos de datos:
  - Estructuras
  - Uniones
  - Enumeraciones
- Las estructuras permiten agrupar varias variables bajo un solo tipo, de forma que se pueden usar como conjunto o aisladamente cuando convenga:

- Declaración:

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

- Uso:

```
struct nombre vble;

vble.campo1A=…;
if (vble.campo1B==…)
   …;
```

Más sobre tipos de datos

- El programador puede definir nuevos tipos de datos:
  - Estructuras
  - Uniones
  - Enumeraciones
- Las estructuras permiten agrupar varias variables bajo un solo tipo, de forma que se pueden usar como conjunto o aisladamente cuando convenga:

- Declaración:

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

- Uso:

```
struct nombre vble;

vble.campo1A=…;
if (vble.campo1B==…)
   …;
```

Más sobre tipos de datos

- Ejemplos de uso de estructuras:
  - Manejo de colores: estructura con tres reales para las componentes r,g,b:

```
struct color
{
    float red,green,blue;
} ;
struct color amarillo;
amarillo.red=1; amarillo.green=1 ; amarillo.blue=0;
```

  - Estructura para dibujo en 2D de un círculo:

```
struct circulo
{
    float centroX,centroY,radio;
    struct color colorBorde,colorFondo;
    int anchoBorde,dibujarFondo;
} ;

struct circulo sol;
...
sol.colorFondo=amarillo;
sol.dibujarFondo=1;
...
```

Ejercicios propuestos

1) Obtener las raíces (reales o complejas conjugadas) de un polinomio de grado 2: $p(x)=a.x^2+b.x+c$.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

2) Realizar un programa que calcule el nº de equipos titulares diferentes que puede realizar un entrenador, según el nº de jugadores convocados y el nº de jugadores que se alinean.

$$C_{n,m} = \binom{n}{m} = \frac{n!}{m!\,.\,(n-m)!}$$

3) Realizar un programa pida un nº entre 0 y 255, y escriba el valor binario equivalente (8 bits).

4) Comprobar que las dos aproximaciones siguientes se acercan al número e cuando n →∞.

$$e = \sum_{n=0}^{\infty} \frac{1}{n!} \qquad \lim_{n\to\infty} \left(1 + \frac{1}{n}\right)^n$$

5) Calcular el mínimo de un polinomio de grado 2 entre dos límites, utilizando la aproximación Golden-Search (http://pages.intnet.mu/cueboy/education/notes/algebra/goldensectionsearch.pdf).

NOTAS: Algunas operaciones matemáticas en C (incluir <math.h>)

x^n = `x*x*…*x` $\sqrt{x}$ = `sqrt(x)`   (x debe ser ≥ 0)

Indice

- Introducción al computador y el lenguaje C
- Datos, expresiones y algoritmos
- **Funciones**
- Tablas y punteros
- Cadenas de caracteres
- Operaciones con valores binarios
- Preprocesador y compilación separada
- E/S en archivos y dispositivos
- Criterios de buena programación

Funciones

- Al desarrollar nuestros algoritmos, detectamos porciones de código repetitivas.
- En estas porciones de código:
  - Sólo cambian las variables afectadas y/o sus valores.
  - Se requieren nuevas variables para un uso temporal.
- Ej: calcular $\binom{m}{n} = \frac{m!}{(m-n)!}$

```
// Algoritmo
comb_m_n = m! / (m-n)!
```

```
// fact_m=m!
fact_m=1
Repetir desde i=1 hasta m
        fact_m=fact_m*i
```

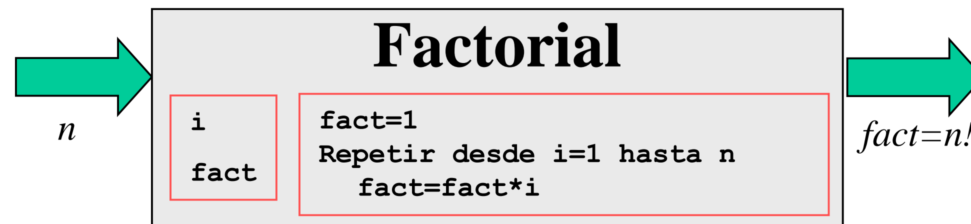
```
// fact_m_n=(m-n)!
fact_m_n=1
Repetir desde i=1 hasta m-n
        fact_m_n=fact_m_n*i
```

  - El cálculo de m! y (m-n)! es idéntico, pero los valores de partida son distintos.
  - El resultado de los dos factoriales se almacenará en variables distintas.
  - Para el cálculo de factorial se requiere temporalmente una variable índice, que después no se utiliza más.

Funciones

- Las porciones de código repetitivas (o que pudieran serlo) se desarrollan en forma de funciones
- Una función, para el resto del código, trabaja como una caja negra:

  - Las entradas (parámetros) de la función son los valores que modifican su comportamiento.
  - La salida de la función es el valor calculado a partir de los parámetros.
  - La función necesita para su desarrollo variables temporales (locales) y un código a ejecutar



Funciones en C

Declaración de una función

```
tipoDvto NombreFuncion(tipo1 param1,tipo2 param2,...)
{
    tipoDvto resultado;
    tipoA varlocalA;
    tipoB varlocalB1,varlocalB2;

    sentencias a ejecutar terminadas en ;
    (sólo pueden usar los parámetros y
    las variables locales)

    return resultado;
}
```

Parámetros (tipo1 param1,tipo2 param2,...)

Cabecera (primera línea)

Declaración variables locales

Código de la función

```
int Factorial(int n)
// Calcula y devuelve n!
{
    int fact;
    int i;

    fact=1;
    for (i=1;i<=n;i++)
        fact=fact*i;

    return fact;
}
```

- El código de la función sólo depende de sí misma: dar nombres generales a su identificador, parámetros y variables locales.
- Una función debe ser declarada antes de ser utilizada en el resto del código.

Invocación de funciones

- Una vez declarada una función, puede ser llamada (o invocada) desde cualquier expresión en otra zona del código (función llamante), pasando como parámetros los valores deseados y utilizando su resultado.
- Cuando se invoca a una función, se deja de ejecutar temporalmente el código de la llamante, y se vuelve al mismo al finalizar.
- No hay que indicar tipos en la llamada, pero éstos deben ser compatibles con los declarados.

```
int Factorial(int n)
// Calcula y devuelve n!
{
    ...
}

...
// En otra zona del programa (función llamante)
    int m, n,comb;
    ... Damos valores a m,n ...
    comb=Factorial(m)/Factorial(m-n);
    ... Usamos comb ...
```

Aunque estas variables se llamen igual, son distintas (están en zonas de memoria diferentes)

Universidad de Oviedo

INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

Funcionamiento interno

- Cuando se invoca a una función, se deja de ejecutar temporalmente el código de la llamante, y se vuelve al mismo al finalizar.
- Las variables locales de la función llamante mantienen sus valores, pero no son utilizables en la invocada.
- Las variables locales y parámetros de la función invocada tienen una **vida temporal**: sólo existen mientras se está ejecutando.
- Los parámetros toman su valor inicial en la invocación, pero son variables nuevas (y por tanto se pueden modificar sin afectar a las variables de la llamante).
- Al regresar a la llamante, sólo se puede utilizar el valor devuelto.

- El detalle del funcionamiento (pila LIFO) se puede observar en:
  http://isa.uniovi.es/~ialvarez/Curso/descargas/Funcionamiento%20Computador.pps

Uso de funciones

- ¡¡¡ Es importante usar funciones !!! :
  - Se reduce la cantidad de código a escribir.
  - Se pueden **realizar y probar por separado**, antes de incluirlas en el programa definitivo.
  - Una función puede llamar a otra(s), permitiendo desarrollar con facilidad algoritmos top-down más complejos.
  - El código es reutilizable para otros programas, ya que la función no depende del resto del programa.
- ¿Cuándo introducir una función?
  - Siempre que se detecte una porción de código que realiza una tarea parcial “auto-contenida”:
    - Sólo depende de valores iniciales para producir un resultado.
    - Es independiente del resto del programa (n! no depende de para qué va a ser usado).
    - Requiere valores temporales que no necesita el resto del programa.

Funciones y estructuras

- Las funciones pueden recibir y devolver valores tipo estructura.
  - Al igual que con el resto de tipos, se realiza una copia local para los parámetros, y se copia en la variable destino el resultado (en estructuras grandes, estas copias pueden ser muy lentas).
  - Ejemplo:

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

struct complejo ProdComplejos(struct complejo a, struct complejo b)
// Calcula y devuelve a*b
{
    struct complejo prod;
    prod.re=a.re*b.re-a.im*b.im;
    prod.im=a.re*b.im+a.im*b.re;
    return prod;
}

...
// En otra zona del programa (función llamante)
    struct complejo uno_menos_j,uno_menos_j_cuadrado;
    uno_menos_j.re=1;   uno_menos_j.im= -1;
    uno_menos_j_cuadrado=ProdComplejos(uno_menos_j,uno_menos_j);
```

Ejercicios propuestos

- Realizar utilizando funciones (cada ejercicio utilizará funciones realizadas en los anteriores):
  1) Calcular el máximo común divisor de 2 enteros mediante el algoritmo de Euclides (http://latecladeescape.com/t/Algoritmo+de+Euclides).
  2) Calcular el mínimo común múltiplo de 2 enteros: mcm(a,b)=a.b/mcd(a,b)
  3) Se divide una misma longitud utilizando 3 escalas:
     - La escala A cada 12 m
     - La escala B cada 18 m
     - La escala C cada 30 m

     ¿ En qué punto después del origen coincidirán por 1ª vez las marcas de las 3 escalas?

     mcm(a,b,c)=mcm(mcm(a,b),c)
  4) ¿Cuál es el área del círculo cuyo centro es la primera coincidencia de las escalas A y B, y que pasa por la primera coincidencia de las 3 escalas?

Universidad de Oviedo

INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

Indice

- Introducción al computador y el lenguaje C
- Datos, expresiones y algoritmos
- Funciones
- **Tablas y punteros**
- Cadenas de caracteres
- Operaciones con valores binarios
- Preprocesador y compilación separada
- E/S en archivos y dispositivos
- Criterios de buena programación

Tablas (arrays)

- En muchas ocasiones, necesitamos usar conjuntos de variables del mismo tipo.

```
int x1,x2,x3,x4;    // Creamos 4 variables con nombres distintos
int media;

media=(x1+x2+x3+x4)/4;
```

- El desarrollo de algoritmos se simplifica si las guardamos como una tabla (o array):

```
int x[4]; // Creamos 4 variables bajo el mismo nombre
          // accesibles como x[0], x[1], x[2], x[3]
int media,suma,i;

suma=0;
for (i=0;i<4;i++)
    suma=suma+x[i];
media=suma/4;
```

Tablas (arrays)

`int x1,x2,x3,x4;`
`int m;`
 	
`m=(x1+x2+x3+x4)/4;`
	`int x[4];`
	`int m,suma,i;`
	
	`suma=0;`
	`for (i=0;i<4;i++)`
	`    suma=suma+x[i];`
	`m=suma/4;`

- ¿Realmente se ha simplificado el algoritmo? La respuesta es sencilla, sólo hay que considerar otras situaciones:
  - Si se necesitan 100 ó 1000 variables en lugar de 4:
    - El código con variables sueltas es mucho más largo y tedioso de escribir.
    - El código con arrays se mantiene igual (sólo cambiar 4 por 100 ó 1000).
  - Si el número de variables a utilizar (n) depende de resultados previos del programa:
    - Con variables sueltas es prácticamente inabordable (hay que escribir código específico para cada valor de n).
    - Con arrays, sólo hay que cambiar 4 por n **(*)**

(*) excepto en la declaración de la variable, donde el tamaño debe ser un valor constante

Tablas (arrays)

- Es importante entender cómo son manejados los arrays por el lenguaje.
- En C, cuando se declara un array de N posiciones:

```
tipo tabla[N];
```

  - N debe ser un valor constante.
  - Se reservan en memoria N posiciones consecutivas para datos del tipo indicado.
  - Los N elementos se acceden mediante `tabla[i]`, donde `i` es un valor entero **entre 0 y N-1 .** Cada elemento es a todos los efectos una variable del tipo declarado.
  - Una vez declarado el array, el lenguaje C no recuerda el tamaño N, es responsabilidad del programador.
  - El identificador a secas (`tabla`) es un valor constante, que indica la dirección de memoria donde comienzan los elementos del array.
  - La posición del elemento `tabla[i]` se calcula sumando `i` a la dirección inicial `tabla`. No hay comprobación de salida de rango.
  - Por todo lo anterior, el programador deberá **manejar 2 datos para cada tabla**: dirección de inicio y nº de elementos, y es responsable de no salirse de sus límites.

Tablas (arrays)

- Utilizando tablas en el programa:
  - Se utilizará una tabla siempre que se precise un nº de datos similares o equivalentes:

```
int nacimientos[365];
float vec3D[3];         // vec3D[0]=x, vec3D[1]=y, vec3D[2]=z
float color[3];         // color[0]=R, color[1]=G, color[2]=B
float complejo[2];      // complejo[0]=real, complejo[1]=imag
char texto[20];         // caracteres ordenados = texto
```

  - Lo más habitual será acceder a los elementos de la tabla mediante bucles, donde el índice debe ser entero (entre 0 y tamaño-1).
  - Es conveniente utilizar #define para las constantes numéricas:

```
#define DIAS_ANYO   365
int nacimientos[DIAS_ANYO],defunciones[DIAS_ANYO];
int crec_anual,i;
crec_anual=0;
for (i=0;i<DIAS_ANYO;i++)
   crec_anual=crec_anual+nacimientos[i]-defunciones[i];
```

Este formato es el típico en C:
- **Empezar en índice 0**
- **Mantenerse mientras el índice es menor estricto que el tamaño**

Tablas (arrays)

- En algunos casos "especiales", es más conveniente manejar la longitud de una manera "diferente":
  - Ejemplo: codificación de un polinomio de grado 3 $p(x)=a.x^3+b.x^2+c.x+d$

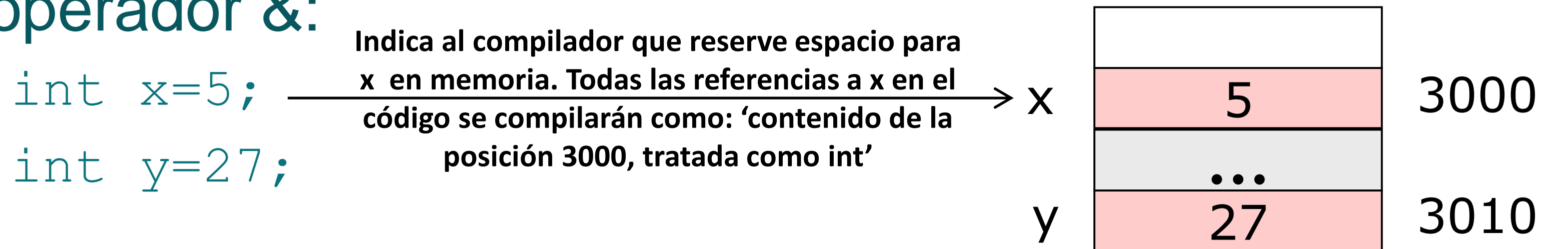
```
#define GRADO 3
float pol[GRADO+1];    // pol[0]=a, pol[1]=b, … , pol[3]=d
float x,px;
int i;
x= …valor de x…;
for (i=0,px=0;i<GRADO+1;i++)
    px=pol[i]*px+x;
```

Se necesitan 4 elementos para grado 3

Nos mantenemos en el bucle mientras el índice es menor estricto que el tamaño

Punteros (pointers)

- Un puntero es un valor (constante o variable) que contiene una posición de memoria donde se encuentra una variable.
- Su tipo de datos es *tipo** (valor entero que indica una dirección donde hay una variable del tipo *tipo*)
- Se puede obtener la dirección de una variable con el operador &:



  - `&x` : es un `int*` que vale 3000 (valor constante)
  - `&y`: es un `int*` que vale 3010 (valor constante)
- El compilador y el entorno de ejecución deciden dónde estarán x e y.

  La única forma de saberlo en el código es mediante el operador &.

Punteros (pointers)

- Las variables de tipo puntero permiten acceder de forma indirecta a una variable.
- Se declaran con:

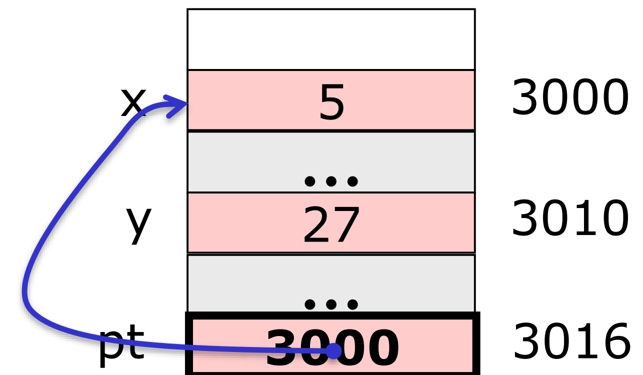
```
tipo* nombre;
```

- y permiten acceder al contenido de la dirección mediante:

```
*nombre;
```

- Ej:

```
int x=5,y=27;
int* pt;
pt=&x;
*pt=44;
pt=&y;
*pt=44;
```



Punteros (pointers)

- Las variables de tipo puntero permiten acceder de forma indirecta a una variable.
- Se declaran con:

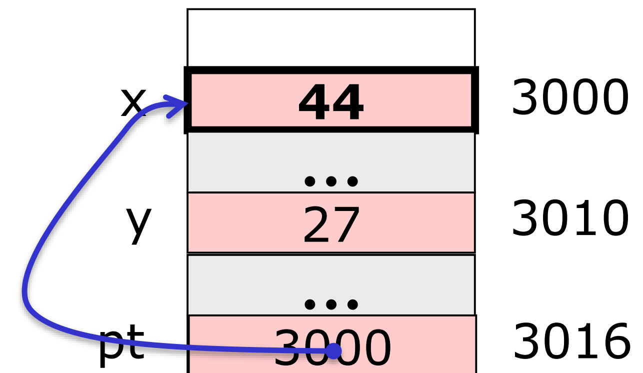
```
tipo* nombre;
```

- y permiten acceder al contenido de la dirección mediante:

```
*nombre;
```

- Ej:

```
int x=5,y=27;
int* pt;
pt=&x;
*pt=44;
pt=&y;
*pt=44;
```



Punteros (pointers)

- Las variables de tipo puntero permiten acceder de forma indirecta a una variable.
- Se declaran con:

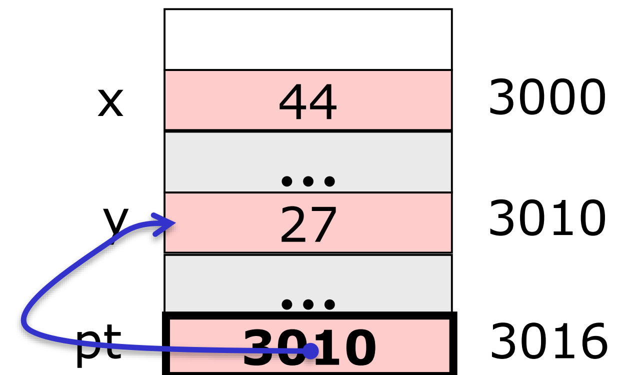
```
tipo* nombre;
```

- y permiten acceder al contenido de la dirección mediante:

```
*nombre;
```

- Ej:

```
int x=5,y=27;
int* pt;
pt=&x;
*pt=44;
pt=&y;
*pt=44;
```



Punteros (pointers)

- Las variables de tipo puntero permiten acceder de forma indirecta a una variable.
- Se declaran con:

```
tipo* nombre;
```

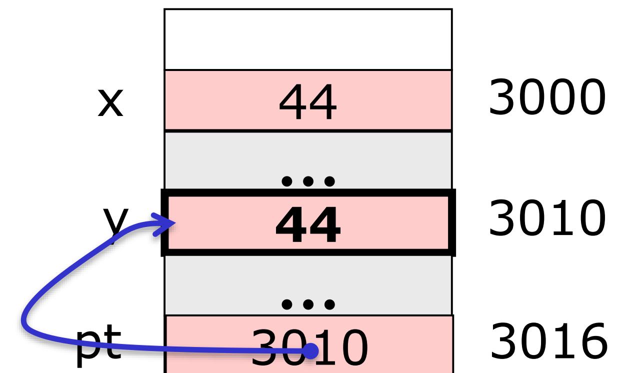
- y permiten acceder al contenido de la dirección mediante:

```
*nombre;
```

- Ej:

```
int x=5,y=27;
int* pt;
pt=&x;
*pt=44;
pt=&y;
*pt=44;
```

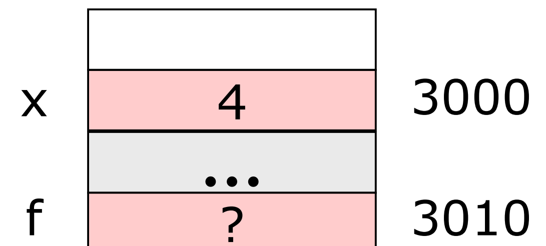
El mismo código afecta a variables diferentes gracias al puntero



Punteros (pointers)

- ¿Para qué sirven los punteros?
  - USO 1: Permiten modificar una variable de una función llamadora desde la función llamada.
  - Ejemplo: función que incremente x y devuelva x!
  - **Sin punteros: no es posible**

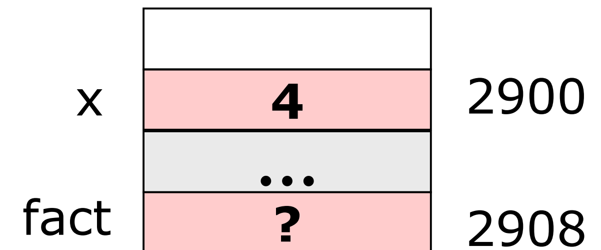
```
int Fact_con_Inc(int x)
{
    int fact;

    x=x+1;
    fact=Factorial(x);
    return fact;
}

...
// Función llamadora
    int x=4,f;
    f=Fact_con_Inc(x);
    ...
```

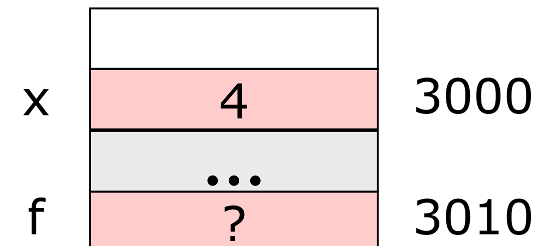


Punteros (pointers)

- ¿Para qué sirven los punteros?
  - USO 1: Permiten modificar una variable de una función llamadora desde la función llamada.
  - Ejemplo: función que incremente x y devuelva x!
  - **Sin punteros: no es posible**

```
int Fact_con_Inc(int x)
{
    int fact;

    x=x+1;
    fact=Factorial(x);
    return fact;
}

...
// Función llamadora
    int x=4,f;
    f=Fact_con_Inc(x);
    ...
```





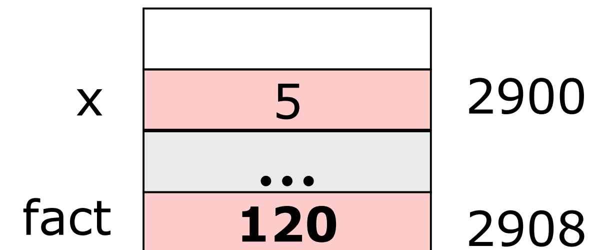
Universidad de Oviedo

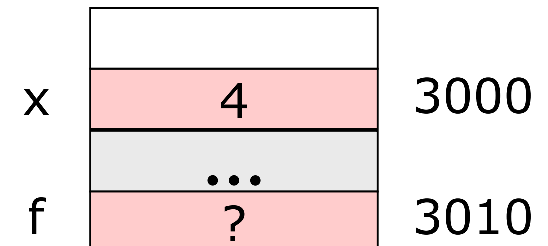
INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

Punteros (pointers)

- ¿Para qué sirven los punteros?
  - USO 1: Permiten modificar una variable de una función llamadora desde la función llamada.
  - Ejemplo: función que incremente x y devuelva x!
  - **Sin punteros: no es posible**

```
int Fact_con_Inc(int x)
{
    int fact;

    x=x+1;
    fact=Factorial(x);
    return fact;
}

...
// Función llamadora
    int x=4,f;
    f=Fact_con_Inc(x);
    ...
```

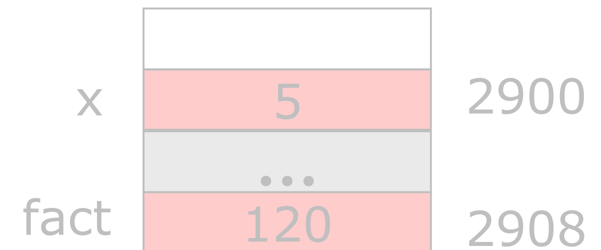


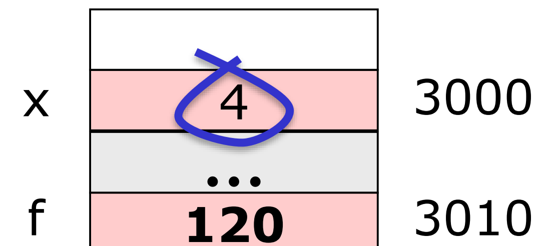


Punteros (pointers)

- ¿Para qué sirven los punteros?
  - USO 1: Permiten modificar una variable de una función llamadora desde la función llamada.
  - Ejemplo: función que incremente x y devuelva x!
  - **Sin punteros: no es posible**

```
int Fact_con_Inc(int x)
{
    int i,fact;

    x=x+1;
    fact=Factorial(x);
    return fact;
}

...
// Función llamadora
    int x=4,f;
    f=Fact_con_Inc(x);
    ...
```

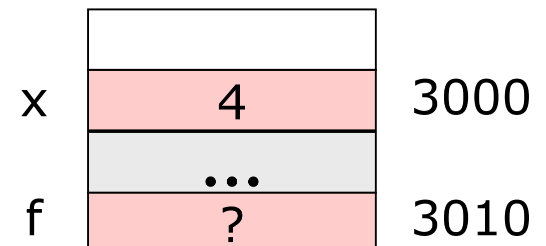




Punteros (pointers)

- ¿Para qué sirven los punteros?
  - USO 1: Permiten modificar una variable de una función llamadora desde la función llamada.
  - Ejemplo: función que incremente x y devuelva x!
  - **Con punteros: es posible**

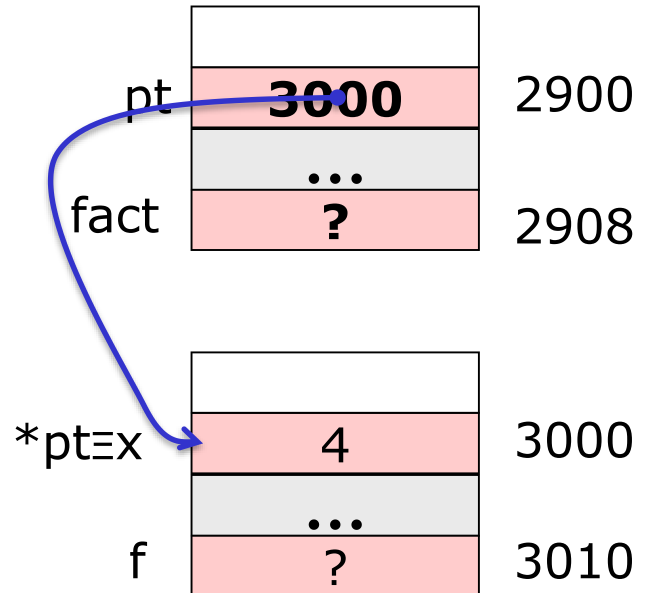
```
int Fact_con_Inc(int* pt)
{
    int fact;

    *pt=*pt+1;
    fact=Factorial(*pt);
    return fact;
}

...
// Función llamadora
    int x=4,f;
    f=Fact_con_Inc(&x);
    ...
```



Punteros (pointers)

- ¿Para qué sirven los punteros?
  - USO 1: Permiten modificar una variable de una función llamadora desde la función llamada.
  - Ejemplo: función que incremente x y devuelva x!
  - **Con punteros: es posible**

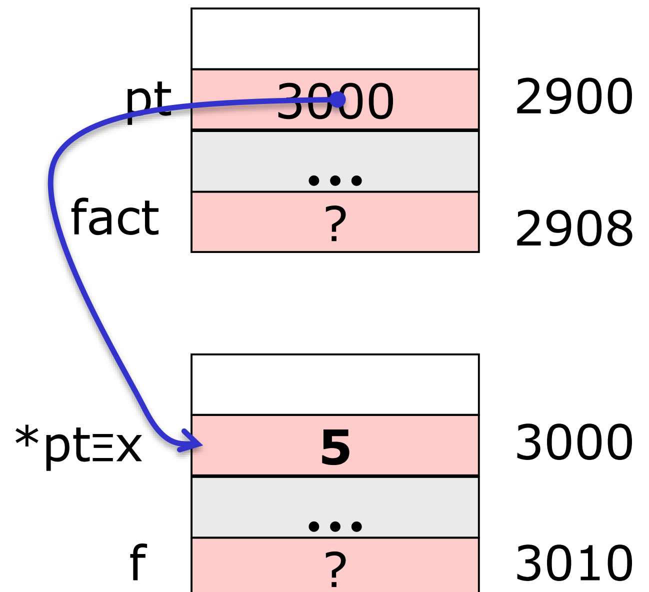
```
int Fact_con_Inc(int* pt)
{
    int fact;

    *pt=*pt+1;
    fact=Factorial(*pt);
    return fact;
}

...
// Función llamadora
    int x=4,f;
    f=Fact_con_Inc(&x);
    ...
```



Punteros (pointers)

- ¿Para qué sirven los punteros?
  - USO 1: Permiten modificar una variable de una función llamadora desde la función llamada.
  - Ejemplo: función que incremente x y devuelva x!
  - **Con punteros: es posible**

```
int Fact_con_Inc(int* pt)
{
    int fact;

    *pt=*pt+1;
    fact=Factorial(*pt);
    return fact;
}

...
// Función llamadora
    int x=4,f;
    f=Fact_con_Inc(&x);
    ...
```



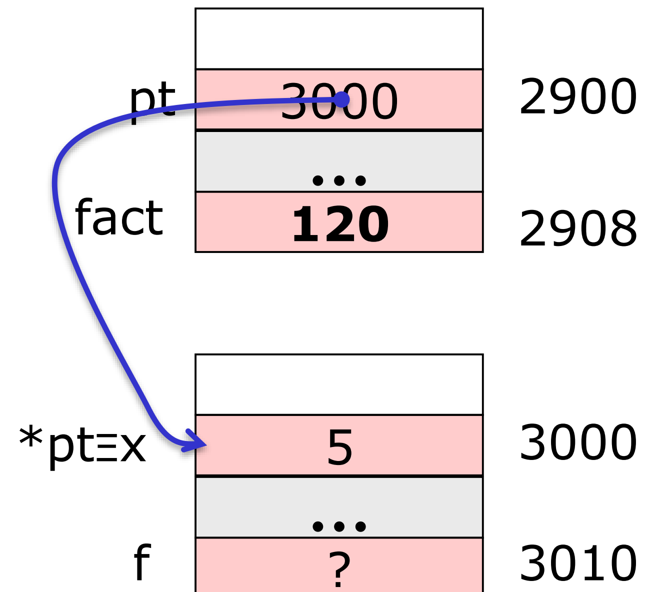
Universidad de Oviedo

INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

Punteros (pointers)

- ¿Para qué sirven los punteros?
  - USO 1: Permiten modificar una variable de una función llamadora desde la función llamada.
  - Ejemplo: función que incremente x y devuelva x!
  - **Con punteros: es posible**

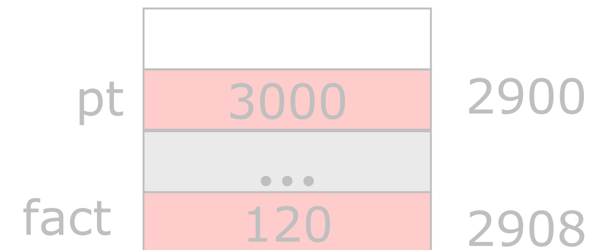
```
int Fact_con_Inc(int* pt)
{
    int fact;

    *pt=*pt+1;
    fact=Factorial(*pt);
    return fact;
}

...
// Función llamadora
    int x=4,f;
    f=Fact_con_Inc(&x);
    ...
```

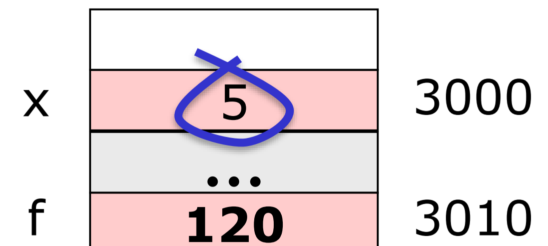


Punteros (pointers)

- ¿Para qué sirven los punteros?
  - USO 1: Permiten modificar una variable de una función llamadora desde la función llamada.
  - Ejemplo: función que incremente x y devuelva x!
  - **Con punteros: es posible**

```
int Fact_con_Inc(int* pt)
{
    int fact;

    *pt=*pt+1;
    fact=Factorial(*pt);
    return fact;
}

...
// Función llamadora
    int x=4,f;
    f=Fact_con_Inc(&x);
    ...
```





Punteros (pointers)

- ¿Para qué sirven los punteros?
  - USO 1: Permiten modificar una variable de una función llamadora desde la función llamada.
  - ¡ Ya habíamos dado este uso en la función scanf() !

```
int x;
printf("Introduce valor de x: ");
scanf("%d",&x);          // scanf() necesita &x para modificar x
printf("x = %d\n",x);    // printf() no necesita &x porque no
                         //           va a modificar x
```

Punteros (pointers)

- ¿Para qué sirven los punteros?
  - USO 2: Permiten usar tablas declaradas en otras funciones.
  - Ejemplo: función que calcula la media

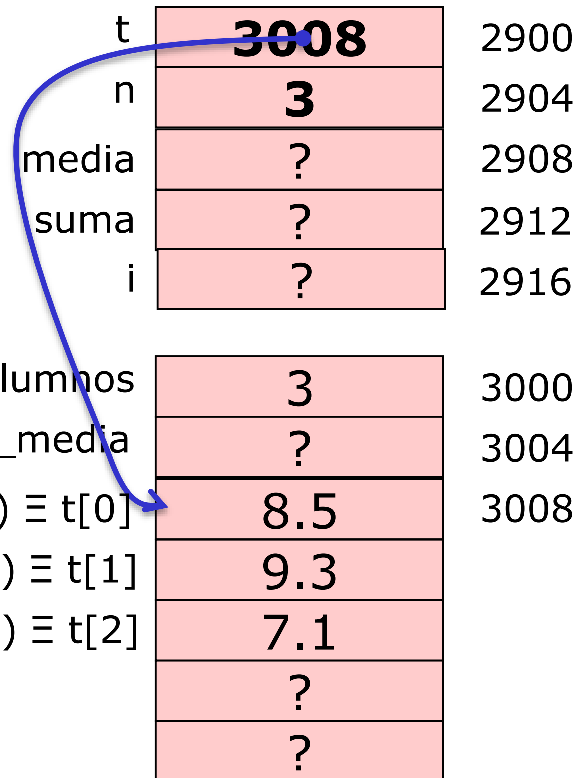
```
float Media(float* tabla,int n)
// Un parámetro no puede ser una tabla completa,
// sólo su dirección de comienzo (el C no tiene noción
// de la tabla completa una vez declarada).
// float* tabla y float tabla[] son equivalentes.
{
    float media,suma;
    int i;
    suma=0;
    for (i=0;i<n;i++)
          suma=suma+tabla[i];
    media=suma/n;
    return media;
}
...
// Función llamadora
    float notas[5],n_alumnos;
    float nota_media;
    ... Dar valores a n_alumnos (ej. 3) y notas[i] ...
    nota_media=Media(notas,n_alumnos);
    ...
```

Variable	Valor	Dirección
n_alumnos	3	3000
nota_media	?	3004
notas →	8.5	3008
	9.3	
	7.1	
	?	
	?	

Punteros (pointers)

- ¿Para qué sirven los punteros?
  - USO 2: Permiten usar tablas declaradas en otras funciones.
  - Ejemplo: función que calcula la media

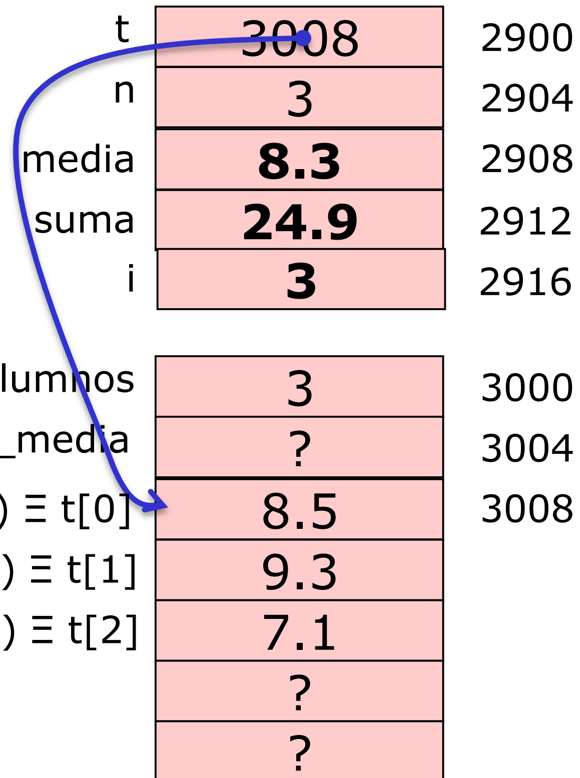
```
float Media(float* t,int n)
// Un parámetro no puede ser una tabla completa,
// sólo su dirección de comienzo (el C no tiene noción
// de la tabla completa una vez declarada).
// float* t y float t[] son equivalentes.
{
    float media,suma;
    int i;
    suma=0;
    for (i=0;i<n;i++)
          suma=suma+t[i];
    media=suma/n;
    return media;
}
...
// Función llamadora
    float notas[5],n_alumnos;
    float nota_media;
    ... Dar valores a n_alumnos (ej. 3) y notas[i] ...
    nota_media=Media(notas,n_alumnos);
    ...
```



Punteros (pointers)

- ¿Para qué sirven los punteros?
  - USO 2: Permiten usar tablas declaradas en otras funciones.
  - Ejemplo: función que calcula la media

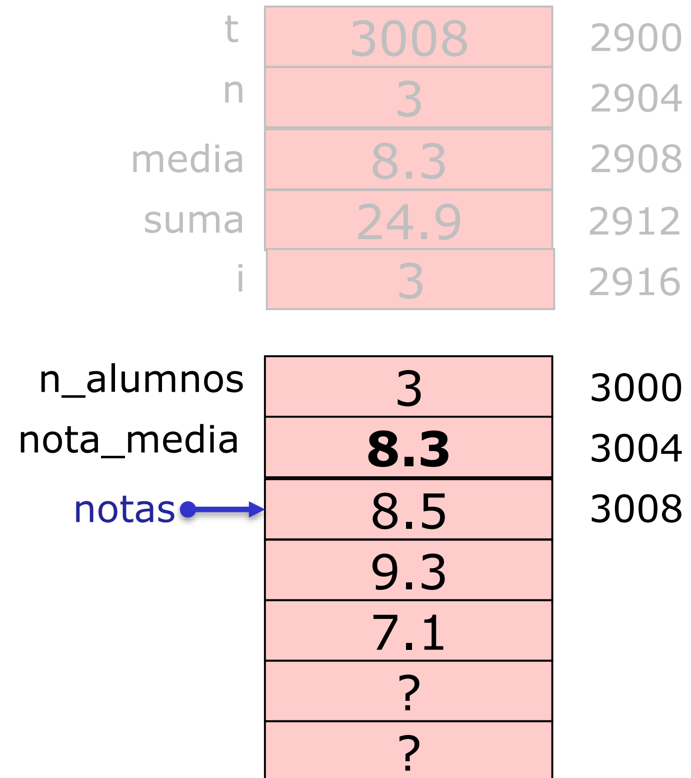
```
float Media(float* t,int n)
// Un parámetro no puede ser una tabla completa,
// sólo su dirección de comienzo (el C no tiene noción
// de la tabla completa una vez declarada).
// float* t y float t[] son equivalentes.
{
    float media,suma;
    int i;
    suma=0;
    for (i=0;i<n;i++)
          suma=suma+t[i];
    media=suma/n;
    return media;
}
...
// Función llamadora
    float notas[5],n_alumnos;
    float nota_media;
    ... Dar valores a n_alumnos (ej. 3) y notas[i] ...
    nota_media=Media(notas,n_alumnos);
    ...
```



Punteros (pointers)

- ¿Para qué sirven los punteros?
  - USO 2: Permiten usar tablas declaradas en otras funciones.
  - Ejemplo: función que calcula la media

```
float Media(float* t,int n)
// Un parámetro no puede ser una tabla completa,
// sólo su dirección de comienzo (el C no tiene noción
// de la tabla completa una vez declarada).
// float* t y float t[] son equivalentes.
{
    float media,suma;
    int i;
    suma=0;
    for (i=0;i<n;i++)
          suma=suma+t[i];
    media=suma/n;
    return media;
}
...
// Función llamadora
    float notas[5],n_alumnos;
    float nota_media;
    ... Dar valores a n_alumnos (ej. 3) y notas[i] ...
    nota_media=Media(notas,n_alumnos);
    ...
```



Punteros y tablas

- Algunas consideraciones **importantes** (I):
  - Una tabla sólo es tabla en el instante de su declaración como variable local. A partir de ahí, su identificador es un puntero constante: el lenguaje C no tiene noción de tabla.
  - Si se declara una tabla como parámetro, no se está declarando una tabla completa, sólo un puntero (su dirección inicial). Siempre se necesitará un 2º parámetro que indique la longitud.
  - Los punteros (y tablas por tanto) son "peligrosos", ya que permiten acceder a "cualquier" posición de memoria.
  - Se puede evitar parcialmente el "peligro" de tablas y punteros usando el modificador **const**: impide modificar el contenido de las direcciones apuntadas.

```
float Media(float tabla[],int n)
{
    ...
    tabla[i]=xx; // ¿Deseado?
}
```

```
float Media(const float tabla[],int n)
{
    ...
    tabla[i]=xx;  // Error de compilación
}
```

Punteros y tablas

- Algunas consideraciones **importantes** (II):
  - Una función no puede devolver una tabla, sólo modificar una de la función llamante, cuya dirección inicial se pase como parámetro.
  - Siempre que se pase una "tabla" a una función, hay que utilizar la dirección de comienzo y el nº de elementos. Una función será más reutilizable si sirve para cualquier nº de elementos.

```
// ¡¡¡¡INCORRECTO!!!!
// ¡OJO! Compila pero ejecuta mal
int* FuncionDevuelveTabla(...)
{
      int t[10];

      ... Dar valores a t[i]...
      return t;
}
```

```
// CORRECTO

void FuncionDevuelveTabla(int t[],int n)
{
      ... Dar valores a t[i]...
}
```

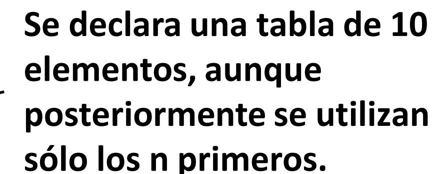
No se maneja información del tamaño de la tabla. La tabla t "desaparece" cuando se deje de ejecutar la función, por lo que el valor devuelto se refiere a una dirección de memoria "desasignada".

Se maneja información del tamaño de la tabla. El puntero t se refiere a una tabla que existe (y seguirá existiendo) en la función llamante.

Punteros y tablas

- Algunas consideraciones **importantes** (III):
  - El tamaño de una tabla debe ser declarado en tiempo de compilación (valor constante).
  - Si no se conoce, se asignará en la declaración un tamaño mayor o igual que el máximo posible, y se usará solamente la parte que se desee.

```
#define NMAX   10

...
      int t[NMAX];
      int n;

      ...

      ... Dar valor a n ...
      ... Asegurar que n>=0 && n<= NMAX ...
      ... Usar t[i], i=0 hasta n-1 ...

      ...
}
```



Punteros y tablas

- Asignación dinámica de memoria:
  - Es posible reservar el tamaño de una tabla en tiempo de ejecución, utilizando funciones especiales (incluir <malloc.h>):

```
void* malloc(int nBytes);     // Busca y reserva memoria para nBytes
                              // consecutivos, y devuelve dirección de
                              // comienzo (ó NULL si no se dispone)
void free(void* memAsignada); // Libera memoria asignada con malloc
```

```
...
     float *t;
     int n;

     ...

     ... Dar valor a n ...
     t=(float*) malloc(n*sizeof(float));
     if (t!=NULL)
     {
          ... Usar t[i], i=0 hasta n-1 ...
          free(t);
     }
     ...
}
```

Se declara una variable tipo puntero que contendrá la dirección de inicio de la tabla.

Se solicita espacio para una tabla que necesita n*4 bytes (4 es el nº de bytes que ocupa un float, lo devuelve el operador sizeof).

Se comprueba que se ha podido asignar la memoria solicitada

Se utilizan normalmente los elementos de la tabla, gracias la dualidad tabla ↔ puntero.

Se debe liberar la memoria asignada cuando ya no sea necesaria.

Punteros, tablas y estructuras

- Un campo de una estructura puede ser una tabla o un puntero.
- Ejemplo:

```
#define N_MAX   100
struct vector
{
    int n;
    float datos[N_MAX];
} ;

float Sumatorio(struct vector v)
{
    ...
}
```

- **¡¡¡ ATENCION !!!**
  - De esta manera, se puede conseguir que una función reciba o devuelva una tabla completa.

PERO NO SE SUELE USAR PORQUE …

  - Al usar tablas estáticas como campos de estructura, se copian completas al pasar o devolver en funciones, lo que puede llevar un gran consumo de tiempo.

Punteros, tablas y estructuras

- Si se desea pasar estructuras grandes a funciones, es más conveniente usar punteros a estructura.
- Ejemplo:

```
#define N_MAX   100
struct vector
{
    int n;
    float datos[N_MAX];
} ;

float Sumatorio(const struct vector* v)
{
    ...
}
```

- Para acceder a los campos de la estructura dado un puntero:
  - (*v).n

  O mejor con el operador flecha (guión seguido de mayor que):
  - v->n

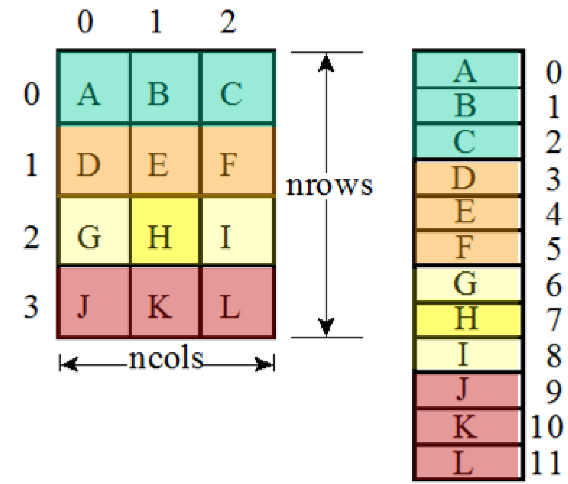
Punteros, tablas y estructuras

- Un ejemplo con tablas y estructuras: matrices
- Una matriz se puede definir como una estructura con 3 campos:

```
struct matrix
{
    int rows,cols;
    float *data;
} ;
```

- Dado que el nº de elementos de una matriz es muy variable, se debe usar asignación dinámica de memoria para los datos.
- Los datos se organizan en posiciones de memoria consecutivas en formato row-major (0-based index):

  $elem_{i,j} \rightarrow data[i*cols+j]$



Punteros, tablas y estructuras

- Ejemplo de funciones:

```
struct matrix AsignaMemoriaMatriz(int nf,
                                  int nc)
{
  struct matrix m;
  m.rows=nf;
  m.cols=nc;
  m.data=(float*) malloc(nf*nc*sizeof(float));
  return m;
}

void LiberaMemoriaMatriz(struct matrix *m)
{
  m->rows=m->cols=0;
  free(m->data);
  m->data=NULL;
}
```

```
struct matrix SumaMatrices(const struct matrix m1,
                           const struct matrix m2)
{
  struct matrix suma;
  if (m1.rows==m2.rows && m1.cols==m2.cols)
  {
    int i,j,offset;
    suma=AsignaMemoriaMatriz(m1.rows,m1.cols);
    for (i=0;i<m1.rows;i++)
    {
       for (j=0;j<m1.cols;j++)
       {
          offset=(i*m1.cols+j);
          suma.data[offset]=m1.data[offset]+m2.data[offset];
       }
    }
  }
  else
  {
     suma.rows=suma.cols=0;
     suma.data=NULL;
  }
  return suma;
}
```

Ejercicios propuestos

- Realizar utilizando funciones:
  1) Calcular la desviación típica de los datos de una tabla de 5 elementos.

$$\sigma = \sqrt{\frac{\sum (x_i - \overline{x})^2}{n-1}}$$

  2) Calcular el índice donde se produce el valor mínimo de una tabla de 5 elementos, y escribir el índice y el valor mínimo.
  3) Calcular la mediana de los valores de una tabla de 5 elementos (obtener una tabla auxiliar ordenada de menor a mayor, de ésta sacar el elemento central).
  4) Actualizar la función del ejercicio anterior para que devuelva un código de error si el nº de elementos no es impar.
  5) Actualizar el ejercicio anterior anterior para que las tablas tengan declaración dinámica.
  6) Realizar una función que mueva cada elemento de una tabla al siguiente, y actualice el primer elemento con un nuevo valor.

Indice

- Introducción al computador y el lenguaje C
- Datos, expresiones y algoritmos
- Funciones
- Tablas y punteros
- **Cadenas de caracteres**
- Operaciones con valores binarios
- Preprocesador y compilación separada
- E/S en archivos y dispositivos
- Criterios de buena programación

Codificación de texto

- Al igual que otros conceptos "abstractos" (días de la semana, tipos de fruta, etc.), el texto se codifica como números enteros:
  - Cada carácter de un texto está codificado por un valor entero.
  - Todos los computadores deben utilizar la misma codificación para entenderse entre ellos: códigos ASCII y UNICODE.
  - Puesto que no hay muchos caracteres diferentes, no es necesario un elevado nº de bits para codificarlos todos:
    - ASCII: 8 bits = 256 caracteres diferentes.
    - UNICODE: 16 bits = 65536 caracteres diferentes.
  - Se utiliza el tipo de datos **char** para indicar un entero con el nº de bits necesarios (8/16) para codificar un carácter.
  - Para declarar constantes se pone el carácter entre comillas simples, para evitar recordar el código de cada carácter.
    - Ejs: 'a' ≡ 97 'b' ≡ 98 'c' ≡ 99 '.' ≡ 46 '0' ≡ 48 '1' ≡ 49 '\n' ≡ 10

Codificación de texto

```
char letra;
letra = 'a';
```

Lo que realmente existe

letra **01100001**

Es un entero que ocupa 1 byte en memoria (8 bits) (una dirección) por ser char

Lo que interpreto (I)

97

Como entero:

A cada bit le corresponde un peso según su posición y la codificación ponderada en base 2:

$1*2^6+1*2^5+1*2^0 = 97$

Las operaciones aritméticas (+,-,*,/) entre valores interpretados de esta forma dan lugar a valores interpretados de la misma forma Así que:

suma = letra + 4;

Hace que suma valga

$1*2^6+1*2^5+1*2^2+1*2^1+1*2^0 = 103$

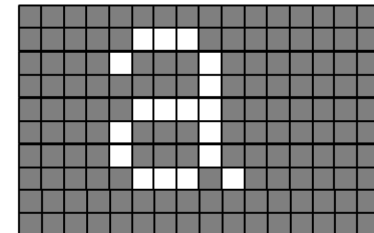
Lo que interpreto (II)

'a'

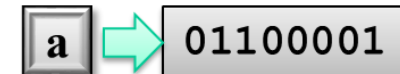
Como carácter:

En una pantalla, esta secuencia de bits se convierte en los puntitos que nos hacen ver la letra 'a', No hay ninguna 'a' hasta que un circuito (hardware del dispositivo pantalla) genera los puntitos

Hardware Pantalla



En un teclado es aún más evidente: ¡¡¡ la 'a' es una pegatina !!! Simplemente, cuando se pulsa sobre la tecla que tiene esa pegatina, el hardware de teclado envía al computador el código binario



Universidad de Oviedo

INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

La tabla de códigos ASCII

- ASCII utiliza 8 bits = 256 valores diferentes:

Estándar (0 a 127)

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

Extendidos (128 a 255)

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	□

Operaciones con caracteres

- Se opera igual que con datos enteros (son enteros).

```
char c1,c2;
c1='a';          // c1=97=0x61=0b01100001
c2=c1+2;         // c2=99=0x63=0b01100011='c'
if (c2>='a' && c2<='z')
     printf("El carácter %c es una letra minúscula\n",c2);
```

- Existen funciones específicas de interés:
  - char getchar(); // Pide un carácter por consola
  - putchar(char c); // Escribe un carácter en consola
  - Se utiliza %c en printf() y scanf()
  - int isalpha(char c); // Devuelve Verdadero si c está entre ‘a’…’z’ ó ‘A’..’Z’
  - char toupper(char c); // Devuelve mayúscula si era minúscula.
  - …
- ¡ Ojo con caracteres extendidos !
  - ‘Ñ’, ‘ñ’, ‘á’, ‘é’, etc. : no siguen reglas de ordenación.
- Códigos para caracteres “especiales” comienzan con \:
  - ‘\n’, ‘\0’, ‘\’’, ‘\”’, ‘\\’, …

Cadenas de caracteres (string)

- Un texto es una secuencia ordenada de caracteres, por tanto en C será un array de char (en inglés string).
  - Como en todo array, se requiere su dirección de comienzo y su longitud.

```
char txt[4];
int n=4;
txt[0]='H';
txt[1]='o';
txt[2]='l';
txt[3]='a';
```

¡ Ojo ! No es la forma en que se hace.

  - Por comodidad, en los arrays de caracteres (string) no se maneja la longitud con una 2ª variable, sino que se gestiona colocando un código especial (0) al final de la tabla.

```
char txt[5];
txt[0]='H';
txt[1]='o';
txt[2]='l';
txt[3]='a';
txt[4]=0;
```

¡ Ojo ! Para el texto *Hola* (4 caracteres) se debe declarar una tabla con un elemento más.
El string queda definido únicamente por su dirección de comienzo (txt). El final se produce donde se encuentre el carácter de código 0 (0 = '\0' ≠ '0' = 48)

Cadenas de caracteres (string)

- Se pueden manejar constantes de tipo string, colocando el texto entre comillas dobles:

```
char txt[5]="Hola";
```

txt[0]	'H'	3000 = txt
txt[1]	'o'	3001
txt[2]	'l'	3002
txt[3]	'a'	3003
txt[4]	0	3004
txt[5]	?	3005
…	…	…

- Una constante tipo string es un const char*, y se puede utilizar en cualquier lugar del código que lo acepte:

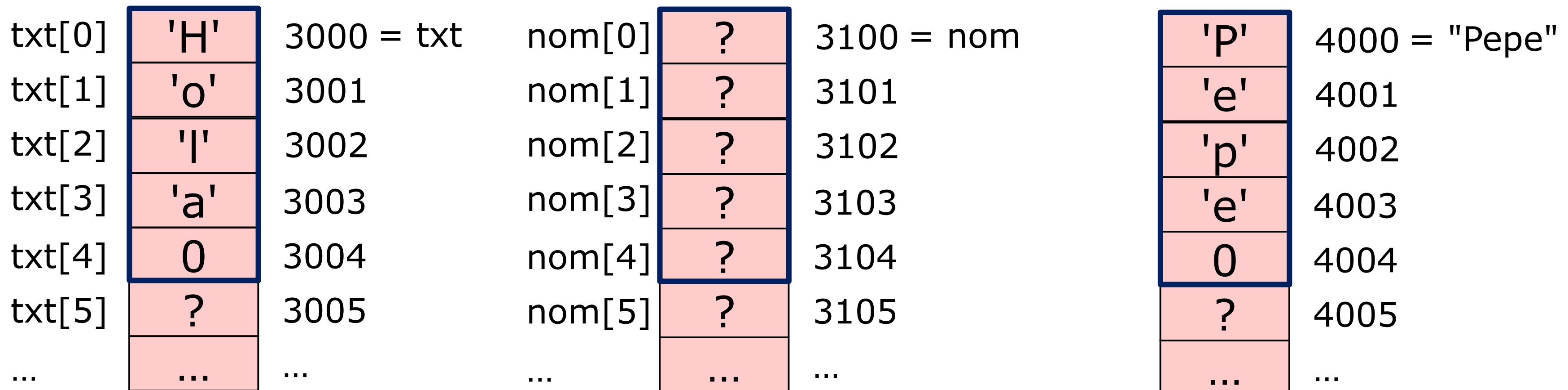
```
// En <stdio.h>
int printf(const char* fmt,...);

// En nuestro programa
int x=3,z=2;
printf("x vale %d, y vale %d\n",x,y);
```

printf() recibe como parámetro la dirección de comienzo de una tabla de char, que se puede obtener de una constante string.

Cadenas de caracteres (string)

- Sólo se puede asignar el contenido de un string a una constante en el momento de su declaración:

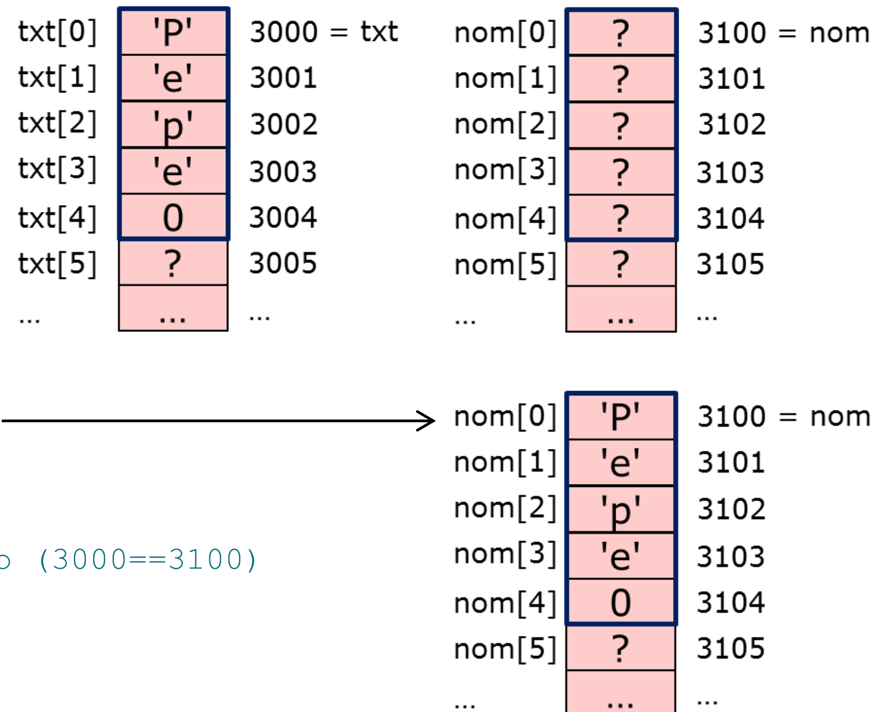
```
char txt[5]="Hola";          // Correcto
char nom[5];
nom="Pepe";                  // Incorrecto, ¡ equivale a 3100=4000 !
nom=txt;                     // Incorrecto, ¡ equivale a 3100=3000 !
```



Cadenas de caracteres (string)

- Todas las operaciones con string requieren recorrer los elementos de la tabla uno por uno, terminando al alcanzar el carácter 0:

```
char txt[5]="Pepe";
char nom[5];
int i,iguales;

// Copiar txt a nom
for (i=0;txt[i]!=0;i++)
    nom[i]=txt[i];
nom[i]=0;
// Aquí: nom vale lo mismo que txt

// Comparar nom y txt
if (nom==txt) // MAL: Siempre falso (3000==3100)
   …
iguales=1;
for (i=0;txt[i]!=0;i++)
    iguales=iguales && (txt[i]==nom[i]);
// Aquí: iguales (V ó F) indica si son iguales
```



Universidad de Oviedo

Operaciones con strings

- Funciones para E/S de strings:
  - char* gets(char* cad); // Pide una cadena por consola
  - int puts(const char* cad); // Escribe una cadena en consola
  - Se utiliza %s en printf() y scanf()
- Ejemplo:

```
char nombre[20];
char apellido[20];

printf("Nombre: ");
gets(nombre);
printf("Apellido: ");
gets(apellido);

printf("Nombre completo: %s %s\n",nombre,apellido);
```

Funciones de cadena de caracteres

- Realizan los bucles más típicos en el tratamiento de strings (copiar, comparar, añadir, buscar, …)
- Trabajan con tablas de char terminadas en el caracter 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)*
  - …

El valor devuelto es el mismo que dst, así que no se suele utilizar (strcpy, strncpy, strcat, strncat).

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* 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 cadena de caracteres

- Ejemplo: pedir nombre y apellido, unirlos, y comprobar si se trata de “Marie Curie”.

```
char nombre[20];
char apellido[20];
char* todo;

printf("Nombre: ");
gets(nombre);
printf("Apellido: ");
gets(apellido);

todo=(char*) malloc(strlen(nombre)+1+strlen(apellido)+1);
strcpy(todo,nombre);
strcat(todo," ");
strcat(todo,apellido);
if (strcmp(todo,"Marie Curie")==0)
    puts("Es una científica\n");
free(todo);
```

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 cadena de caracteres

- Ejemplo: pedir comando de texto y extraer valor entero en una variable (…. POS = 90 ….)

```
char texto[40];
int posicion;
char* cmd;

printf("Comando: ");
gets(texto);

cmd=strstr(texto,"POS");
if (cmd!=NULL)
{
    for (cmd=cmd+strlen("POS");*cmd==' ';cmd++)
          ;
    if (*cmd=='=')
          posicion=atoi(cmd+1);
}
```

Cadenas de caracteres

- A tener en cuenta al usar cadenas de caracteres:
  - Son tablas como cualquier otra: requieren su reserva de espacio en memoria, no hay comprobación de salida de rango, etc.
  - No se usa la longitud de la tabla, sino el código especial 0. ¡ No se puede olvidar añadir el carácter 0 al manipular strings !
  - Se pueden manipular:
    - Carácter por carácter, usando índices o punteros.
    - Usando funciones de la librería estándar.
    - Creando funciones propias, con los mismos criterios vistos para tablas (excepto el parámetro longitud, que no suele utilizarse).
  - Pero siempre hay que asegurar que el tamaño reservado es capaz de albergar los datos de la tabla:

```
char txt[5]="Pepe";

strcat(txt," Pérez");          // Incorrecto: en txt no hay sitio para todo
```

  - Es muy habitual el uso de asignación dinámica de memoria, ya que son tablas de tamaño muy variable.

Ejercicios propuestos

1) Pedir por consola el nombre de una unidad lógica, directorio y archivo, y componer el camino completo hacia el archivo (ej: “c:\user\practicas\ejemplo.txt”).
2) Comprobar si el archivo anterior tiene extensión “txt” y se encuentra en el subdirectorio “practicas”.
3) Realizar una función que, dado un comando con el formato “POSICION=valor entero”, devuelva el valor entero. Se debe admitir que la palabra clave POSICION sea POS ó POSI ó POSIC ó … ó POSICION, y que haya espacios en blanco antes y después del ‘=‘.
4) Añadir a la función anterior la posibilidad de devolver un código de error si no se encuentra el comando o el formato es incorrecto.
5) Pedir por consola una cadena con el formato “[v1,v2,v3,…vn]” y extraer de la misma los contenidos de una tabla de float con esos valores.

Indice

- Introducción al computador y el lenguaje C
- Datos, expresiones y algoritmos
- Funciones
- Tablas y punteros
- Cadenas de caracteres
- **Operaciones con valores binarios**
- Preprocesador y compilación separada
- E/S en archivos y dispositivos
- Criterios de buena programación

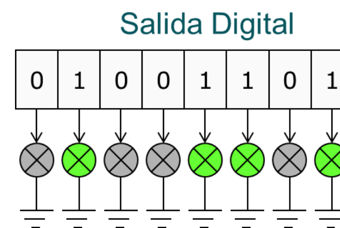
Operaciones con valores binarios

- Recordatorio: un entero se codifica en el computador con valores binarios (codificación ponderada en base 2).

- `int x=23;` x 000...010111

El nº de bits utilizados depende del tipo de datos y del procesador. Se puede saber con el cálculo 8*sizeof(int).

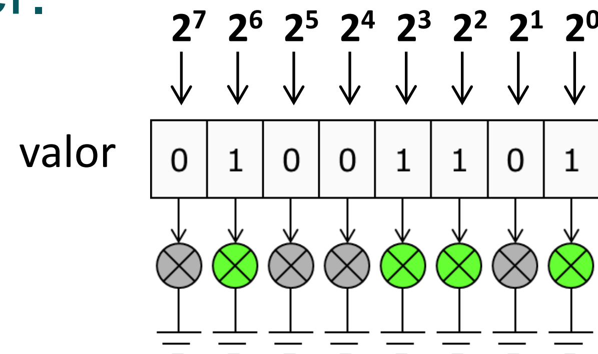
- En ciertas ocasiones, el programador puede tener interés en el valor particular de algún(os) bit(s), y no en el valor completo (la cantidad que representa):
  - Cuando los bits representan el estado de valores binarios diferentes (ej. E/S digital).



  - Cuando se desea comprimir informaciones de varios valores en un solo entero.

Operaciones con valores binarios

- En los casos anteriores, el valor decimal y las operaciones aritméticas aportan poca información o difícil de extraer.



- `valor=77;      // 2⁶+2³+2²+2⁰`
  - ¿Está activa la luz de peso 2?
  - Quiero activar la luz de peso 4.
  - Quiero cambiar el estado de las luces de peso 6 y 7 para generar una intermitencia.

Operadores lógicos de bit

- Operadores de C para manejar bits (sólo valores enteros)
- Binarios (2 operandos):

$\forall$ i=0..n-1

  - & (y): entA **&** entB → entR — $BIT_i(entR) = BIT_i(entA)$ 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):

$\forall$ i=0..n-1

~ entA → entR — $BIT_i(entR) = \overline{BIT_i(entA)}$

Ejemplos:

```
int v1=77;        00..01001101
int v2=23;        00..00010111
v1 & v2    ->     00..00000101 (=5)
v1 | v2    ->     00..01011111 (=95)
v1 ^ v2    ->     00..01011010 (=90)
~v1        ->     11..10110010 (=-78)
```

Operadores de desplazamiento de bits

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

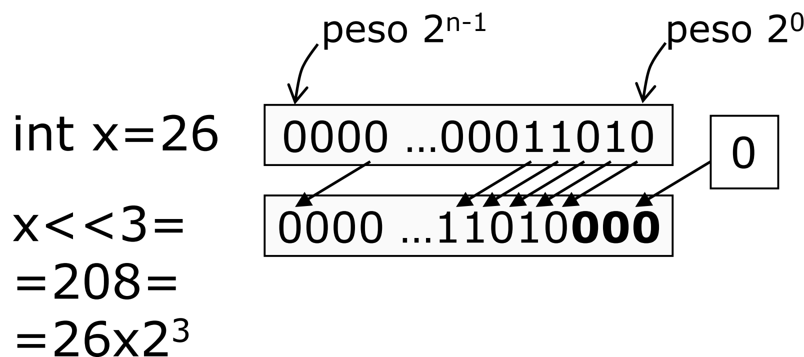
    $\forall$ i=entB..n-1 BIT$_i$(entR)= BIT$_{i\text{-entB}}$(entA)

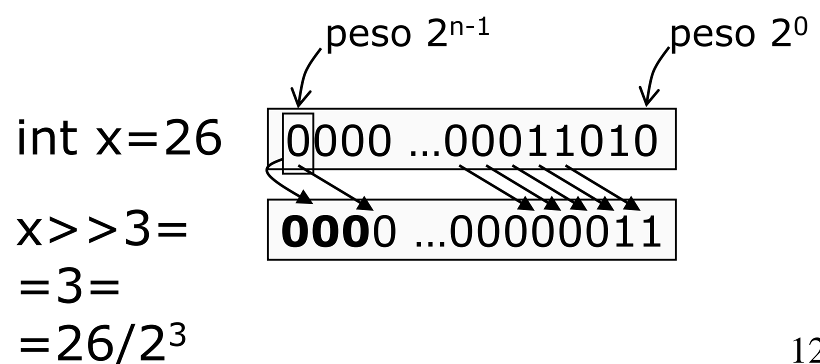
    $\forall$ i=0..entB-1 BIT$_i$(entR)= 0

  - >> (desplazamiento a derecha): entA **>>** entB → entR

    $\forall$ i=0..entB-1 BIT$_i$(entR)= BIT$_{i\text{+entB}}$(entA)

    $\forall$ i=entB..n-1 BIT$_i$(entR)=
    - 0: si entA es unsigned
    - Bit signo entA: si entA es signed



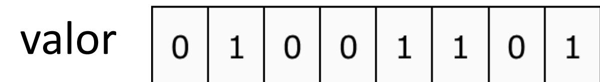


Operaciones con valores binarios

- Para indicar constantes de cuyo valor interesan los bits, se usa la codificación hexadecimal (base 16, dígitos 0..9A..F).
  - Un código hexadecimal equivale a 4 dígitos binarios:

$0_h=0000_b$	$1_h=0001_b$	$2_h=0010_b$	$3_h=0011_b$
$4_h=0100_b$	$5_h=0101_b$	$6_h=0110_b$	$7_h=0111_b$
$8_h=1000_b$	$9_h=1001_b$	$A_h=1010_b$	$B_h=1011_b$
$C_h=1100_b$	$D_h=1101_b$	$E_h=1110_b$	$F_h=1111_b$

  - En C, las constantes enteras en hexadecimal se preceden de 0x
    - `valor=77;` // $2^6+2^3+2^2+2^0$
    - `valor=0x4D;` // $4_h=0100_b$, $D_h=1101_b$

valor


  - Se pueden leer/escribir valores en hexadecimal utilizando %x en printf() y scanf() (no usar 0x)

Operaciones con valores binarios

- Operaciones habituales con el bit de peso *p* de la variable entera *x* (afectando sólo a ese bit):

```
int x,p;
int mascara;
…Dar valores a p y x…
mascara=1<<p;            // mascara = 0..00100..000
                         // 1<<p = 2^p
```

2^p ↓ (señala el bit 1 de la máscara)

  - ¿Está activo el bit de peso *p* de la variable *x*?

```
if (x & mascara)
   …
```

  - Poner a 1 el bit de peso *p* de la variable *x*:

```
x=x | mascara;
```

  - Poner a 0 el bit de peso *p* de la variable *x* :

```
x=x & ~mascara;
```

  - Cambiar el valor del bit de peso *p* de la variable *x* :

```
x=x ^ mascara;
```

Operaciones con valores binarios

- Operaciones habituales con varios datos comprimidos en un entero:

2^5 2^0

```
int v;                      // v = … zzyxxx
int vx,vy,vz;

// Extraer vx,vy,vz de v
…Dar valor a v …
vx= v & 0x7;                // vx= … 000xxx
vy= (v & 0x8) >> 3;         // vy= … 00000y
vz= (v & 0x30) >> 4;        // vz= … 0000zz
```

Máscara Z

Rotación Z

```
// Cambiar valor de los campos de v
…Dar valores a vx,vy,vz …
v= (v & ~0x7) | (vx & 0x7);         // v= … zzyxxx
v= (v & ~0x8) | ((vy << 3) & 0x8); // v= … zzyxxx
v= (v & ~0x30)| ((vz <<4) & 0x30); // v= … zzyxxx
```

Rotación Z

Máscara Z

Operaciones con valores binarios

- Calcular máscara y rotación para un bit de comienzo y un nº de bits de un campo
  - Ej: `v = … zzyxxx` bitstart_z = 4, nbits_z=2
  - `Rotación = bitstart_z`
  - `(1<<n)    = 0001000000` (2^n)
  - `(1<<n)-1  = 0000111111` (n)
  - `Máscara   = (((1<<(bitstart_z+nbits_z))-1) & ~((1<<bitstart_z)-1))`

    0…0111111 & 1…1110000 = 0…0110000

```
#define BITSTART_X    0
#define NBITS_X       3
#define MASK_X        (((1<<(BITSTART_X+NBITS_X))-1) & ~((1<<BITSTART_X)-1))
#define BITSTART_Y    (BITSTART_X+NBITS_X)
#define NBITS_Y       1
#define MASK_Y        (((1<<(BITSTART_Y+NBITS_Y))-1) & ~((1<<BITSTART_Y)-1))
#define BITSTART_Z    (BITSTART_Y+NBITS_Y)
#define NBITS_Z       2
#define MASK_Z        (((1<<(BITSTART_Z+NBITS_Z))-1) & ~((1<<BITSTART_Z)-1))

vz= (v & MASK_Z) >> BITSTART_Z;     // vz= … 0000zz
```

Ejercicios propuestos

1) Realizar una función que escriba en pantalla una cadena de caracteres que indique el contenido binario de un valor.
2) Realizar una función que pida por teclado una cadena de caracteres que contenga 0s y 1s, y devuelva el valor entero que codifica a dicho valor binario.

- Usando el simulador (utilizar función Sleep(200) para temporizar):

3) Realizar un programa que encienda las luces en función del estado de activación de los interruptores.
4) Realizar una programa que haga parpadear las luces cuyos interruptores están activos.
5) Realizar un programa que permita visualizar una luz que va desplazándose de izquierda a derecha (de forma circular).
6) Realizar un programa que hace parpadear las luces 0 a 3 en función del estado de sus interruptores, y que a la vez traslade una sola luz entre 4 y 7.

Indice

- Introducción al computador y el lenguaje C
- Datos, expresiones y algoritmos
- Funciones
- Tablas y punteros
- Cadenas de caracteres
- Operaciones con valores binarios
- **Preprocesador y compilación separada**
- E/S en archivos y dispositivos
- Criterios de buena programación

Universidad de Oviedo

INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

Alto nivel → código máquina

- Pasos en la obtención de código máquina:
  - Escribir código fuente (programa editor)
  - Compilar código fuente (programa compilador)
  - Enlazar código objeto y librerías (programa enlazador)

Editor

.C
```
main()
{
   int i;
   . . .
   printf("x=%d..
}
```

Compilador

.OBJ
```
_main:
00011100
00010100
...
exports _main
extern _printf
```

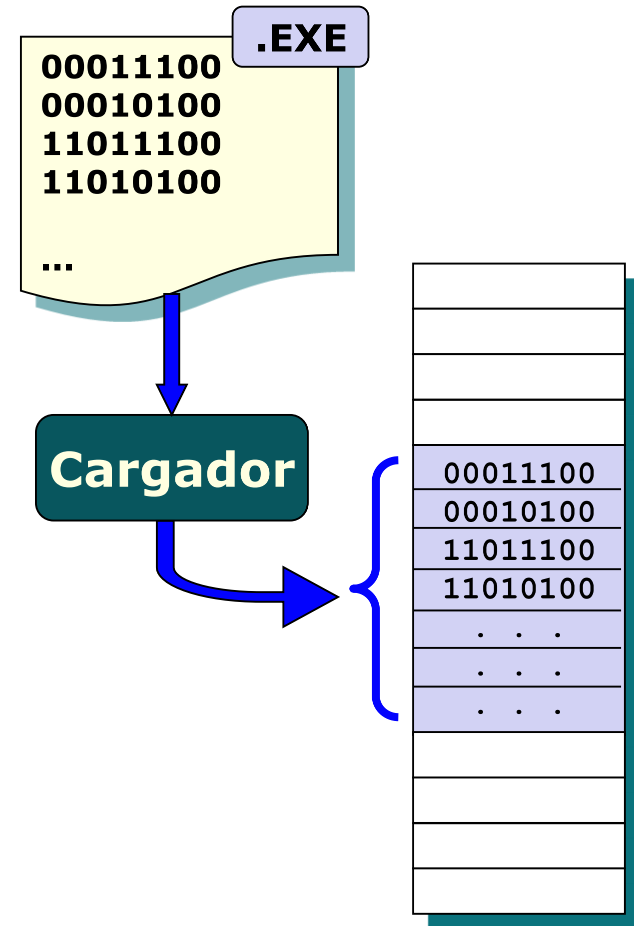
.LIB
```
_printf:
11011100
11010100
...
extern _main
exports _printf
...
```

Enlazador

.EXE
```
00011100
00010100
11011100
11010100

...
```

Ejecución del programa

- Ejecución de un programa:
  - El programa Cargador (Loader) vuelca en memoria el archivo ejecutable.
  - El cargador pone en el PC (Contador de Programa) la dirección de la 1ª instrucción del programa (el resto se ejecutan secuencialmente).
  - O bien cargar en memoria y ejecutar paso a paso con un programa depurador.



Compilación separada

- También es posible escribir el código en varios archivos de código fuente, compilar **por separado**, y enlazar conjuntamente.

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"     // Inclusión archivo de cabecera,
                                       // con < > para cabeceras estándar

#define IDENTIF texto                  // Definición de ctes. y macros

#pragma …                              // Indicaciones al compilador

#ifxxx …                               // Directivas de …
#else                                  // … compilación …
#endif                                 // … condicional

#error                                 // Genera error de compilación
```

El preprocesador

- Las directivas del preprocesador actúan sobre el texto que va a ser compilado, no en tiempo de ejecución.
- Ejemplo:

Preprocesador y compilación separada

- Cuando se realiza compilación separada, se utiliza la directiva **#include** para que todos los módulos utilicen las mismas definiciones (prototipos de función, constantes, etc).

uno.c
```
#include "dos.h"

main()
{
    float x=PI/4;
    if (FnDos(x,...))
        ...
}
```

Pre pro cesador

```
int FnDos(...);

main()
{
    float x=3.1416/4;
    if (FnDos(x,...))
        ...
}
```

Com pila dor

uno.obj
```
00010100111
11010010011
01001010111
...
```

dos.h
```
#define PI 3.1416
int FnDos(...);
```

dos.c
```
#include "dos.h"

int FnDos(...)
{
    if (x>PI)
        return 1;
    return 0;
}
```

Pre pro cesador

```
int FnDos(...);

int FnDos(...)
{
    if (x>3.1416)
        return 1;
    return 0;
}
```

Com pila dor

dos.obj
```
11010010011
00010100111
01001010111
...
```

Preprocesador y compilación separada

- Diferencias fundamentales entre archivo de código fuente (.c) y archivo de cabecera (.h):
  - Sólo los .c son compilados para generar .obj
  - Los .h sólo sirven para ser incluidos en uno o varios .c
  - Los .c llevan el código ejecutable (funciones, sentencias, expresiones, …).
  - Los .h sólo llevan declaraciones que sean necesarias para los .c que los incluyan:
    - Prototipos de funciones (cabecera terminada en ;)
    - Constantes
    - Nuevos tipos (struct, enum, typedef, …)
    - Otros #include necesarios
  - Es práctica habitual hacer un .h para cada módulo .c, que contenga las declaraciones necesarias (y sólo esas) para utilizar las funciones de ese módulo.

El preprocesador

- Definición de constantes y macros
  - Sin parámetros:

    #define N 50

    Todas las apariciones del identificador N serán sustituidas por el valor 50 antes de compilar
  - 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 comprobación 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:

miprog.c

```
#include "matrices.h"
#include "determinante.h"

. . .
```

Error compilación: Declaración doble de struct matriz

matrices.h

```
struct matriz
{
   . . .
} ;

. . .
```

determinante.h

```
#include "matrices.h"

. . .
```

miprog.c

```
#include "matrices.h"
#include "determinante.h"

. . .
```

matrices.h

```
#ifndef _INC_MATRICES_H
#define _INC_MATRICES_H
struct matriz
{
   . . .
} ;
. . .
#endif
```

determinante.h

```
#ifndef _INC_DETERM_H
#define _INC_DETERM_H
#include "matrices.h"

. . .

#endif
```

Ejercicios propuestos

1) Realizar un programa que calcule la desviación típica de los datos de una tabla mediante dos módulos de código fuente (principal.c y funciones.c) y un archivo de cabecera (funciones.h).
2) Realizar mediante una macro el cálculo de la máscara binaria dado el bit de comienzo y el nº de bits.
3) Realizar un programa que, en un bucle infinito, pida el primer valor de una tabla de float, y desplace los datos de la tabla añadiendo el nuevo valor al primer elemento. El programa debe permitir visualizar o no el contenido de la tabla en cada pasada en función de una opción de compilación (#define, #ifdef).

Universidad de Oviedo

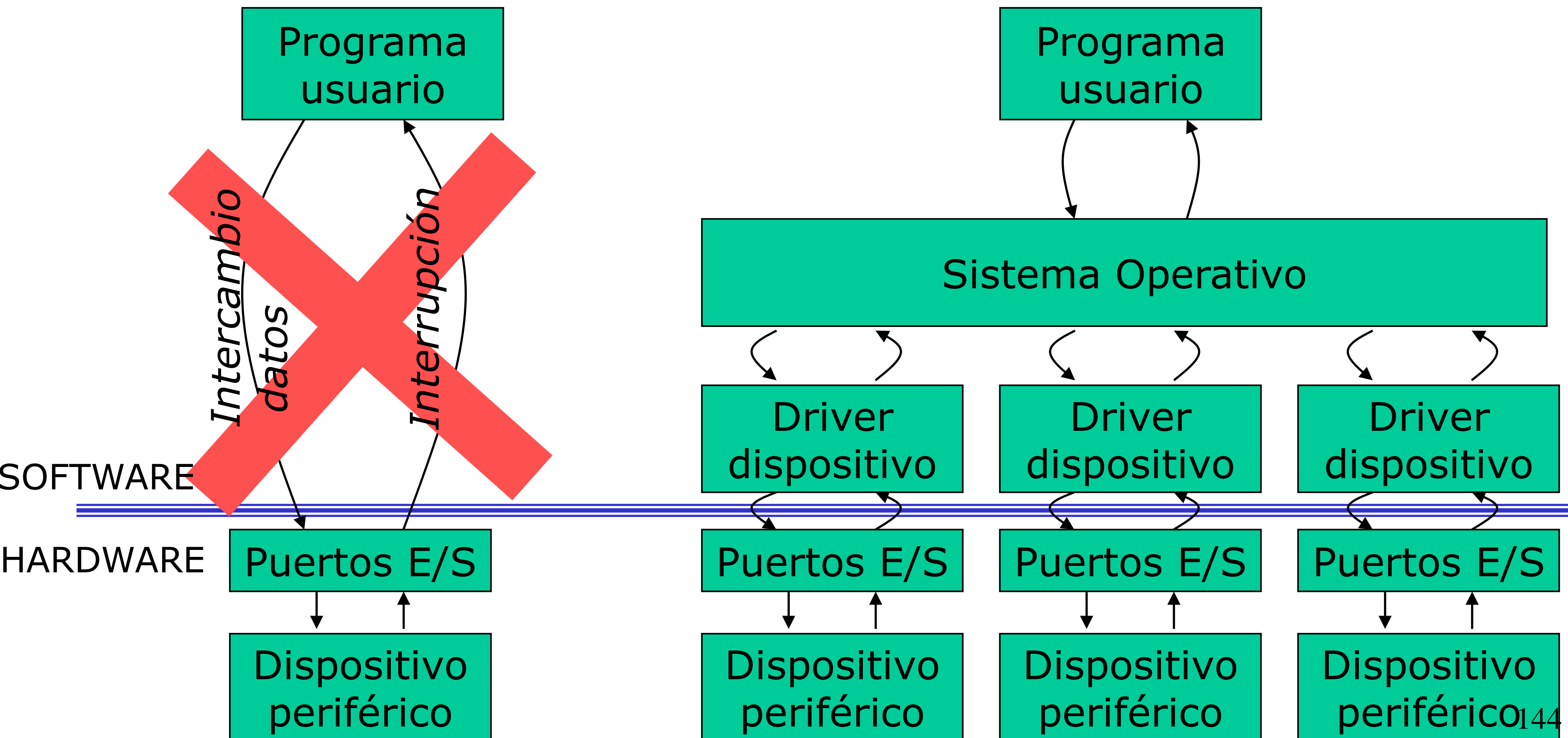
INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

Indice

- Introducción al computador y el lenguaje C
- Datos, expresiones y algoritmos
- Funciones
- Tablas y punteros
- Cadenas de caracteres
- Operaciones con valores binarios
- Preprocesador y compilación separada
- **E/S en archivos y dispositivos**
- Criterios de buena programación

Funciones de E/S por stream

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



Universidad de Oviedo

E/S por stream

Funcionamiento de la E/S por stream

PROGRAMA	S.O.	DRIVER	DISPOSITIVO
Solicitud acceso dispositivo →	Comprueba si existe y está libre →	Inicializa →	Acciones al acceder a puertos
	← Asigna identificador	← Compone Estado	←
Id. de stream ←			
Solicitud lectura id. →	Busca driver asignado →	Comprueba Puertos y estado →	Acciones al acceder a puertos
	← Compone resultado	¿Devolver resultado?	←
Datos leídos ←		↑ Lee puertos y compone estado	← Genera interrupción

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()

Ejercicios propuestos

1) Realizar un programa que lea los datos de una matriz de reales de un archivo de texto, supuesto el siguiente formato:

matriz.txt

n m
$x_{0,0}$ $x_{0,1}$ ... $x_{0,m-1}$
$x_{1,0}$ $x_{1,1}$ ... $x_{1,m-1}$
...
$x_{n-1,0}$ $x_{n-1,1}$ ... $x_{n-1,m-1}$

2) Realizar un programa que lea líneas de un archivo de texto y calcule el nº de líneas que contienen un texto determinado.
3) Realizar un programa que cree lea un archivo de texto y cree un segundo archivo sustituyendo todas las apariciones del texto “<x>” por el valor de la variable entera x (solicitada por teclado).
4) Leer de un archivo binario tantos datos tipo float como sea posible en un vector columna, y escribir un archivo de texto en el formato del ejercicio 1.
5) Realizar un programa similar al ejercicio 1, pero excluyendo del formato del archivo la línea con los valores n,m.

Indice

- Introducción al computador y el lenguaje C
- Datos, expresiones y algoritmos
- Funciones
- Tablas y punteros
- Cadenas de caracteres
- Operaciones con valores binarios
- Preprocesador y compilación separada
- E/S en archivos y dispositivos
- **Criterios de buena programación**

Criterios de buena programación

- El presente curso es un resumen: se debe consultar la documentación y/o libros/tutoriales más detallados.
- Algunas "omisiones" (I):
  - **Tipos de datos:**
    - Existen diversos tipos de datos enteros (long, int, short, char) según el nº de bits utilizados, en formato signed (por defecto) o unsigned. (enlace)
    - El lenguaje C no comprueba la salida de rango en operaciones de enteros: es responsabilidad del programador, y puede ser importante. (enlace)
    - Para números reales se puede usar float, double y long double. (enlace)
    - Se pueden utilizar reales en coma fija para acelerar la ejecución en procesadores sin unidad de coma flotante. (enlace)
    - La variables cuyo tipo de datos necesita más de un byte se almacenan en direcciones consecutivas, en formato little-endian o big-endian. (enlace)
    - El operador sizeof() devuelve el tamaño en bytes de un tipo de datos o variable. (enlace)
    - Se pueden "crear" nuevos tipos de datos con struct, union, enum y typedef. (enlace)

Criterios de buena programación

- Algunas “omisiones” (II):
  - **Variables:**
    - Además de las variables locales y parámetros, se pueden utilizar variables globales y estáticas. (enlace)
    - Se pueden utilizar arrays multidimensionales, con asignación de memoria estática o dinámica, aunque no se recomienda como norma general. (enlace)
  - **Operadores y expresiones**:
    - Operadores con asignación: += -= *= ++ -- (enlace)
    - Operador alternativa ? : (enlace)
    - Operador de separación , (enlace)
    - Seguir correctamente las reglas de evaluación de expresiones. (enlace)

Criterios de buena programación

- Algunas “omisiones” (III):
  - **Funciones:**
    - La función main() puede tener parámetros y devolver un resultado, que permiten configurar la ejecución desde el programa cargador. (enlace)
    - Existen punteros a función, que permiten llamar a funciones distintas en tiempo de ejecución. (enlace)
    - Se dispone de funciones de librería estándar para las operaciones más habituales (enlace) :
      - Matemáticas (incluir <math.h>)
      - Cadena de caracteres (incluir <string.h>)
      - Asignación dinámica de memoria (incluir <malloc.h>)
      - E/S por stream (incluir <stdio.h>)
      - Conversión de datos
      - Caracteres
      - Manipulación de memoria
      - Sistema y entorno
      - Fecha y hora
      - Búsqueda y ordenación
      - Generación de números aleatorios

Criterios de buena programación

- Al escribir el programa:
  - Realizar funciones y probar por separado, antes de incluirlas en el programa definitivo.
  - Las funciones deben ser lo más genéricas posible, dependiendo únicamente de sus entradas para producir los resultados:
    - No incluir printf() o scanf() habitualmente en funciones.
    - Las funciones deben devolver códigos de error si no pueden hacer su trabajo con los parámetros recibidos.
    - No usar en lo posible variables globales en funciones.
  - Agrupar funciones similares en un módulo fuente, y usar compilación separada.
  - No intentar optimizar el código a la primera.
  - Usar #define para constantes.
  - Añadir comentarios en el código, especialmente en cabeceras de función y código no auto-explicativo.

Criterios de buena programación

- Al ejecutar y probar el programa:
  - No hay que esperar que todo funcione a la perfección a la primera: un código que compila bien no significa que ejecute lo que se espera.
  - Probar las funciones previamente por separado: facilita la localización de errores.
  - Probar en modo depuración: no añadir printf() para depurar.
  - Comprobar la respuesta ante situaciones anómalas que se pueden producir con cierta probabilidad.