Tema 1: Introducción a la Computación

Contenido

- Información. Soportes. Codificación.
- Estructura básica y funcionamiento de un computador digital
- Concepto de algoritmo. El proceso de la programación.
- Lenguajes de programación
- El lenguaje C

1. Información. Soportes. Codificación.

- Información: acción y efecto de informar, de dar noticia de algo. Se produce un tratamiento.
- Se distingue: la información inicial (dato) y el resultado del tratamiento de los datos (información).

- Informática: término que aparece en los diccionarios hacia 1972, ligado al término información.

> *"Conjunto de conocimientos científicos y técnicas que hacen posible el tratamiento automático de la información por medio de ordenadores"*

1. Información. Soportes. Codificación.

El tratamiento consta de 3 etapas básicas:

- Entrada: recogida de datos.
- Proceso: tratamiento de los datos.
- Salida: obtención de la información resultante.

1. Información. Soportes. Codificación.

- El tratamiento de la información puede automatizarse mediante el empleo del ordenador.
- El comportamiento y el pensamiento humano se da en secuencias lógicas.
- Esta ordenación se adquiere a través de un proceso que podemos llamar programación.
- Los portadores de la información que recibimos a través de nuestros sentidos son fenómenos energéticos, ondas lumínicas, mecanismos químicos, ...

1. Información. Soportes. Codificación.

- Los mensajes recibidos suelen tener formato de información analógica.
- Por razones tecnológicas, este tipo de formato no es fácil de procesar salvo por los seres humanos.
- Esto llevó a la transformación de la información a digital.
- La información discretizada, se denomina digital.

1. Información. Soportes. Codificación.

- Los dispositivos de tratamiento de la información actuales son binarios. Tienen la capacidad de permanecer en solo dos estados.

- Generalmente se denominan 0 y 1 y corresponden a los dos dígitos de representación del sistema binario.

- Bit: uno cualquiera de dichos valores. Representa la unidad mínima de información.

- ¿Cuántos mensajes distintos son codificables mediante una secuencia de 8 bits?

1. Información. Soportes. Codificación.

- Byte: una secuencia de 8 bits
  - 1 Kb = 2^{10} = 1024 bytes
  - 1 Mb = 1024 Kb
  - 1 Gb = 1024 Mb
  - 1 Tb = 1024 Gb

- Una secuencia de bits suficientemente larga denota cualquier número natural.

- Mediante secuencias de bits podemos representar cualquier elemento de un conjunto enumerable.

1. Información. Soportes. Codificación.

- No podemos representar de forma precisa todos los elementos de conjuntos no enumerables.

- Codificación: Representación de símbolos de un alfabeto mediante los de otro.

- Sistema de codificación: tablas de correspondencia entre caracteres alfanuméricos y cadenas de dígitos binarios.

- La elección depende del tamaño del conjunto de caracteres y de consideraciones estadísticas: BCD, ASCII, EBCDIC, ...

2. Estructura básica y funcionamiento de un computador digital

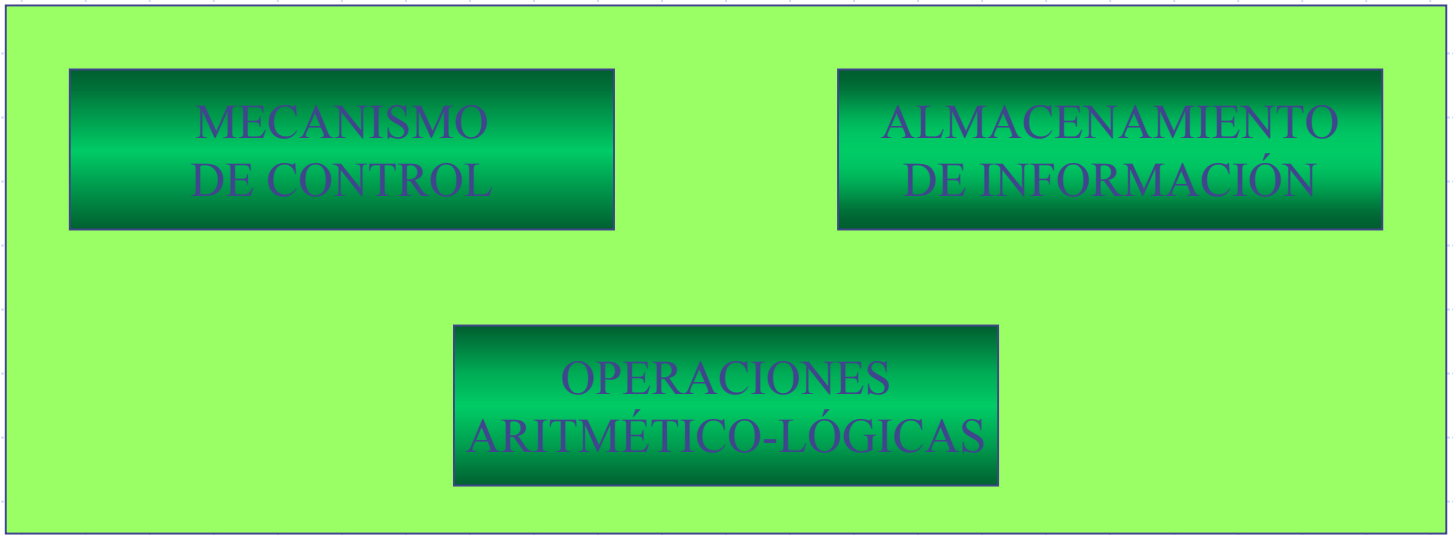
- Necesitamos un lugar donde almacenar los datos de entrada.

- El tratamiento de los datos precisa realizar una serie de acciones (instrucciones) sobre ellos en un orden determinado (programa) que también se debe almacenar.

- Por tanto, dos tipos de información a almacenar en un ordenador: datos de entrada y el programa que los maneja.

2. Estructura básica y funcionamiento de un computador digital

- Una vez almacenados, se deben tratar de forma adecuada. Necesitamos un mecanismo de control.
- Un primer esquema de ordenador:



2. Estructura básica y funcionamiento de un computador digital

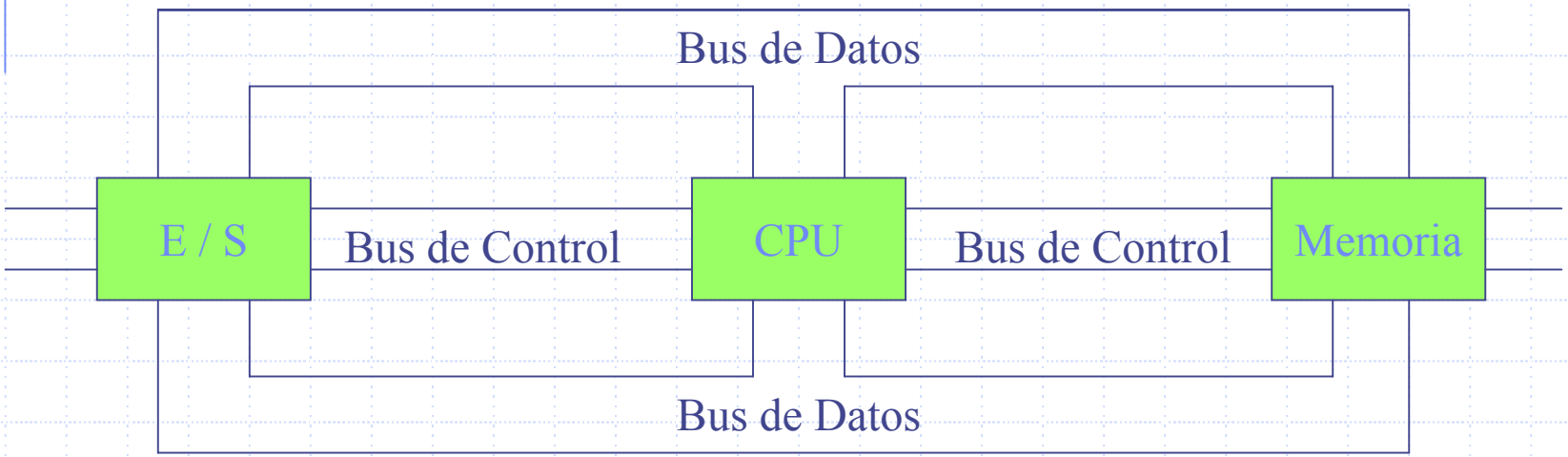
- A partir del esquema, un ordenador es una máquina compuesta por los siguientes dispositivos físicos:
  - **Memoria**: zona de almacenamiento de la información.
  - **Unidad aritmético-lógica**: parte encargada de realizar las operaciones.
  - **Unidad de control**: dirige todas las operaciones del ordenador.

2. Estructura básica y funcionamiento de un computador digital

- Un ordenador consta de 3 módulos principales:
  - Memoria Central
  - Unidad Central de Proceso (CPU):
    - La Unidad de Control (CU)
    - La Unidad Aritmético-Lógica (ALU)
  - Unidades de entrada/salida

2. Estructura básica y funcionamiento de un computador digital

- Gráficamente:



3. Concepto de algoritmo. El proceso de la programación.

- Algoritmo: lista o secuencia de instrucciones que especifican la secuencia de operaciones elementales necesarias para resolver en un tiempo finito cualquier problema de un tipo específico dado.

- Ejemplo: las reglas para efectuar las 4 operaciones con números escritos en forma decimal (siglo IX).

- El primer algoritmo conocido fue debido a Euclides (300 AC).

3. Concepto de algoritmo. El proceso de la programación.

- Ejemplo: algoritmo para obtener el *máximo común divisor* de dos números enteros:
  - 1) Sean a y b dos números enteros dados
  - 2) Comparar los dos números de forma que:
  - 3) Si son iguales, es el resultado. Finalizar.
  - 4) Si a < b, intercambiar a y b.
  - 5) Restar el segundo del primero, sustituyéndolos por el sustraendo y el residuo, respectivamente. Volver al paso 2).

3. Concepto de algoritmo. El proceso de la programación.

- Todo algoritmo debe cumplir estas condiciones:
  - 1) **Finitud**: todo algoritmo debe acabar siempre tras un número finito de pasos.
  - 2) **Definibilidad**: toda regla debe definir sin ambigüedad la acción a desarrollar.
  - 3) **Generalidad**: un algoritmo debe resolver toda una clase de problemas y no un problema aislado particular.
  - 4) **Eficacia**: se debe pretender que la ejecución de un algoritmo resuelva el problema de forma rápida y eficiente.

3. Concepto de algoritmo. El proceso de la programación.

- Un buen algoritmo debe ser:
  - finito, no ambiguo, general y eficaz.
- Técnicas de representación y diseño de algoritmos:
  - Pseudocódigo.
  - Diagramas de flujo.
- En Matemáticas, una clase de problemas está resuelta cuando se encuentra un algoritmo que la resuelve. Si existe, la clase se denomina decidible o computable.
- Existen muchas clases de problemas no decidibles.

3. Concepto de algoritmo. El proceso de la programación.

- Dado un problema:
  - **Fase de resolución**: determinar el algoritmo que lo resuelve.
  - **Fase de implementación del algoritmo**: traducción del mismo a un lenguaje determinado (codificación).

- El término programación se asocia comúnmente con el de codificación y es un error.

3. Concepto de algoritmo. El proceso de la programación.

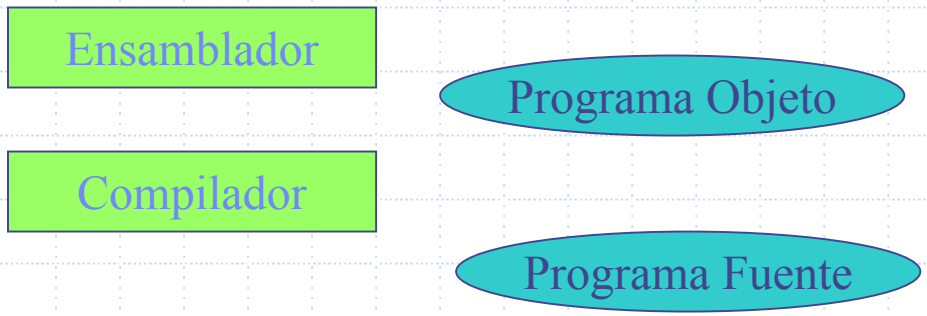
- El desarrollo de un programa pasa por las fases de:
  - **Análisis**: definición del problema.
  - **Algoritmo**: desarrollo de la secuencia lógica de pasos para la resolución del problema.
  - **Codificación**: conversión del algoritmo en un programa escrito en un lenguaje de programación.
  - **Ejecución y prueba**.
  - **Mantenimiento**.
- Estas 5 etapas constituyen el ciclo de vida de un proyecto informático.

4. Lenguajes de programación

- Lenguaje de programación: conjunto de reglas, símbolos y palabras especiales usadas para construir un programa.

- Los lenguajes de programación se clasifican de acuerdo a lo próximos que estén a la máquina o al hombre:

  - Lenguaje máquina
  - Lenguaje ensamblador
  - Lenguaje de alto nivel



5. El lenguaje C

- Ha alcanzado una gran popularidad debido a que está ligado a otro fenómeno de la informática de los años 80: el sistema UNIX.

- Tanto el sistema UNIX como el lenguaje empleado para su implementación fueron creados por los Laboratorios Bell: Ken Thompson y Dennis Ritchie.

- C es un lenguaje estructurado de alto nivel de propósito general pero también es relativamente cercano a la arquitectura real de las máquinas.

5. El lenguaje C

- Ejemplo: suma de dos números enteros en C.

```
#include <stdio.h>
int main() {
        int a, b, c  ;

        scanf("%d %d", &a, &b) ;
        c = a + b  ;
        printf("La suma de %d y %d es %d\n", a, b, c) ;
        return ;
}
```

Tema 2: Elementos básicos de un programa

Contenido

- Estructura de un programa en C
- Variables
- Tipos de datos
- Expresiones
- La instrucción de asignación
- Constantes
- Entrada y Salida en C

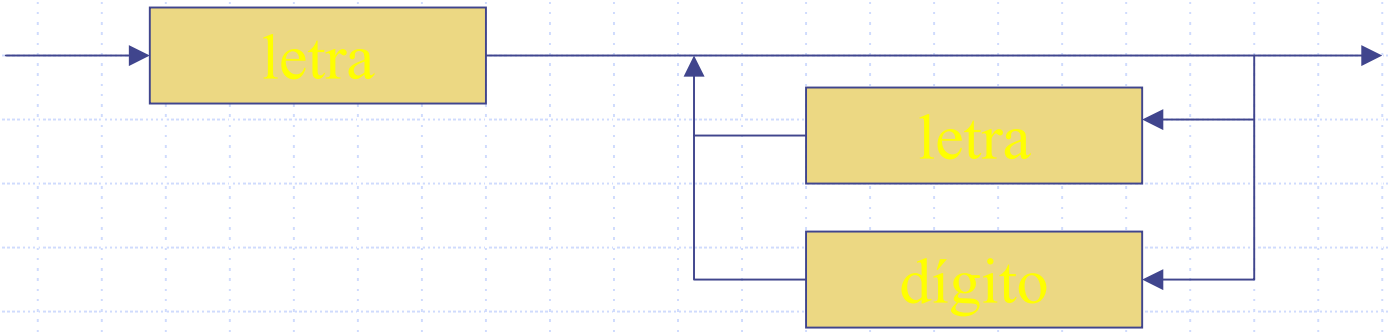
1. Estructura de un programa en C

- La estructura principal de un programa es:

```
main ( )
{
        instrucción_1 ;
        instrucción_2 ;
        …
        instrucción_n ;
}
```

2. Variables

- Para denominar a los elementos que intervienen en un programa se usan identificadores.
- Algunos están definidos en el lenguaje y tienen un uso predeterminado: palabras reservadas.
- Identificadores: nombres que están asociados con procesos y objetos y se usan para referenciarlos.



2. Variables

- La memoria se divide en un gran número de posiciones separadas que almacenan parte de los datos.
- Podríamos referirnos a las posiciones de memoria por sus direcciones, como en código máquina.
- Los lenguajes de alto nivel ofrecen una alternativa: usar identificadores como los nombres de las posiciones de memoria.
- Se debe usar un nombre para representar una sola cosa.

2. Variables

- Un programa opera con datos. Estos datos pueden ser de uno de los tipos simples: **`int`**, **`float`**, **`double`**, **`char`**.

- Los datos se almacenan en memoria.

- Durante la ejecución del programa diferentes valores pueden almacenarse en la misma posición de memoria en tiempos distintos.

- Variable: estrictamente hablando, la posición de memoria es la variable y su contenido es el valor de la variable.

2. Variables

- Una variable se caracteriza por un nombre y cuatro atributos básicos: ámbito, tiempo de vida, valor y tipo.
  - **Nombre**: se usa para identificar y hacer referencia a la variable.
  - **Ámbito**: es el conjunto de sentencias del programa en las que la variable es conocida y por tanto manipulable.
  - **Tiempo de vida**: el intervalo de tiempo en el que un área de memoria está ligada a la variable. La acción de conseguir un área de memoria se denomina asignación de memoria y puede ser estática (antes de la ejecución del programa) o dinámica (durante la ejecución).

2. Variables

- **Valor**: se representa en forma codificada en el área de memoria ligada a la variable y se interpreta de acuerdo con el tipo de la variable. La ligadura entre una variable y el valor contenido en su área de memoria es dinámica pero puede ser fija y resulta una constante.

- **Tipo**: especificación de la clase de valores que se pueden asociar a la variable junto con las operaciones que se pueden usar para crear, acceder y modificar tales valores.

2. Variables

- Antes de escribir las instrucciones de un programa es necesario declarar las variables que se van a utilizar.

- Es decir, reservar celdas de memoria para almacenar los datos durante la ejecución de un programa, dándoles un nombre y un tipo.

- Durante el programa, se hace referencia al dato de una variable mediante el nombre dado a la variable y con el tipo de dato asignado se establece la longitud y las operaciones que se pueden realizar con el dato.

2. Variables

- Sintaxis:

```
tipo_de_dato nombre_de_variable ;
```

- Existen dos lugares donde se pueden declarar las variables:
  - Antes de la función principal `main`: variables globales.
  - Dentro de la función `main`: variables locales.

- En cualquier lugar del programa se pueden poner comentarios, textos que no forman parte del código:

```
/* el texto que queramos */
```

3. Tipos de datos

- A nivel de *lenguaje máquina*: datos como cadenas de bits manipulados mediante operaciones lógicas, aritméticas, de desplazamiento, ...

- En los *lenguajes de alto nivel*: abstracción en los datos. La información almacenada en la memoria se ve como un valor entero, real, lógico, ...

- Estos valores constituyen los tipos predefinidos, que identifican el comportamiento abstracto de un conjunto de objetos dotado de un conjunto de operaciones.

3. Tipos de datos

- Los datos que maneja un programa se deben clasificar previamente en tipos de datos.

- Mediante la asignación de un tipo de dato a una variable se realizan dos tareas importantes:
  - Se establece la longitud de la celda de memoria destinada a almacenar el dato
  - Se establece el conjunto de operaciones que se pueden realizar sobre dicho dato.

- Un traductor para un lenguaje de programación proyecta esta visión abstracta en una implementación concreta.

3. Tipos de datos

- Un dato es toda información con la que opera el computador.

- En C cada elemento de datos debe ser de un tipo específico.

- El tipo de datos determina cómo se representan los elementos de datos en el computador y qué tipo de procesamiento se puede efectuar sobre ellos.

3. Tipos de datos

- Existen dos grupos de tipos de datos:
  - Simples (sin estructura)
  - Compuestos (estructurados)

- Los datos estructurados son colecciones de datos simples con relaciones definidas entre ellos.

- Los tipos de datos simples en C son:

  `int`, **`char`**, **`float/double`** y `punteros`.

3.1 Enteros

- El tipo entero es un subconjunto finito de los números enteros: no tienen componentes fraccionarios y pueden ser positivos o negativos.

- Tamaño: los enteros mínimo y máximo representables en un ordenador de 16 bits son -32768 y 32767.

- Los números fuera de este rango no se pueden representar como tipo de datos entero: *overflow*.

3.1 Enteros

- Operaciones:
  - Aritméticas:

    +, **-**, *******, **/** y **%**.

    donde **/** es la división entera y **%** es el módulo (resto de la división entera)
  - De comparación:

    >, **>=**, **<**, **<=**, **==** y **!=**.

- El resultado de comparar dos datos de tipo entero da otro dato de tipo entero, 0 (para falso) ó 1 (para cierto).

3.1 Enteros

- Declaración:

```
int variable1, variable2, … ;
```

- Este tipo de dato tiene una longitud de 16 bits.
- Es posible aumentar la representación en bits a 32 bits de una variable entera mediante el tipo `long int`.
- Existen varios calificadores asociados a los enteros:

```
short int
long int
unsigned int
```

3.2 Reales

- El tipo real consiste en un subconjunto de los números reales.
- Tienen un punto decimal y pueden ser positivos o negativos.
- En la memoria se representan como números en coma flotante, constituidos por un exponente y su signo, y por la mantisa y su signo: 0.314e-7.
- El rango de representación es, aproximadamente, de 10^{-38} a 10^{+38}.

3.2 Reales

- Operaciones:
  - Aritméticas:

    +, **-**, ***** y **/**.

    donde **/** es la división real. El programa la distingue de la división entera comprobando el tipo de los operandos.
  - De comparación:

    >, **>=**, **<**, **<=**, **==** y **!=**.
- El resultado de comparar dos datos de tipo real da un dato de tipo real, 0.0 (para falso) ó 1.0 (para cierto).

3.2 Reales

- Declaración de variables de tipo real:

```
float variable1, variable2, … ;
```

- El tipo real se declara con la palabra reservada `float` y tiene una longitud de 32 bits.

- Es posible disponer de una representación de 64 bits mediante el tipo `double`, que se comporta de la misma forma y usa las mismas operaciones que el tipo `float`.

3.3 Caracteres

- El tipo carácter es el conjunto finito y ordenado de caracteres que el ordenador reconoce.
- En general, la mayoría de ordenadores reconoce:
  - Caracteres alfabéticos: a, …, z, A, …, Z.
  - Caracteres numéricos: 0, …, 9.
  - Caracteres especiales: +, -, *, <, …
- Los caracteres se representan mediante 1 byte.
- La interpretación que hace un programa C se realiza siguiendo unas tablas de conversión: código ASCII.

3.3 Caracteres

- Operaciones: como la representación interna de los caracteres es mediante cadenas de bits, en C es posible realizar operaciones aritméticas y de comparación.

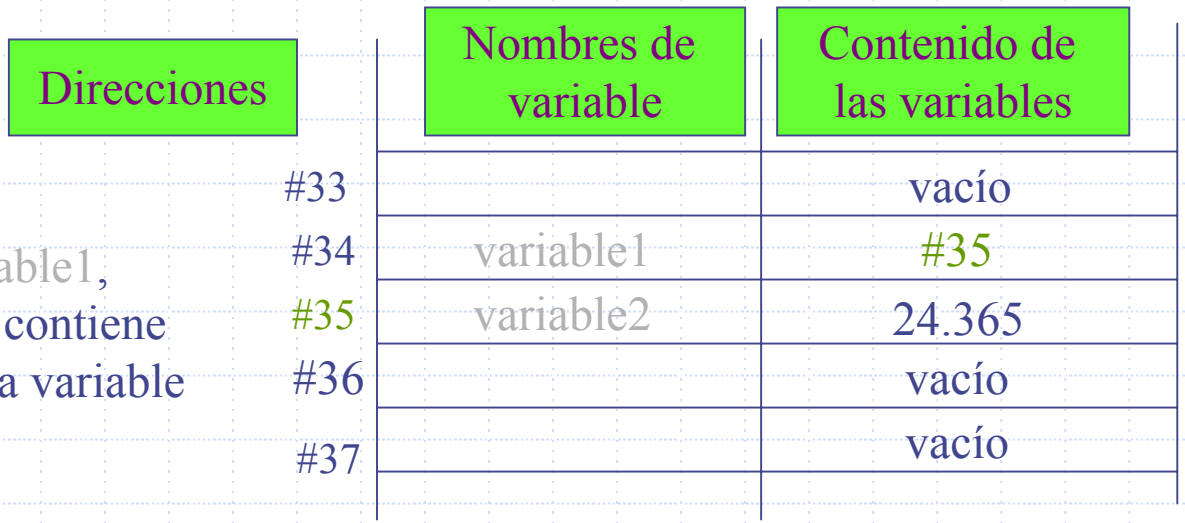
- Ejemplo:
  - `'A'<'B'` da como resultado 1.
  - la suma `'B'+1` da como resultado la cadena de bits 01000011 que corresponde al carácter 'C'.

- Declaración:

```
char variable1, variable2, ... ;
```

3.4 Punteros

- Mediante este tipo de datos, en C es posible almacenar direcciones de celdas de memoria en variables,
  - es decir, es posible que en una variable declarada de tipo puntero se almacene la dirección de otra variable de otro tipo de dato cualquiera.

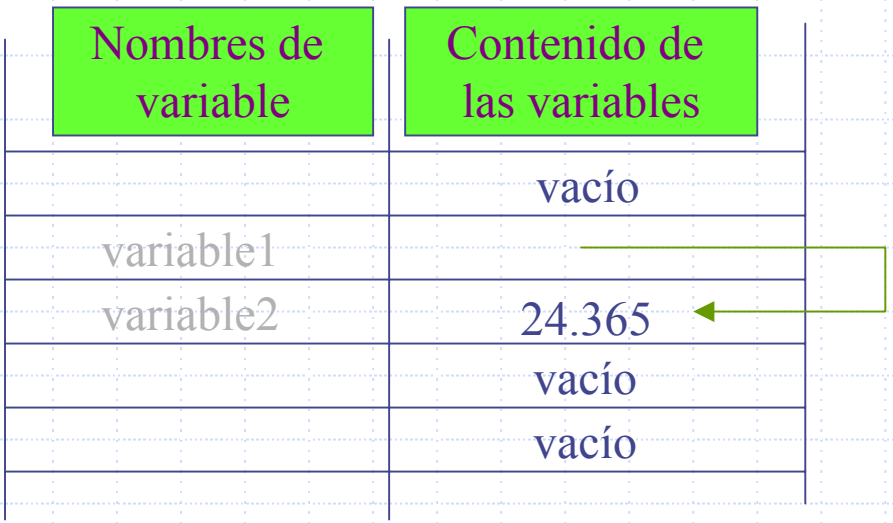


3.4 Punteros

- En la práctica, no se suelen manejar las direcciones concretas.

- Es más útil decir que la variable1 contiene la dirección de la variable2 sin importar cuál sea su dirección exacta.

La variable variable1, de tipo puntero, apunta a la variable variable2.



3.4 Punteros

- Declaración:

```
tipo_de_dato *nombre_del_puntero ;
```

- Por ejemplo:

```
int *punt1 ;
```

indica que la variable de tipo puntero `punt1` contendrá direcciones de memoria que apuntarán a variables de tipo entero `int`.

3.4 Punteros

- Todas las variables de tipo puntero tienen la misma longitud, la de una dirección de memoria, pero hay que especificar el tipo de dato que contendrá la variable.

- Una vez declarado, un puntero sólo puede emplearse para apuntar a variables del mismo tipo.

- El manejo del tipo puntero es muy común en C.

- Existen conexiones entre el tipo puntero y los vectores, el paso de parámetros a funciones, ...

4. Expresiones

- Las expresiones son combinaciones de constantes, variables, símbolos de operación, paréntesis y nombres de funciones especiales.

- Ejemplo:

```
a - (b+3) * c
```

- Las expresiones en C pueden ser:
  - Aritméticas (resultado **`int`** o **`float`**)
  - Relacionales (resultado **`int`**)
  - Lógicas (resultado **`int`**)

4. Expresiones

- Las expresiones aritméticas usan
  - variables y constantes reales o enteras, y
  - los operadores **+**, **-**, *****, **/** y **%**.

- La división devuelve un valor real siempre que al menos uno de los operandos sea real.

- Ejemplo:
  - No es lo mismo **`3/2`** que **`3.0/2.0`**

4. Expresiones

- Para evaluar expresiones compuestas existen reglas de prioridad o precedencia:
  - Las operaciones encerradas entre paréntesis se evalúan primero.
  - Los operadores cumplen el siguiente orden de precedencia:

    1º) /, *, %

    2º) +, -
  - Dentro del mismo orden, se sigue la asociatividad de izquierda a derecha.

4. Expresiones

- Ejemplo: ¿Cuál es el resultado?

```
main() {
        int a, b, c ;
        a = 3 ;
        b = 2 ;
        c = 2 + 3 / 2 ;
}
```

4. Expresiones

- Operadores reducidos: permiten expresar de una forma más concisa una operación.

- Ejemplo:

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

- Casos especiales:

```
a = a + 1 ;
a += 1 ;    o bien    a++ ;
a -= 1 ;    o bien    a-- ;
```

4. Expresiones

- Expresiones relacionales: dependiendo del tipo de variable (`int` o `float`) al que se da el resultado el valor será las constantes enteras `1` ó `0` o las constantes reales `1.0` ó `0.0`.

4. Expresiones

- Expresiones lógicas: Son expresiones que se forman combinando constantes y variables numéricas y de carácter utilizando operadores lógicos.

Operador	Significado	Operador en C
AND	Y	&&
OR	O	||
NOT	NO	!

4. Expresiones

- Las conectivas lógicas son:

dat1	dat2	!dat1	dat1&&dat2	dat1||dat2
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

5. La instrucción de asignación

- Es la instrucción básica y sirve para darle valores a una variable, es decir, para almacenar datos en la celda de memoria que representa una variable declarada.

- El formato general es:

```
nombre_variable = expresión ;
```

- La acción de asignar es destructiva.

5. La instrucción de asignación

- Ejemplo: Intercambiar el contenido de dos variables.

(solución incorrecta)

```
main() {
        int A, B ;
        A = 5 ;
        B = 10 ;
        B = A ;
        A = B ;
}
```

5. La instrucción de asignación

- Ejemplo: Intercambiar el contenido de dos variables.

(solución correcta)

```
main() {
        int A, B, aux ;
        A = 5 ;
        B = 10 ;
        aux = A ;
        A = B ;
        B = aux ;
}
```

6. Constantes

- En la mayor parte de los lenguajes de programación los objetos básicos a usar son
  - las variables y las constantes.

- En el ejemplo anterior: `A = 5 ;` tenemos la variable `A`, el operador de asignación `=` y un valor constante `5`.

- Los programas contienen ciertos valores que no deben cambiar durante la ejecución: las constantes.

- Las constantes pueden ser numéricas (enteras o reales) y de tipo carácter (encerrado entre comillas simples).

6. Constantes

- Existen caracteres especiales que se representan con la barra \ y una letra:
  - el retorno de carro `'\n'`, el tabulador `'\t'`, ...
- Las variables no toman por defecto un valor inicial.
- Una posibilidad que permite C es la de asignar un valor constante en la declaración de una variable:

```
int A = 0, B = 10 ;
```

6. Constantes

- A veces puede aparecer la misma constante en un programa en varios sitios.

- Para facilitar el cambio de su valor, existe la posibilidad de usar constantes simbólicas.

- Las constantes simbólicas son identificadores que se emplean en lugar de constantes en sí.

- Sintaxis:

```
#define VALOR 3
```

6. Constantes

- Una utilidad:
  - usar constantes simbólicas para representar los valores cierto y falso en lugar de 0 y 1.

```
#define CIERTO 1
#define FALSO 0
```

- Estas líneas de programa no llevan `;`
- Son pseudoinstrucciones que sólo interpreta el precompilador.

6. Entrada y Salida en C

- Entrada de un programa: conjunto de instrucciones encargadas de leer datos de fuentes externas e introducirlos en variables.

- Fuentes externas: cualquier dispositivo periférico como dispositivo de entrada:
  - teclado, disco duro, ratón, ...

- Instrucción:

```
scanf("formato", lista_de_variables) ;
```

7. Entrada y Salida en C

- En **formato** se indica el tipo de dato de las variables que aparecen en `lista_de_variables`.

- Ejemplo:

```
scanf("%d", &a) ;
```

7. Entrada y Salida en C

- Ejemplo: suma de dos números enteros introducidos por teclado.

```
main() {
        int a, b, c ;
        scanf("%d%d", &a, &b) ;
        c = a + b ;
}
```

7. Entrada y Salida en C

- En el **`formato`** no deben existir espacios en blanco ni símbolos innecesarios.
- Formatos de tipo para **`scanf`**:

Formato del tipo	Descripción
`d`	números enteros decimales
`ld`	números enteros decimales largos
`u`	números enteros sin signo
`e`, `f`, `lf`, `g`	números reales
`c`	un carácter ASCII

7. Entrada y Salida en C

- Otra función usada para la introducción de datos es **getchar**, que permite leer de forma sencilla caracteres por teclado.

- Sintaxis:

```
variable = getchar() ;
```

- Ejemplo:

```
main() {
        char a ;
        a = getchar() ;
}
```

7. Entrada y Salida en C

- Las instrucciones de salida permiten presentar datos en pantalla.

- Sintaxis:

```
printf("formato", lista_de_argumentos) ;
```

- El campo **formato** tiene la misma sintaxis y significado que en **scanf**, pero aquí pueden aparecer más cosas.

- En **lista_de_argumentos** pueden aparecer también constantes y las variables no llevan delante **&**.

7. Entrada y Salida en C

- Ejemplo:

```
printf("La solución es %d\n", a) ;
```

da como salida, si `a` tiene almacenado el valor `25`:

```
La solución es 25
```

- Para controlar el formato de salida (parte entera y decimal) de un número real existen modificadores de formato:

```
printf("El número real: %2.3f\n",13.5e-1) ;
```

```
El número real: 1.350
```

6. Entrada y Salida en C

- También se puede especificar el número de posiciones para la salida de números enteros:

```
printf("%10d\n%10d\n", 25, 3600) ;
```

```
        25
      3600
```

- El teclado en C está asociado a un fichero, el fichero de entrada estándar (**stdin**) y la pantalla al fichero de salida estándar (**stdout**).

7. Entrada y Salida en C

- Las funciones **`scanf`** y **`printf`** acceden a estos ficheros, que están preparados por defecto para leer datos del teclado y escribirlos en pantalla.

- Para poder utilizar las funciones **`scanf`** y **`printf`** es necesario incorporar al programa la línea:

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

- Esta línea de código introduce la información que necesita el compilador para generar el código ejecutable asociado a la entrada/salida.

7. Entrada y Salida en C

- Ejemplo: suma de dos números enteros.

```
#include <stdio.h>
main() {
  int a,b,c ;
  printf("Dame dos números enteros: ") ;
  scanf("%d%d", &a, &b) ;
  c = a + b ;
  printf("La suma de %d y %d es %d\n", a, b, c) ;
}
```

7. Entrada y Salida en C

- Ejemplo: Calcular el consumo de gasolina (litros en 100 Km) de un coche en un recorrido determinado, conocidos los Km. circulados y el dinero gastado.

7. Entrada y Salida en C

```
#include <stdio.h>
#define LITRO 0.69
main() {
        float consumo, km, dinero, litros ;
        printf("Indica los Km. recorridos: ") ;
        scanf("%f", &km) ;
        printf("Indica el dinero gastado: ") ;
        scanf("%f", &dinero) ;
        litros = dinero / LITRO ;
        consumo = 100 * litros / km ;
        printf("El consumo es %5.2f\n", consumo) ;
}
```

Tema 3: Estructuras de Control

Contenido

- 1. Estructuras de selección:
  - Selección simple
  - Operador condicional
  - Selección compuesta
- 2. Estructuras de repetición:
  - while_do
  - do_while
  - for

1. Estructuras de Selección

- Hasta ahora las instrucciones de un programa tienen una ejecución secuencial ...

- Nos planteamos la posibilidad de modificar el orden de ejecución de las instrucciones de forma estructurada:
  - Haciendo que se ejecuten unas u otras en función de condiciones (Estructuras de Selección).
  - Haciendo que se repitan un cierto número de veces en función de una condición (Estructuras de Repetición).

1. Estructuras de Selección: Selección simple

- Se realiza un test lógico para llevar a cabo una de dos posibles acciones.
- Sintaxis:

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

```
if (expresión)
        sentencia1 ;
```

- La **expresión** debe ir entre paréntesis. La **sentencia1** se ejecuta si la **expresión** tiene un valor no nulo. En caso contrario se ejecuta la **sentencia2**.

1. Estructuras de Selección: Operador condicional

- Veamos una forma más breve de escribir sentencias condicionales sencillas.
- Por ejemplo, si tenemos:

```
if (estado == 'S')
        tasa = 0.2 * pago ;
else
        tasa = 0.14 * pago ;
```

- Este código es equivalente a:

```
tasa = (estado=='S') ? (0.2*pago) : (0.14*pago) ;
```

1. Estructuras de Selección: Selección simple

- Ejemplo:

Escribir en C un programa que pida al usuario el precio de un producto y si el artículo es o no de lujo (s/n), de forma que obtenga el precio real aplicándole un 25% de IVA.

1. Estructuras de Selección: Selección simple

- Ejemplo estructura anidada:

Escribir en C un programa que pida al usuario tres valores enteros y que devuelva por pantalla el valor máximo de los tres valores introducidos.

1. Estructuras de Selección: Selección múltiple

- Se selecciona un grupo de sentencias entre varios disponibles. Uso de la sentencia **`switch`**.

- Sintaxis:

```
switch (expresión) {
        case exp-const: sentencias ;
        case exp-const: sentencias ;
        default: sentencias ;
}
```

1. Estructuras de Selección: Selección múltiple

La **`expresión`** devuelve un valor entero (o carácter). Cada **`case`** se etiqueta con uno o más valores constantes enteros o expresiones constantes enteras.

Si un **`case`** coincide con el valor de la expresión, se ejecutan sus sentencias. El **`default`** se ejecuta si ninguno se satisface.

El uso de **`break`** (o **`return`**) es necesario para abandonar la sentencia **`switch`**. Si no se maneja con cuidado puede causar comportamientos raros.

1. Estructuras de Selección: Selección múltiple

```
switch (indicador) {
        case -1:
                y = fabs(x) ; x++ ; break ;
        case 0:
                y = sqrt(x) ; break ;
        case 1:
                y = x ; x-- ; break ;
        case 2:
        case 3:
                y = 2 * (x-1) ; break ;
        default:
                y = 0 ;
                break ;        /*No hace falta, pero pongámoslo */
}
```

2. Estructuras de Repetición

- Son estructuras de control cuyo objetivo es repetir un conjunto de instrucciones en función de unas condiciones.

- Hay tres tipos de estructuras de control:

  - `while`
  - `do_while`
  - `for`

2. Estructuras de Repetición: *while*

- La sintaxis general es:

```
while (expresión)
        proposicion/es ;
```

- Se evalúa la **`expresión`** y, si el resultado es diferente de cero (es cierta), se ejecuta la proposición (simple o compuesta).

- El ciclo continúa hasta que la **`expresión`** se hace cero (falsa).

2. Estructuras de Repetición: *do_while*

- La sintaxis general es:

```
do {
        proposicion/es ;
} while (expresión) ;
```

- Se ejecuta la proposición (simple o compuesta), después se evalúa la **expresión** y, si el resultado es diferente de cero (es cierta), se vuelve a ejecutar la proposición.
- El ciclo continúa hasta que la **expresión** se hace cero (falsa).

2. Estructuras de Repetición: *for*

- La sintaxis general es:

```
for (expr_1;expr_2;expr_3)
        sentencia ;
```

donde **expr1** es una expresión de asignación, **expr2** es una expresión lógica y **expr3** es una expresión unaria o una expresión de asignación.

- La sentencia **for** es equivalente a:

```
expr_1 ;

while (expr_2)
{
        sentencia
        expr_3 ;
}
```

Tema 4: Abstracción de Operaciones

Contenido

- 1. Introducción
- 2. Transferencia de la información
- 3. Paso de parámetros a una función
- 4. Objetos locales y globales
- 5. Recursividad
- 6. Ámbito de las variables

1. Introducción

- Supongamos que tenemos en un programa la necesidad de calcular las combinaciones de un número *n* sobre *m*. La solución puede ser:

1.- Introducción

```
#include <stdio.h>
main () {
        int n, m, i ;
        float nf, mf, nmf, result ;
        printf("Dame el valor de n y m: ") ;
        scanf("%d %d", &n, &m) ;
        nf  = 1 ;
        for (i=n; i>0; i--)
                nf *= i ;
        mf = 1 ;
        for (i=m; i>0; i--)
                mf *= i ;
        nmf = 1 ;
        for (i=n-m; i>0; i--)
                nmf *= i ;
        result = nf / (mf * nmf) ;
        printf("El factorial de %d es %f, el de %d es %f\n", n, nf, m, mf) ;
        printf("La combinatoria es: %f\n", result) ;
        return ;
}
```

1. Introducción

Construir programas *largos* y *monolíticos* tiene una serie de problemas:

- Implementación del algoritmo se complica.
- Repetición de un conjunto de bloques.
- Más difícil de entender.

Todos estos problemas se pueden suavizar con la programación modular.

1. Introducción

```
#include <stdio.h>
main () {
        int n, m ;
        float nf, mf, nmf, result ;
        float fact(int) ;

        printf("Dame el valor de n y m: ") ;
        scanf("%d %d", &n, &m) ;
        nf = fact(n) ; mf = fact(m) ; nmf = fact(n-m) ;
        result = nf / (mf * nmf) ;
        printf("El factorial de %d es %f, el de %d es %f\n",n,nf,m,mf) ;
        printf("Por lo que la combinatoria es: %f\n", result) ;
        return ;
}

float fact(int x) {
        float r = 1 ;
        int i ;
        for (i=x; i>0; i--) r *= i ;
        return r ;
}
```

1. Introducción

Los problemas tienen conjuntos de instrucciones que realizan una tarea con entidad propia.

La forma de resolver el problema teniendo en cuenta esta característica intrínseca de los programas es mediante la programación modular.

1. Introducción

La programación modular es una técnica que permite:

Dividir la complejidad del problema en tareas más simples y más sencillas de implementar. Cada una de estas tareas es un módulo.

Para que la composición del problema sea correcta, los módulos deben cumplir las siguientes características: alta cohesión y bajo acoplamiento.

1. Introducción

Las técnicas usadas para la modularización de un programa son:

Técnica descendente (*top-down*), consiste en resolver el programa principal supuesto que los subprogramas ya están escritos.

Técnica ascendente (*bottom-up*), consiste en resolver previamente los subprogramas.

1. Introducción

Si hasta ahora habíamos hablado de programas y decidimos usar la descomposición, a cada módulo le llamaremos subprograma, o más concretamente en C, funciones.

De tal forma que el programa original que habíamos presentado tendría dos funciones: la principal y el subprograma (o módulo o función) `fact()`.

1. Introducción

```
#include <stdio.h>

main () {

        float fact(int x);

        int n, m, i;
        float nf, mf, nmf, result;

        printf("Dame el valor de n y m: ");
        scanf("%d %d", &n, &m);

        nf = fact(n); mf = fact(m); nmf = fact(n-m);
        result = nf/mf/nmf;

        printf("\t-----------------------------\n");
        printf("El factorial de %d es %f, el de %d es %f\n",n,nf,m,mf);
        printf("Por lo que la combinatoria es: %f\n", result);
        printf("\t-----------------------------\n");


        return;
}

float fact(int x) {
        float r = 1;
        int i;

        for (i=x; i>0; i--) r *= i;
        return r;
}
```

Declaración de la función

Llamadas a la función

Código de la función

1. Introducción

El uso de subprogramas aporta ventajas:

- Claridad en estructuración.
- Más fácil de resolver: *divide y vencerás*.
- Más fácil de entender.
- Los subprogramas se pueden tratar y probar por separado.

Los subprogramas en C se llaman funciones. Éstas pueden "devolver" valores o no.

1. Introducción

Las ventajas del uso de funciones son:

- Se puede llamar a una función desde diferentes partes de un programa.
- Se puede transferir un conjunto de datos distinto cada vez, obteniendo resultados diferentes en cada caso.
- Los programas son más sencillos de implementar y corregir.
- Se puede crear un conjunto de funciones genéricas para usar como librerías por diferentes programas (reutilización de código).

2. Transferencia de información

- Una función permite agrupar un conjunto de instrucciones en un bloque que realizará una tarea determinada.
- Todo programa contiene una función principal (**`main`**) que es la encargada de llevar el control de ejecución del programa y de llamar a ejecución a las funciones en los puntos del programa donde se necesiten.
- Para que la función se ejecute debe ser siempre llamada por otra función (excepto la función **`main`**).

2. Transferencia de información

- Cuando una función F1 realiza una llamada a otra F2 se transfiere el control de ejecución de F1 a F2, empezando la ejecución de F2.
- F1 puede transferir ciertos datos a F2 en el momento de realizar la llamada. A estos datos se les llama *argumentos de entrada* de la función F2.
- Cuando F2 concluye su ejecución, devuelve el control a la función que la ha llamado, F1. Adicionalmente, F2 puede devolver un resultado a F1 que se denomina *salida* de la función F1.

2. Transferencia de información

- La estructura de una función en C se divide en dos partes:
  - Cabecera de la función
  - Cuerpo de la función

- En la cabecera se define el nombre de la función y el modo en que se va a realizar la transferencia de información: argumentos de entrada y tipo de dato de salida.

2. Transferencia de información

¿Cual es la estructura de una función en C?
Tiene una cabecera y un cuerpo.

```
tipo_dato nombre_funcion(lista_args) {
        /* Declaración de variables */
        /* Instrucciones */
        ……….
}
```

2. Transferencia de información

- *Tipo_dato*: tipo de dato del valor que devuelve. En caso que no devuelva ningún resultado, se debe especificar **`void`**.
- *Lista_args*: Parejas *tipo_dato variable*, separadas por comas, a los que se denomina *argumentos* o *parámetros formales*. Permiten la transferencia de información a la función desde el punto donde es invocada.

El cuerpo de la función está encerrado entre llaves y está formado por la declaración de variables y las instrucciones que resuelven la tarea.

2. Transferencia de información

Algunos ejemplos de cabeceras de funciones:

Función que dice si un entero es primo:

```
int numprim (int a)
```

Función que imprime el cálculo de alguna operación:

```
void imprimir (int a, float b, char c)
```

2. Transferencia de información

Dadas dos funciones F1 y F2, donde F1 llama a F2, se pueden establecer diferentes flujos de información:

- F1 puede transferir datos a F2, a través de los argumentos formales especificados en la cabecera de F2.
- F2 puede devolver un resultado (un dato) a F1.

Estas dos transferencias de información en las llamadas a funciones no son obligatorias.

2. Transferencia de información

Una llamada a una función es una instrucción donde se especifica el nombre de la función invocada y los valores que se transfieren a dicha función.

Para llamar a una función desde main o desde cualquier otra función, se usa la sintaxis:

```
nombre_funcion (arg1, arg2, …, argn) ;
```

Si la función no requiere ningún argumento se escribe el nombre seguido de paréntesis:

```
nombre_función () ;
```

2. Transferencia de información

- A los argumentos que aparecen en la llamada se les denomina *argumentos* o *parámetros actuales* o reales y contienen los valores con los que la función invocadora va a llamar a la función.

- Estos argumentos deben coincidir en número y tipo con los argumentos formales definidos en la cabecera de la función.

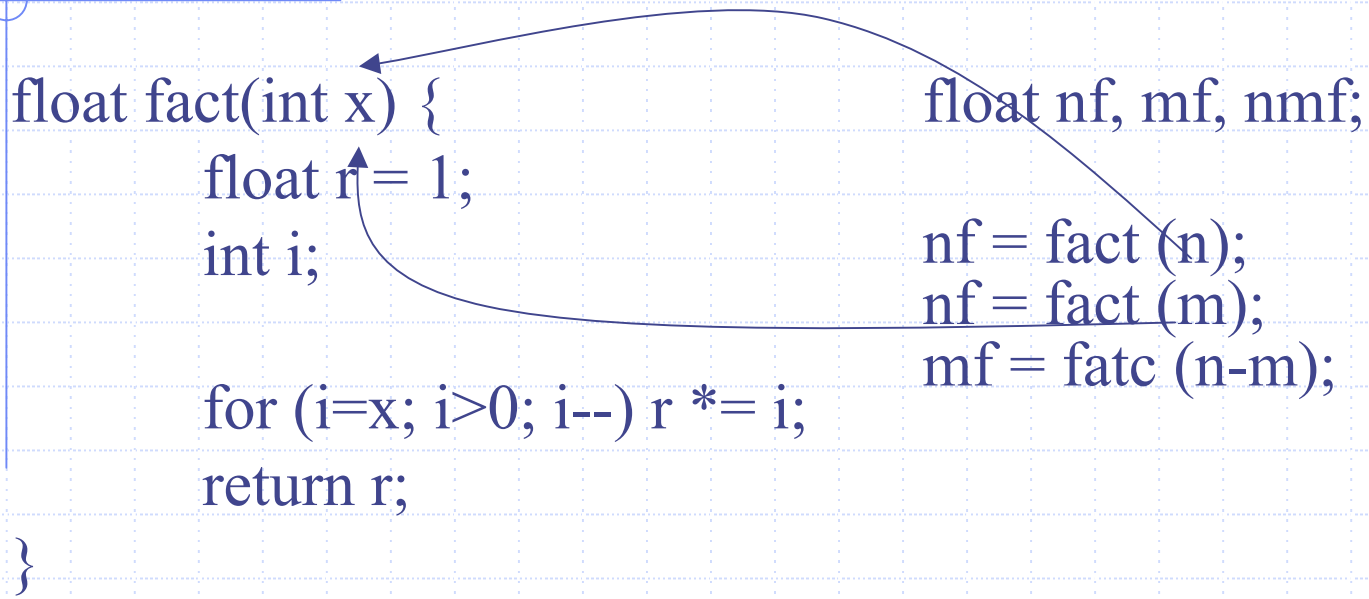
2. Transferencia de información

Los identificadores que aparecen en la cabecera de la función son los parámetros formales. Son la entrada al subprograma.

Los identificadores que aparecen en la llamada de la función se llaman parámetros actuales.

Deben coincidir en número y tipos. Su emparejamiento es POR POSICIÓN, *nunca* POR NOMBRE.

2. Transferencia de información



```
float fact(int x) {
        float r = 1;
        int i;

        for (i=x; i>0; i--) r *= i;
        return r;
}
```

```
float nf, mf, nmf;

nf = fact (n);
nf = fact (m);
mf = fatc (n-m);
```

2. Transferencia de información

- Además de la introducción de información, una función invocada puede devolver un resultado a la función invocadora.

- Si la función llamada no devuelve nada, se debe especificar el tipo **`void`** en su cabecera. Si devuelve algún dato, entonces debe especificarse su tipo. En ese caso, hay que incluir la instrucción **`return`** dentro del cuerpo de la función.

2. Transferencia de información

- La sentencia **`return`** realiza dos tareas:
  - Devuelve el control de ejecución a la función invocadora.
  - Devuelve, si se desea, un valor resultado a la función invocadora.

- La sintaxis de la sentencia **`return`** es:

```
return expresión ;
```

donde, si no se incluye ninguna expresión, la función no devuelve ningún dato y si se pone una constante, variable o expresión, entonces devuelve ese dato a la función invocadora.

2. Transferencia de información

- La única restricción es que el valor contenido en la expresión sea del mismo tipo que el definido en la cabecera de la función.

- Solamente se puede incluir una expresión en una sentencia **`return`**, por tanto, sólo se puede devolver un valor al punto de llamada.

- Una función puede contener varias sentencias **`return`**, con expresiones diferentes, pero sólo una se debe ejecutar.

2. Transferencia de información

Por ejemplo:

```
if (a>b)
        return a ;
else
        return b ;
```

Al devolver el valor a la función invocadora, ésta puede recogerlo de varias formas:
a) Asignándolo a una variable. En este caso, el tipo de la variable debe coincidir con el tipo del dato devuelto por la función.

```
variable = nombre_funcion(arg1, arg2, …, argn) ;
```

2. Transferencia de información

b) Utilizar directamente el dato devuelto en una instrucción. En ese caso no es necesario guardarlo en una variable.

```
#include <stdio.h>
main () {
        int num1, num2, resultado ;
        int suma(int, int) ;
        printf("Dame dos enteros: ");
        scanf ("%d%d", &num1, &num2) ;
        printf("La suma es %d\n", suma(num1,num2)) ;
}
int suma (int a, int b) {
        return (a+b) ;
}
```

2. Transferencia de información

También puede suceder que la función invocada no devuelva ningún resultado. En ese caso su invocación es como una instrucción más del programa.

```
#include <stdio.h>
main () {
        void leer() ;
        leer() ;
}
void leer () {
        int num ;
        printf("Dame un número entero: ") ;
        scanf("%d\n", &num) ;
        printf("El número leido es %d\n", num) ;
}
```

2. Transferencia de información

Ejercicio: Construir un programa en C para calcular el máximo común divisor de 3 números usando una función que calcule el máximo común divisor de dos números.

3. Paso de parámetros a una función

- Toda función dispone de una zona de memoria (*registro de activación*) donde se almacenan las variables definidas dentro del cuerpo de la función, las variables que actúan como argumentos formales así como distintos resultados que se vayan produciendo durante la ejecución.
- En el momento en que una función comienza su ejecución se realiza una reserva de memoria para todo eso. Cuando finaliza su ejecución, se libera esa zona de memoria.

3. Paso de parámetros a una función

- Cuando se transfiere un valor como argumento a una función, la función copia el valor transferido a la zona de memoria del correspondiente parámetro formal.
- La función puede modificar el valor recibido, pero todas las modificaciones se realizan en la zona de memoria reservada para la función invocada.
- Por tanto, no se modifica el valor del parámetro actual.

3. Paso de parámetros a una función

- La única conexión posible de una función con quien la llame debe ser SOLO los parámetros formales.
- Cuando la función es llamada se crea en memoria un espacio para sus objetos locales. Tras la ejecución ese espacio se libera.
- El procedimiento para pasar los parámetros para que se ejecuten en una función es el paso de parámetros por valor.

3. Paso de parámetros a una función

```
#include <stdio.h>

main () {

     void modificar(int );
     int n = 3;

     printf("Valor de n antes de modificar es %d",n);
     modificar(n);
     printf("Valor de n despu□s de modificar es %d",n);
     return;
}

void modificar(int x) {

     x = x*6;
     printf("Valor de n dentro de modificar es %d",x);
}
```

¿Cómo lo solucionamos?

3. Paso de parámetros a una función

```
#include <stdio.h>

main () {
     void modificar(int *);
     int n = 3;

     printf("Valor de n antes de modificar es %d",n);
     modificar(&n);
     printf("Valor de n despu□s de modificar es %d",n);
     return;
}

void modificar(int *x) {

     *x = *x*6;
     printf("Valor de n dentro de modificar es %d",x);
}
```

3. Paso de parámetros a una función

```
#include <stdio.h>

main () {

     void cambiar(int *, int *);
     int n, m;

     printf("Dame los valores : ");
     scanf("%d %d",&n,&m);
     cambiar(&n, &m);
     printf("Valor de n es %d y de m es %d",n,m);
     return;
}

void cambiar(int *x, int *y) {
     int z;

     z = *x;
     *x = *y;
     *y = z;
}
```

4. Objetos locales y globales

- Las variables que se incluyen dentro de una función son llamadas variables locales. Por ejemplo, los parámetros formales son locales a la función.
- Las variables declaradas en un programa con subprogramas son variables globales. Estas variables suelen ser las declaradas fuera de cualquier función.
- La amplitud de una variable es la parte de código en el que la variable es conocida.

5. Recursividad

- Un objeto se define recursivamente si él mismo interviene de alguna forma en su definición.

```
n >= 0
      1.- fact(n) = n * fact(n-1)
      2.- fact(0) = 1
```

5. Recursividad

Un suprograma es recursivo si:

- Realiza una llamada a sí mismo.
- La realiza a otro que llama al primero

Al escribir la recursión debemos identificar:

1.- El caso base que rompe la recursión.

2.- La llamada recursiva

6. Ámbito de las variables

Tipos de almacenamiento: Hay dos formas de caracterizar variables: por su tipo de dato y por su tipo de almacenamiento.

- El tipo de dato se refiere al tipo de información representada por la variable.
- El tipo de almacenamiento se refiere a la permanencia de la variable y a su ámbito dentro del programa, que es la porción del programa en donde se reconoce la variable.

6. Ámbito de las variables

Tipos de almacenamiento diferentes en C: automática, externa y estática.

Están identificadas por las palabras clave **`auto`**, **`extern`**, y **`static`**.

Variables automáticas: Se declaran dentro de una función y son locales a ella.

Variables automáticas definidas en funciones diferentes son independientes unas de otras, incluso si tienen el mismo nombre.

6. Ámbito de las variables

Para decir que una variable es automática se precede su declaración con la palabra auto, aunque ésta no es necesaria, ya que si no se pone nada explícitamente, son automáticas.

```
#include <stdio.h>
main()
{       auto int n ;
        long int factorial(int n) ;
        printf("\nn = ") ;
        scanf("%d", &n) ;
        printf("\nn != %ld", factorial (n)) ;
}
```

6. Ámbito de las variables

```
long int factorial (auto int m)
{       auto int n ;
        auto long int prod = 1 ;
                if (m > 1)
                for (n=2; n <= m; ++n) prod *= n ;
        return (prod) ;
}
```

Variables externas: Las variables que están en **main** son privadas o locales a ella. Ninguna otra función puede tener acceso directo a ellas. Eso mismo ocurre para variables de otras funciones.

6. Ámbito de las variables

Cada variable local de una función comienza a existir sólo cuando se llama a la función y desaparece cuando la función termina. Se les suele llamar variables automáticas.

Como alternativa, es posible definir variables que son externas a todas las funciones, esto es, variables a las que toda función puede tener acceso por su nombre.

5. Ámbito de las variables

Una variable externa debe definirse una sola vez y fuera de cualquier función. Esto fija un espacio de almacenamiento para ella.

La variable externa también debe declararse en cada función que desee tener acceso a ella. Esto establece el tipo de la variable.

6. Ámbito de las variables

La declaración debe ser una proposición **extern** explícita o puede estar implícita en el contexto (si la definición ocurre dentro del archivo fuente antes de su uso por una función particular). Por ejemplo:

```
#include <stdio.h>
#include <math.h>
#define CNST 0.0001

double a, b, x1, x2, y1 ;

main()
{
        :
}
```

6. Ámbito de las variables

Variables estáticas: Se debe hacer la distinción entre un programa de archivo simple donde el programa completo está contenido en un archivo fuente simple, y el programa de archivo múltiple, donde las funciones que componen el programa están contenidas en archivos fuente separados. Nos centramos en el primer caso.

6. Ámbito de las variables

En un programa de archivo simple las variables estáticas se definen dentro de las funciones individuales y tienen, por tanto, el mismo ámbito que las variables automáticas; son locales a la función en la que están definidas.

A diferencia de las variables automáticas, las variables estáticas retienen sus valores durante toda la vida del programa. Como consecuencia, si se sale de una función y posteriormente se vuelve a entrar, las variables estáticas definidas dentro de esa función retendrán sus valores.

6. Ámbito de las variables

Se definen en una función de igual forma que las automáticas, sólo que precedidas de la palabra reservada `static`.

Estas variables no pueden ser accedidas fuera de las funciones que las definen.

Tema 5: Vectores

Contenido

- Introducción
- Definición de vector
- Operaciones con vectores
- Paso de vectores como argumentos a funciones
- Vectores multidimensionales
- Vectores y punteros

1. Introducción

- Hasta ahora hemos trabajado con datos de tipo simple.
- Existen también datos compuestos o estructurados que permiten almacenar o agrupar colecciones de datos de tipo simple o compuesto.
- Si todos los datos almacenados son del mismo tipo el tipo de dato compuesto se denomina vector.
- Si los datos de la agrupación son de distinto tipo se obtiene el tipo de dato compuesto estructura.

2. Definición de vector

- Un vector es un tipo de dato estructurado que almacena datos homogéneos (carácter, entero o real o también vectores, estructuras, listas, ...)
- A los datos almacenados se les denomina elementos.
- Al número de elementos se le denomina tamaño o rango del vector.
- Para acceder a los elementos individuales se emplea un índice (entero no negativo) que indica la posición del elemento dentro del vector.

2. Definición de vector

- Declaración:

```
tipo_elemento nombre_vector[rango] ;
```

- Ejemplo:

```
#define rango 30
int datos[10] ;
char texto[40] ;
float temperatura[rango] ;
datos[5] = 56 ;
printf("%d",datos[5]) ;
```

2. Definición de vector

- Las variables vector, como ocurre con el resto de tipos, pueden ser inicializadas en la declaración:

```
float temperatura[4] = {12.3,25.5,26.0,21.4} ;
```

- También se puede dar un valor inicial sólo a una parte de los elementos:

```
float temperatura[4] = {12.3,25.5} ;
```

- El tamaño del vector no es necesario especificarlo si se introducen valores iniciales:

```
float temperatura[] = {12.3,25.5,26.0,21.4} ;
```

3. Operaciones con vectores

- En C no se permiten operaciones que impliquen vectores completos salvo para vectores de caracteres.

- 1. Entrada y salida.

  Ejemplo: Programa que genera un vector de 5 componentes y muestra su contenido por pantalla.

3. Operaciones con vectores

```
#include <stdio.h>
#define rango 5
main () {
      int vector[rango], i ;
      for (i=0;i<rango;i++){
            printf("Número: ") ;
            scanf("%d",&vector[i]) ;
      }
      for (i=0;i<rango;i++)
            printf("\n El valor del elemento %d es
                  %d",i+1,vector[i]) ;
}
```

3. Operaciones con vectores

- 2. Operaciones de acceso a un vector.
- A un elemento de un vector se accede del mismo modo que a una variable simple del mismo tipo.

Ejemplo: Programa que lee las notas de los alumnos de una clase para una asignatura, calcula la media y determina cúantos alumnos superan o igualan la media y cuántos están por debajo de ella.

3. Operaciones con vectores

```
#include <stdio.h>
#define alumnos 50
main () {
      float notas[alumnos], media, suma=0.0 ;
      int i, mayor, menor ;
      printf("Introduce las notas") ;
      for (i=0;i<alumnos;i++){
            do {
              printf("\n Nota alumno %d: ",i+1) ;
              scanf("%f",&notas[i]) ;
              if ((notas[i]<0) || (notas[i]>10))
                printf("\n Nota incorrecta") ;
            } while ((notas[i]<0)||(notas[i]>10)) ;
      suma = suma+notas[i] ;
      }
```

3. Operaciones con vectores

```
        media=suma/alumnos ;
        mayor=menor=0 ;
        for (i=0;i<alumnos;i++)
              if (media<=notas[i])
                    mayor++ ;
              else menor++ ;
        printf("\n La media es %f \n",media) ;
        printf("Mayor igual a la media: %d\n",mayor);
        printf("Menor a la media: %d\n",menor) ;
}
```

4. Paso de vectores como argumentos a funciones

- El nombre de un vector puede usarse como argumento actual en una función.
- Para pasar un vector a una función, en el argumento actual se escribe el nombre del vector sin corchetes ni índices.
- El argumento formal debe ser declarado como un vector dentro de la declaración de argumentos de la función.

4. Paso de vectores como argumentos a funciones

- En C no es necesario especificar el tamaño del vector en los argumentos formales y es posible poner un par de corchetes vacíos.
- Esto permite crear funciones más genéricas en las que no exista limitación del tamaño del vector.

Ejemplo: Programa que almacena valores en un vector de números reales y calcula su media. Vamos a construir las funciones para dimensiones genéricas del vector.

4. Paso de vectores como argumentos a funciones

```
#include <stdio.h>
#define rango 100
main () {
      float vector[rango], med;
      void leervector(int, float []) ;
      float media(int, float []) ;

      leervector(rango,vector) ;
      med = media(rango,vector) ;
      printf("La media es %f\n",med) ;
}
```

4. Paso de vectores como argumentos a funciones

```
void leervector (int dim, float a[])
{
        int i ;
        for (i=0;i<dim;i++){
                printf("\n Número %d: ",i+1) ;
                scanf("%f",&a[i]) ;
        }
}
float media (int dim, float x[]) {
        int i ;
        float total=0.0 ;
        for (i=0;i<dim;i++)
                total = total+x[i] ;
        return (total/dim) ;
}
```

5. Vectores multidimensionales

- Se define un vector bidimensional como un vector cuyos elementos son a su vez vectores. Se les conoce comúnmente como matrices.
- Declaración:

```
tipo_dato nombre_vector[rango1][rango2] ;
```

- Ejemplo: la lectura de un vector bidimensional de números reales.

5. Vectores multidimensionales

```
#include <stdio.h>
#define rango 5
main () {
        float matriz[rango][rango], i, j;
        for (i=0;i<rango;i++)
                for (j=0;j<rango;j++) {
                        printf("Elemento [%d][%d]:",i,j);
                        scanf("%f",&matriz[i][j]) ;
                }
}
```

5. Vectores multidimensionales

- Cuando se pasan vectores bidimensionales a una función, la declaración de argumentos formales debe incluir especificaciones explícitas de la segunda de las dimensiones, pudiendo dejar vacía la primera.

```
tipo_dato funcion(tipo_dato vector[][rango2]) ;
```

- Ejemplo: Programa para realizar la suma de dos matrices de números reales.

5. Vectores multidimensionales

```
#include <stdio.h>
#define N 5
main ()
{
        float mat1[N][N],mat2[N][N],mat3[N][N] ;
        void leermatriz(float [][N]);
        void suma(float [][N],float [][N],float [][N]);
        void vermatriz(float [][N]);

        printf("\n Valores de la primera matriz:");
        leermatriz(mat1);
        printf("\n Valores de la segunda matriz:");
        leermatriz(mat2);
        suma(mat1,mat2,mat3) ;
        vermatriz(mat3) ;
}
```

5. Vectores multidimensionales

```
void leermatriz (float matriz[][N])
{
      int i,j ;
      for (i=0;i<N;i++)
        for (j=0;j<N;j++){
            printf("\n Elemento [%d][%d]: ",i,j) ;
            scanf("%f",&matriz[i][j]) ;
      }
}
void suma(float m1[][N],float m2[][N],float m3[][N])
{
      int i,j ;
      for (i=0;i<N;i++)
            for (j=0;j<N;j++)
                  m3[i][j]=m1[i][j]+m2[i][j] ;
}
```

5. Vectores multidimensionales

```
void vermatriz (float m[][N])
{
  int i,j ;
  for (i=0;i<N;i++)
    for (j=0;j<N;j++)
      printf("\nElemento [%d][%d] es %f",i,j,m[i][j])

}
```

5. Vectores multidimensionales

- Los vectores multidimensionales son vectores que contienen vectores que a su vez contienen vectores ...
- Cada una de las nuevas dimensiones se expresa a través de un nuevo índice entre corchetes.
- Cuando se pasan como argumentos a una función, la declaración debe incluir especificaciones explícitas de tamaño en todos los índices excepto el primero.

6. Vectores y punteros

- En C existe una fuerte relación entre punteros y vectores.
- Cualquier operación que pueda realizarse por indexación de un vector, también puede hacerse con punteros. La versión con punteros será, por lo general, más rápida.
- La declaración `int a[10] ;` define un vector de tamaño 10 (10 objetos consecutivos `a[0],…,a[9]`).
- La notación `a[i]` se refiere al elemento i-ésimo del array.

6. Vectores y punteros

- Si `pa` es un puntero a un entero declarado como `int *pa ;` entonces `pa = &a[0] ;` hace que `pa` apunte al elemento cero de `a`, es decir, `pa` contiene la dirección de `a[0]`.
- Si `pa` apunta a un elemento particular de un array, entonces, por definición, `pa+1` apunta al siguiente elemento.
- Si `pa` apunta a `a[0]`, `*(pa+1)` se refiere al contenido de `a[1]`, `pa+i` es la dirección de `a[i]` y `*(pa+i)` es el contenido de `a[i]`.

6. Vectores y punteros

- La correspondencia entre indexación y aritmética de punteros es muy estrecha.
- Por definición, el valor de una variable o expresión de tipo vector es la dirección del elemento cero del vector.
- Después de la asignación `pa = &a[0]` ; `pa` y `a` tienen valores idénticos.
- Puesto que el nombre de un vector es sinónimo para la localidad del elemento inicial, la asignación `pa = &a[0]` puede escribirse también como `pa = a`.

6. Vectores y punteros

- Una referencia a `a[i]` también puede escribirse como `*(a+i)`. Al evaluar `a[i]`, C la convierte inmediatamente a `*(a+i)`. Ambas formas son equivalentes.
- Al aplicar el operador `&` a ambas partes de esta equivalencia, se deriva que `&a[i]` y `a+i` también son idénticas: `a+i` es la dirección del elemento i-ésimo delante de `a`.
- Si `pa` es un puntero, las expresiones pueden usarlo con un subíndice: `pa[i]` es idéntico a `*(pa+i)`.

6. Vectores y punteros

- En resumen, cualquier expresión de vector e índice es equivalente a una expresión escrita con un puntero y un desplazamiento.

- Pero existe una diferencia entre un nombre de un vector y un puntero: un puntero es una variable, por esto `pa = a` y `pa++` son legales. Pero un nombre de un vector no es una variable, así `a = pa` y `a++` son ilegales.

6. Vectores y punteros

- Cuando se pasa un nombre de vector a una función, lo que se pasa es la localización del elemento inicial.
- Dentro de la función que se llama, este argumento es una variable local y, por lo tanto, un parámetro de nombre de vector es un puntero, esto es, una variable que contiene una dirección.
- También es posible pasar parte de un vector a una función, pasando un puntero al inicio del subvector. Por ejemplo, si `a` es un vector, `f(&a[2])` y `f(a+2)` pasan a la función `f` la dirección del subvector que comienza en `a[2]`.

6. Vectores y punteros

- Dentro de `f`, la declaración de parámetros puede ser `f(int arr[])` o `f(int *arr)`
- También es posible indexar hacia atrás en un vector: `p[-1]`
- Puesto que en sí mismos los punteros son variables, pueden almacenarse en vectores como otras variables.
- En el caso de los vectores multidimensionales, la forma de tratamiento respecto a los punteros es análoga a los unidimensionales.

6. Vectores y punteros

- Por ejemplo, supongamos que queremos declarar `x` como vector bidimensional de enteros con 10 filas y 20 columnas.
- Podemos declarar `x` como `int (*x)[20] ;` en vez de `int x[10][20] ;`
- En la primera declaración `x` se define como puntero a un grupo contiguo de vectores unidimensionales de 20 elementos enteros. Así `x` apunta al primero de los vectores de 20 elementos, que es en realidad la primera fila del vector bidimensional original. Similarmente `(x + 1)` apunta al segundo vector de 20 elementos, etc.

6. Vectores y punteros

- Para acceder al elemento 2, 5 de este vector podemos escribirlo como `x[2][5]` o bien `*(*(x+2)+5)`.
- `x + 2` es un puntero a la fila 2. Por tanto, el objeto de ese puntero `*(x + 2)` es toda la fila.
- Como la fila 2 es un vector unidimensional, `*(x + 2)` es un puntero al primer elemento de la fila 2. Sumamos 5 a ese puntero, `*(*(x+2)+5)` se refiere al elemento en la columna 5 de la fila 2.
- Usar una forma u otra de acceso a los elementos de los arrays es una elección absolutamente personal.

Tema 5-II: Cadenas

Contenido

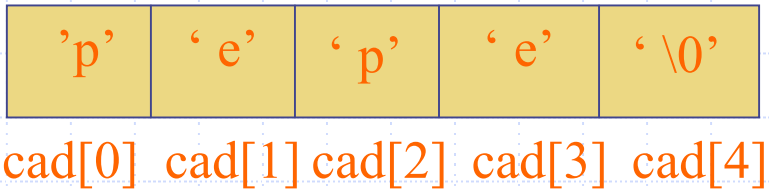
- Definición de una cadena
- Operaciones con cadenas

1. Definición de una cadena

- Una cadena de caracteres es un conjunto de caracteres almacenados en un vector.
- Las cadenas tienen un tratamiento especial en los lenguajes de programación y suele permitirse realizar operaciones de lectura y escritura directamente sobre la totalidad de sus elementos.
- En C existe un archivo de cabecera que contiene funciones especializadas para el tratamiento de cadenas de caracteres: **`<string.h>`**

1. Definición de una cadena

- Una cadena de caracteres es un vector de caracteres que finaliza con un carácter especial **\0** denominado carácter final de cadena.
- Por ejemplo, la cadena "pepe" se almacena como:



Hay que tener la precaución de reservar un elemento para indicar el fin de cadena.

```
char cad[5] ;
```

1. Definición de una cadena

- Al declarar una cadena de caracteres se le puede dar un valor inicial:

```
char ciudad[7]="Madrid" ;
```

- En este caso, es el compilador el que finaliza la cadena con el carácter `'\0'`.
- Para el compilador, `'x'` representa un solo carácter. En cambio, `"x"` representa una cadena de caracteres formada por dos caracteres `x` y `\0`.
- La declaración anterior es equivalente a:

```
char ciudad[7]={'M','a','d','r','i','d','\0'} ;
```

2. Operaciones con cadenas

- Los vectores de caracteres pueden ser tratados carácter a carácter o en su totalidad:

1. Operaciones carácter a carácter

1.1 Asignación de caracteres: usando la instrucción de asignación y dando el valor **\0** al último carácter de la cadena.

```
char ciudad[7] ;
ciudad[0]='M' ; ciudad[1]='a' ;
ciudad[2]='d' ; ciudad[3]='r' ;
ciudad[4]='i' ; ciudad[5]='d' ;
ciudad[6]='\0' ;
```

2. Operaciones con cadenas

1.2 Lectura y escritura: para leer o escribir los elementos de un vector de caracteres se usará un bucle con `scanf` y `printf`.

- Ejemplo: lectura del nombre de una ciudad.

```
#include <stdio.h>
main () {
        int i ;
        char ciudad[10] ;
        for (i=0;i<9;i++) {
                /* fflush(stdin) ; */
                scanf("%c",&ciudad[i]) ;
        }
        ciudad[9]='\0' ;
}
```

2. Operaciones con cadenas

- Ejemplo: lectura del nombre de una ciudad.

```
#include <stdio.h>
#define N 50
main () {
        int i=0 ;
        char ciudad[N],ch ;
        scanf("%c",&ch) ;
        while (ch!='\n') {
                ciudad[i]=ch ;
                i++ ;
                scanf("%c",&ch) ;
        }
        ciudad[i]='\0' ;
}
```

2. Operaciones con cadenas

- En C existe otra instrucción que para leer caracteres: `getchar()`, que devuelve el carácter introducido desde teclado.

```
#include <stdio.h>
#define N 50
main () {
        int i=0 ;
        char ciudad[N],ch ;
        while ((ch=getchar())!='\n') {
                ciudad[i]=ch ;
                i++ ;
        }
        ciudad[i]='\0' ;
}
```

2. Operaciones con cadenas

```
#include <stdio.h>
#define N 50

main ()
{
        char ciudad[N] ;
        void leer(char []) ;
        void escribir(char []) ;

        leer(ciudad) ;
        escribir(ciudad) ;
}
```

2. Operaciones con cadenas

```
void escribir(char cadena[]) {
        int i=0 ;
        while (cadena[i]!='\0')
                printf("%c",cadena[i++]) ;
        printf("\n") ;
}

void leer(char cadena[]) {
        int i=0 ;
        char ch ;
        scanf("%c",&ch) ;
        while (ch!='\n') {
                cadena[i]=ch ;
                i++ ;
                scanf("%c",&ch) ;
        }
        cadena[i]='\0' ;
}
```

2. Operaciones con cadenas

2. Funciones de cadenas de caracteres

- En C existen varias funciones para el tratamiento de cadenas de caracteres que facilitan la lectura, escritura o asignación de valores.

2.1. Lectura y escritura de cadenas, usando `%s` en la cadena de control.

- La función **`scanf`** lee hasta que encuentra un carácter en blanco, un tabulador o un retorno de carro.
- La función **`printf`** muestra por pantalla el contenido del vector hasta encontrar el carácter `'\0'`.

2. Operaciones con cadenas

```
#include <stdio.h>
#define N 50
main ()
{
        char ciudad[N] ;

        scanf("%s",ciudad) ;
        printf("%s",ciudad) ;
}
```

- Existen dos funciones especializadas que realizan la misma función de lectura y escritura:

```
gets(vector_caracteres) ;
puts(vector_caractares) ;
```

`gets` lee hasta encontrar el carácter retorno de carro.

2. Operaciones con cadenas

```
#include <stdio.h>
#define N 50
main ()
{
        char ciudad[N] ;
        gets(ciudad) ;
        puts(ciudad) ;
}
```

- Con el archivo de cabecera `<string.h>` se pueden usar más operaciones sobre cadenas.
- Operación de asignación. No se puede asignar un valor a una cadena de caracteres con =.

2. Operaciones con cadenas

- Para ello se usa la función strcpy:

```
char *strcpy(cadena_destino,cadena_origen);
```

- Ejemplo:

```
char nombre1[20],nombre2[20] ;
strcpy(nombre1,"Pepe Perez") ;
strcpy(nombre2,nombre1) ;
```

- Longitud de una cadena. Usando la función

```
long int strlen(nombre_cadena) ;
```

- Funciones de comparación. La función **strcmp** acepta dos cadenas y devuelve un valor entero:

```
int strcmp(cadena1,cadena2) ;
```

2. Operaciones con cadenas

- El resultado es:
  - Un valor negativo si la primera cadena alfabéticamente precede a la segunda.
  - El valor cero si son idénticas.
  - Un valor positivo si la segunda precede alfabéticamente a la primera.
- La función **`strncmp`** es muy similar. Acepta dos cadenas y devuelve un valor entero, dependiendo del orden relativo de los primeros longitud caracteres.

```
int strncmp(cadena1,cadena2,longitud)
```

2. Operaciones con cadenas

- Funciones de concatenación. La función `strcat` permite encadenar dos cadenas

```
char *strcat(cadena_destino,cadena_origen) ;
```

añade una copia de `cadena_origen` al final de la `cadena_destino.`

- La función `strncat` es similar, salvo que sólo copia los primeros `longitud` caracteres:

```
char *strncat(cadena_destino,cadena_origen,longitud);
```

2. Operaciones con cadenas

- Funciones de búsqueda. La función `strchr` busca la primera ocurrencia del carácter indicado dentro de la cadena. Devuelve la cadena formada a partir del carácter encontrado incluido:

```
char *strchr(cadena,caracter) ;
```

devuelve `char *` indicando que devuelve la dirección de memoria del primer elemento de la cadena.

- En caso de que no exista ninguna ocurrencia del carácter, la función devuelve una cadena inexistente o dirección nula (`NULL`).

Tema 6: Estructuras (registros)

Contenido

- Definición de una estructura
- Estructuras anidadas
- Inicialización
- Operaciones sobre estructuras
- Definición de tipos de datos propios
- Paso de estructuras como parámetros
- La estructura union

1. Definición de una estructura

- Permite agrupar datos de diferentes tipos.
- Nuevo tipo de dato compuesto: `estructura`.
- Los datos que contiene pueden ser de tipo simple o de tipo compuesto.
- Cada uno de los elementos de la estructura se denomina miembro de la misma.

1. Definición de una estructura

- Sintaxis:

```
struct nombre {
        tipo_dato miembro1 ;
        tipo_dato miembro2 ;
        ...
} ;
```

- Una estructura define un tipo de dato, no una variable.
- Después habrá que declarar variables del tipo.

1. Definición de una estructura

- Para declarar variables hay dos formas:

1) Incluir las variables en la propia definición de la estructura.

```
struct nombre_estructura {
        int numero ;
        char tipo_cuenta ;
        float saldo;
} cliente1, cliente2 ;
```

1. Definición de una estructura

2) Definir el tipo con un nombre y después declarar las variables de ese tipo de dato.

```
struct nombre {
        int numero ;
        char tipo_cuenta ;
        float saldo;
} ;

struct nombre cliente1, cliente2 ;
```

1. Definición de una estructura

- Si se desea almacenar un conjunto de variables estructura puede hacerse con un vector:

```
struct cliente {
        int numero ;
        char tipo_cuenta ;
        float saldo ;
} clientes[1000] ;
```

donde **clientes** es un vector de estructuras.

2. Estructuras anidadas

- Las estructuras de datos pueden contener como miembros a otras estructuras:

```
struct fecha {
        int mes ;
        int dia ;
        int anyo ;
} ;
struct tarjetas {
        int numero ;
        char tipo_cuenta ;
        struct fecha pago ;
        float saldo ;
} cliente1, cliente2 ;
```

3. Inicialización

- Las variables de tipo estructura pueden inicializarse, como ocurre con cualquier otra variable, en el momento de la declaración:

```
struct tarjetas cliente1 =
        {12345, 'A', {21,5,2001}, 10000.0};
```

- Si tenemos un vector de estructuras, si se quiere, se pueden inicializar sólo las primeras componentes.

4. Operaciones sobre estructuras

- Se realizan normalmente sobre cada uno de los miembros de la estructura, salvo la asignación, que puede aplicarse sobre la estructura completa.

4.1 Acceso a una estructura

- Como los miembros se procesan individualmente, se debe tener acceso a cada uno de los miembros de una variable estructura:

```
variable.miembro
```

4. Operaciones sobre estructuras

- Si un miembro de una estructura es a su vez otra estructura:

```
variable.miembro.submiembro
```

4.2 Asignación, lectura y escritura:

```
cliente[6].saldo = 15.0 ;
scanf("%f",&cliente[5].saldo) ;
printf("El nombre es %s",cliente[1].nombre) ;
```

4. Operaciones sobre estructuras

4.3 Copia de los valores de estructuras:

- En C se permite copiar los datos de una variable estructura en otra sin necesidad de acceder a sus miembros.

```
cliente[5] = cliente[3] ;
```

5. Definición de tipos de datos propios

- En C se puede asignar un nombre a un determinado tipo de datos para después definir variables de este nuevo tipo de datos, usando la palabra reservada **typedef.**

```
typedef tipo_dato nombre_tipo ;
```

Ejemplo:

```
typedef char caracter ;
caracter A, B, Ch ;
```

6. Paso de estructuras como parámetros

- Se puede hacer de dos formas:
  - pasando uno o más miembros de la estructura, o
  - la estructura completa.

Ejemplo:

Incrementar el saldo de las tarjetas en un 5% si el último pago se ha realizado en los 6 primeros meses del año.

6. Paso de estructuras como parámetros

```
#include <stdio.h>
#define N 100
main()
{
int i ;
float incremento(float, int) ;
struct fecha {
        int mes ; int dia ; int anyo ;
}
struct tarjetas {
        long int num_tarjeta ;
        char tipo_cuenta ;
        char nombre[80] ;
        float saldo ;
        struct fecha ult_pago ;
} cliente[N] ;
```

6. Paso de estructuras como parámetros

```
        for (i=0; i<N; i++)

        cliente[i].saldo +=
                        incremento(cliente[i].saldo,
                        cliente[i].ult_pago.mes) ;
        }

float incremento (float elsaldo, int unmes) {
        if (unmes <= 6)
                return 0.05*elsaldo ;
        else return 0.0 ;
}
```

6. Paso de estructuras como parámetros

- Ejemplo:

  Crear un programa que emplee una función para introducir valores en los distintos elementos del vector cliente.

7. La estructura union

- Existe en C una estructura para almacenar datos muy parecida a **`struct`** en su definición pero muy diferente en su comportamiento:

```
union [nombre_union] {
        tipo_dato miembro1 ;
        tipo_dato miembro2 ;
        ...
} ;
```

7. La estructura union

- La diferencia es que una variable declarada de un tipo `union` sólo podrá almacenar uno de los miembros de la lista incluida en la declaración del tipo.

- El compilador reserva espacio en memoria para el miembro de mayor tamaño y todos los miembros apuntarán a esa dirección de memoria.

7. La estructura union

- Ejemplo: guardar un número que puede ser entero o real.

```
#include <stdio.h>
main() {
union {
        int num_entero ;
        float num_real ;
} numero ;
```

7. La estructura union

```
        numero.num_entero = 25 ;
        printf("\n %d",numero.num_entero) ;
        printf("\n %4.2f", numero.num_real ;
        numero.num_real = 36.15 ;
        printf("\n %d",numero.num_entero) ;
        printf("\n %4.2f", numero.num_real ;
}
```

- El resultado por pantalla es:

```
25
??
??
36.15
```

Tema 7: Estudio de algunos algoritmos

Contenido

- Introducción. Definiciones previas: talla, eficiencia temporal, eficiencia espacial.
- Algoritmos de búsqueda:
  - Lineal
  - Binaria o dicotómica
- Algoritmos de ordenación:
  - Selección directa (directos)
  - Fusión (rápidos)

1. Introducción

- Vamos a estudiar algoritmos clásicos para:
  - Determinar si, dado un elemento, se encuentra o no en un vector
  - Ordenar los elementos de un vector según algún criterio.
- Para cada problema veremos distintas soluciones, con características específicas de consumo de espacio y tiempo.
- Debemos poder comparar los algoritmos para determinar cuál es el mejor.

1. Introducción

- Talla de un problema: el parámetro o conjunto de parámetros que representan el volumen de datos a ser tratados para resolver el problema.
- Eficiencia temporal de un algoritmo: es una medida, función de la talla del problema a ser resuelto, del tiempo de cómputo necesario para su ejecución.
- Eficiencia espacial de un algoritmo: es una medida, función de la talla del problema, de la cantidad de memoria necesaria para resolverlo.

1. Introducción

- Es importante independizar dichas medidas de cualquier ordenador de forma que los resultados no dependan de las características de los mismos.

- El estudio debe centrarse en características intrínsecas del algoritmo o programa.

1. Introducción

- Procedimiento general:
  - 1. Definir la talla del problema.
  - 2. Razonar si hay comportamientos distintos para una talla dada.
  - 3. Si los hay, calcular dos eficiencias temporales: para el caso mejor y para el caso peor.
  - 4. Si no los hay, dar un solo resultado, la eficiencia temporal del algoritmo.
- Si el algoritmo es recursivo, usar las ecuaciones o relaciones de recurrencia.
- Si el algoritmo es iterativo, mediante sumatorios para los bucles.

2. Algoritmos de búsqueda

- Supongamos la siguiente declaración:

```
#define N …
typedef struct {
        tipo1 clave ;
        tipo2 resto ;
} elemento ;
elemento vector[N] ;
```

donde `N` se define como un valor entero, `tipo1` es un tipo cualquiera que permite comparación y `tipo2` es un tipo cualquiera.

2. Algoritmos de búsqueda

- El problema general consiste en:

Dado un `vector` y un valor cualquiera del tipo `tipo1`, encontrar el índice `i`, con `0<=i<N` tal que

- `vector[i].clave==valor`, si se encuentra en el vector,
- o `i=-1` en caso contrario.

2. Algoritmos de búsqueda

- Búsqueda lineal (vector desordenado)

```
int busca1(elemento vec, tipo1 val) {
        int i ;
        i=-1 ;
        do {
                i++ ;
        } while ((vec[i].clave!=val) && (i != N-1)) ;
        if (vec[i].clave == val)
                return i ;
        else
                return -1 ;
        }
```

2. Algoritmos de búsqueda

- La talla es `N`, el número de elementos del vector.
- Hay comportamientos distintos para una talla dada.
- Caso mejor: el elemento se encuentra inmediatamente.

$$COSTE_{MEJOR}(N) = 7 \text{ operaciones} = K \text{ operaciones}$$

- Caso peor: es el último elemento o no se encuentra en el vector.

$$COSTE_{PEOR}(N) = (3 + 4 * N \text{ operaciones}) = (K + K' * N)$$

2. Algoritmos de búsqueda

- Búsqueda lineal (vector ordenado)
- Si el vector está ordenado (ascendentemente) por una clave, es posible realizar una búsqueda "más eficiente".

```
int busca2(elemento vec, tipo1 val) {
        int i ;
        i=-1 ;
        do {
                i++ ;
        } while ((vec[i].clave<val) && (i != N-1)) ;
        if (vec[i].clave == val)
                return i ;
        else
                return -1 ;
        }
```

2. Algoritmos de búsqueda

- En este caso, cambian las características del caso mejor y peor, aunque la eficiencia resultante es similar a la anterior:
- Caso mejor: el elemento se encuentra en la primera posición del vector o es menor que el primero.

$$COSTE_{MEJOR}(N) = 7 \text{ operaciones} = K \text{ operaciones}$$

- Caso peor: el elemento no se encuentra en el vector o es mayor que el último.

$$COSTE_{PEOR}(N) = (3 + 4 * N) = (K + K' * N)$$

2. Algoritmos de búsqueda

- Búsqueda binaria o dicotómica

  Un tercer método de búsqueda, similar al de bisección para determinar la raíz de una función.

- Consiste en dividir el vector (ordenado) en dos partes de igual tamaño, estudiar en qué parte se puede encontrar el elemento y buscar de nuevo sólo en esa parte.

- La solución precisa que las claves se encuentren ordenadas ascendentemente.

2. Algoritmos de búsqueda

```
int busca3(elemento v[N],int ini,int fin,tipo1 val) {
        int mitad ;
        if (ini <= fin) {
        mitad=(ini+fin) / 2 ;
        if (v[mitad].clave == val)
                return mitad ;
        else
                if (val < v[mitad].clave)
                        return busca3(v,ini,mitad-1,val) ;
                else
                        return busca3(v,mitad+1,fin,val) ;
        }
        else
                return -1
}
```

2. Algoritmos de búsqueda

- Caso mejor: el elemento se encuentra en la mitad del vector.

$$COSTE_{MEJOR}(N) = K \text{ operaciones}$$

- Caso peor: el elemento se encuentra en la última iteración o no se encuentra en el vector.
- Como la solución es recursiva, construimos las ecuaciones o relaciones de recurrencia:

$$coste(N) = coste(N/2)+K \quad \text{para } N>0$$
$$coste(N) = K' \quad \text{para } N=0$$

2. Algoritmos de búsqueda

$coste(N) = coste(N/2)+K = coste(N/2^2)+2K = \ldots =$

$coste(N/2) = coste(N/2^2)+K$

$= coste(N/2^i)+iK =$

$N/2^i = 1 \qquad i = \log_2(N)$

$= K'+K \log_2(N) = O(\log_2(N))$

2. Algoritmos de ordenación

- Se desea ordenar (ascendentemente) los elementos de un vector por los valores de su campo clave.

- Dos métodos de ordenación:
  - Directos: Selección directa
  - Rápidos: Fusión

2. Algoritmos de ordenación

- Ordenación por selección directa:

Dados **N** elementos,

  - 1. Seleccionar el elemento de clave mínima
  - 2. Intercambiarlo con el primero del vector.
  - Repetir estas dos operaciones con los **N-1** elementos restantes hasta que quede un solo elemento.

- Necesitamos dos iteraciones:
  - Exterior, el elemento que vamos a intercambiar (de **0** a **N-2**).
  - Interior, el mínimo del subvector restante.

2. Algoritmos de ordenación

```
void seleccion(elemento v[N]){
   int i, j, min ;
   elemento aux ;
   for (i=0; i<N-2; i++){
      min = i ;
      for (j=i+1; j<N; j++)
         if (v[j].clave < v[min].clave)
            min = j;
      aux = v[i] ;
      v[i] = v[min] ;
      v[min] = aux ;
   }
}
```

2. Algoritmos de ordenación

- Ordenación por fusión:

Antes de ver el algoritmo vamos a ver el algoritmo de mezcla natural de dos secuencias ordenadas.

- Sean dos vectores **A** y **B** de dimensión **N** ordenados ascendentemente. Se trata de construir un vector **C** de dimensión **2*N** que contenga los elementos de **A** y **B** conservando el orden.

2. Algoritmos de ordenación

```
void mezcla(elemento a[N], elemento b[N],
                                    elemento c[N*2]) {
      int f, i, j, k ;
      i = 0 ;
      j = 0 ;
      k = 0 ;
      while ((i<N) && (j<N)) {
            if (a[i].clave < b[j].clave) {
              c[k] = a[i] ;
              i++ ;
              k++ ;
             }
```

2. Algoritmos de ordenación

```
        else
          if (a[i].clave > b[j].clave) {
            c[k] = b[j] ;
            j++ ;
            k++ ;
          }
          else {
            c[k] = a[i] ;
            c[k+1] = b[j] ;
            i++ ;
            j++ ;
            k = k+2 ;
          }
    }
```

2. Algoritmos de ordenación

```
        for (f=i; f<n; f++) {
                c[k] = a[f] ;
                k++ ;
        }
        for (f=j; f<n; f++) {
                c[k] = b[f] ;
                k++ ;
        }
}
```

2. Algoritmos de ordenación

- La eficiencia temporal del algoritmo de mezcla natural COSTE(N)=K*N

- Vamos a construir un algoritmo de ordenación basado en este algoritmo:

57	43	15	40	91	17	19	10
57	43	15	40	91	17	19	10
57	43	15	40	91	17	19	10
43	57	15	40	17	91	10	19
15	40	43	57	10	17	19	91
10	15	17	19	40	43	57	91

2. Algoritmos de ordenación

```
void ordfusion(elemento v[N],int ini, int fin) {
        int media ;
        if (ini < fin) {
                media = (ini+fin)/2 ;
                ordfusion(v, ini, media) ;
                ordfusion(v, media+1, fin) ;
                mezcla2(v, ini, fin) ;
        }
}
```

2. Algoritmos de ordenación

```
void mezcla2(elemento v[N], int ini, int fin) {
      int h, i, j, k, med ;
      elemento aux[N] ;
      med = (ini+fin)/2 ;
      i = ini ;
      j = med+1 ;
      k = 0 ;
      while ((i<=med) && (j<=fin)) {
         if (v[i].clave < v[j].clave) {
            aux[k] = v[i] ;
            i++ ;
            k++ ;
         }
```

2. Algoritmos de ordenación

```
            else
              if (v[j].clave < v[i].clave) {
                  aux[k] = v[j] ;
                  j++ ;
                  k++ ;
              }
              else {
                  aux[k] = v[i] ;
                  aux[k+1] = v[j] ;
                  i++ ;
                  j++ ;
                  k = k+2 ;
              }
    }
```

2. Algoritmos de ordenación

```
        for (h=i; h<=med; h++) {
              aux[k] = v[h] ;
              k++ ;
        }
        for (h=j; h<=fin; h++) {
              aux[k] = v[h] ;
              k++ ;
        }
        for (i=ini; i<=fin; i++)
              v[i] = aux[i] ;

}
```

2. Algoritmos de ordenación

- La eficiencia temporal del algoritmo es

$$COSTE(N) = 2K*N = K'*N$$

- La eficiencia temporal del algoritmo de ordenación, como está resuelto recursivamente, usando las relaciones de recurrencia, es:

$$COSTE(N) = K \qquad \text{para } N=1$$
$$COSTE(N) = 2COSTE(N/2)+K'N+K'' \qquad \text{para } N>1$$

2. Algoritmos de ordenación

$$COSTE(N) = 2COSTE(N/2)+K'N+K'' =$$

$$2COSTE(N/2^2)+K'N/2+K''$$

$$= 2^2COSTE(N/2^2)+2K'N+3K'' =$$

$$2COSTE(N/2^3)+K'N/2^2+K''$$

$$= 2^3COSTE(N/2^3)+3K'N+7K'' = \ldots$$

$$= 2^iCOSTE(N/2^i)+iK'N+(2^i-1)K'' =$$

2. Algoritmos de ordenación

$= 2^i COSTE(N/2^i)+iK'N+(2^i-1)K'' =$

$N/2^i = 1 \quad \Rightarrow \quad i = \log_2(N)$

$= KN+K'N \log_2 N+(N-1)K'' =$

$= K'N \log_2 N+(K+K'')N-K''$

$\mathbf{O(N \log_2 N)}$

Tema 8: Gestión de Ficheros

Contenido

1. Introducción

Hasta este tema, la única E/S ha sido sobre el teclado y monitor. En este tema se revisarán las funciones de librerías que nos permiten comunicar con los dispositivos de almacenamiento secundario.

¿Por qué se requiere esta comunicación?

Normalmente por el manejo de gran cantidad de datos en diferentes ejecuciones.

2. Ficheros de texto y binarios

Un fichero es un conjunto de datos almacenados en un dispositivo de almacenamiento secundario, independientemente de la información que guarde.

Es una cadena de *bytes* consecutivos terminada por el carácter especial EOF. Estos bytes se interpretan según la forma en la que se agruparon al crear los ficheros: como números reales, o enteros, o caracteres, o direcciones de memoria, etc.

2. Ficheros de texto y binarios

Hay básicamente dos formas en las que se puede almacenar información en un fichero: como caracteres de texto o como información binaria.

En ambos casos tratamos de secuencias de bytes acabadas en EOF. Sólo cambia la forma en la que el programa que trata el fichero va a interpretar la información contenida en él.

3. Ficheros de acceso secuencial y aleatorio

Son formas de acceso a esa información y se usan para cualquier tipo de archivo.

El acceso secuencial implica acceder a los diferentes datos uno tras otro empezando al principio. Sólo se puede insertar al final.

El acceso aleatorio permite acceder a cualquier dato de forma aleatoria.

4. Ficheros de texto

En este caso, los bytes del fichero se interpretan como caracteres acabados con el carácter EOF. La información almacenada se puede ver con cualquier editor de textos.

4. Ficheros de texto

Sea cual sea el tipo de fichero y su forma de acceso, actúan como almacenes de información y como tales es necesario ABRIRLOS y CERRARLOS.

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

FILE * arch;
arch = fopen ("c:\\tmp\\prueba.txt","r") ;
```

4. Ficheros de texto

A partir de la apertura del fichero se usará el puntero que devuelve la función `fopen` y que identifica al fichero (definido en `<stdio.h>`).

Tipo	Significado
"r"	Abrir un archivo existente sólo para lectura
"w"	Abrir un archivo sólo para escritura. Si existe, será destruido. Si no existe, será creado nuevo
"a"	Abrir archivo existente para añadir al final. Si no existe se crea
"r+"	Abrir archivo existente para lectura y escritura
"w+"	Abrir archivo nuevo para lectura y escritura. Si existe, será destruido
"a+"	Abrir archivo nuevo para leer y añadir. Si no existe se crea uno nuevo

4. Ficheros de texto

```
#include <stdio.h>
main () {
        /* Declaracion */
        FILE *arch ;

        /* Apertura */
        arch = fopen("p.txt","r") ;

        /* Operaciones sobre el archivo */

        /* Cierre */
        fclose(arch) ;
        return ;
}
```

4. Ficheros de texto

La lectura de un carácter se realiza usando la siguiente función:

```
int fgetc (FILE * pf)
```

```
#include <stdio.h>
main () {
        FILE * arch ; /* Declaración */
        char ch ;

        arch = fopen ("p.txt", "r") ; /* Apertura */

        while ((ch = fgetc(arch)) != EOF)
              printf ("%c", ch) ;
        fclose (arch) ; /* Cierre */
        return ;
}
```

4. Ficheros de texto

La escritura de un carácter se realiza usando la siguiente función:

```
int fputc (int c, FILE * pf)
```

```
#include <stdio.h>
main () {
      FILE *arch ;                  /* Declaracion */
      char vector[] = "prueba" ;
      int i ;
      arch = fopen ("p.txt","w") ; /* Apertura */
      i = 0 ;
      while (vector[i] != '\0') {
            fputc (vector[i], arch) ;
            i++ ;
      }
      fclose (arch) ;               /* Cierre */
      return ;
}
```

4. Ficheros de texto

También está permitido leer y escribir más de un carácter en el fichero. Las funciones a usar son:

```
int fscanf (FILE * pf, char * format, [pointer,])
int fprintf (FILE * pf, char * format, [arg,])
```

4. Ficheros de texto

```
#include <stdio.h>

main () {

        FILE * arch ;                          /* Declaración *
        char vector[] = "prueba" ;

        arch = fopen ("p.txt","w") ;   /* Apertura */

        fprintf (arch, "%s", vector) ;

        fclose (arch) ;                         /* Cierre */

        return ;

}
```

4. Ficheros de texto

Escritura de datos de diferentes tipos sobre un fichero de texto.

4. Ficheros de texto

```
#include <stdio.h>
main () {
   FILE * arch ;              /* Declaración */
   float precio ;
   int unidades ;
   char pieza[50] ;

   printf ("Introduce pieza, cantidad y precio: ") ;
   scanf ("%s%d%f", pieza, &unidades, &precio) ;
   arch = fopen ("p.txt","w") ;        /* Apertura */
   fprintf (arch,"%s %d %f ",pieza,unidades,precio) ;
   fclose (arch) ;             /* Cierre */

   return ;
}
```

5. Ficheros binarios

Las operaciones de apertura y cierre de estos archivos son las mismas que antes: **`fopen`** y **`fclose`**.

También los modos de acceso son los mismos... añadiéndoles una b: **`rb`**, **`wb`**, **`ab`**, **`rb+`** ...

Para *lectura* y *escritura* se usan las funciones **`fread`** y **`fwrite`**. Estas funciones leen y escriben bytes:

```
size_t fread(void *p, size_t size, size_t n, FILE *pf)

size_t fwrite(void *p,size_t size, size_t n, FILE *pf)
```

5. Ficheros binarios

donde :

- El puntero `p` apunta a la variable en la cual se guardan/recogen los datos a tratar.
- El argumento `size` es el número de bytes de los datos implicados.
- `N` es el número de datos a tratar.
- El último es el descriptor del fichero.

5. Ficheros binarios

```
#include <stdio.h>

main () {
  FILE * org ;                          /* Declaración */
  float t[4], v[] = {1.56, 4e-4, 32.01, 0.21e1} ;
  org = fopen ("p.txt","wb") ;          /* Apertura */
  fwrite (v, sizeof(float), 4, org) ;
  fclose (org) ;

  org = fopen ("p.txt","rb") ;          /* Apertura */
  fread (t, sizeof(float), 4, org) ;
  fclose (org) ;

  printf("Numeros: %f %f %f %f\n",t[0],t[1],t[2],t[3]) ;
  return ;
}
```

6. Acceso aleatorio a ficheros

Acceso secuencial: Sólo permite leer datos desde el principio o escribir datos al principio o al final del mismo.

Para poder acceder a cualquier parte del fichero se usan unas funciones especiales... El fichero es el mismo y se crea de la misma forma.

Al abrir un fichero, se devuelve un campo de valor 0 que referencia a qué byte del fichero se está accediendo.

6. Acceso aleatorio a ficheros

Existe una función para modificar ese valor y hacer que apunte a donde queramos:

```
fseek(FILE *pf, log int desplazamiento, int modo)
```

El modo nos dice a partir de dónde se hace el desplazamiento:

- **SEEK_SET**: Desde el principio.
- **SEEK_CUR**: A partir de la posición actual.
- **SEEK_END**: A partir del final.

Tema 9: Variables Dinámicas

Contenido

- 1. Introducción: punteros
- 2. Listas Enlazadas:
  Listas enlazadas y ordenadas
- 3. Pilas
- 4. Colas

1. Introducción

ESTÁTICO vs DINÁMICO: En el primer caso, se reserva el espacio por el programador en compilación. En el segundo, se va ocupando a medida que se necesita.

Una variable de tipo *puntero* permite almacenar la dirección de otro dato. Las variables que almacenan los datos a las que un puntero referencia se llaman variables dinámicas.

1. Introducción

Supongamos que **v** es una variable que representa un determinado dato. El operador de dirección & da la dirección de un objeto:

```
p = &v ;
```

asigna la dirección de v a la variable **p**: se dice "**p** apunta a **v**".

Este operador sólo se aplica a objetos que están en memoria.

1. Introducción

La variable **p** debe declararse como un puntero a elementos del tipo de la variable v. Por ejemplo:

```
int * p, v ;
```

Declaramos **v** como un entero y **p** como un puntero a entero. Es decir, **p** puede referenciar posiciones de memoria que almacenen enteros. Veremos que el símbolo * es usado de forma diferente fuera de la declaración.

1. Introducción

El operador de indirección `*` cuando se aplica a un puntero, da acceso al objeto al que señala el puntero: Así `*p` es el contenido de la posición de memoria a la que apunta `p`.

Al declarar una variable como puntero, por defecto se debe decir que apunta a `NULL`, es decir, a ningún sitio. Esta variable está definida en `<stdlib.h>`.

1. Introducción

Supongamos la siguiente declaración de variables:

```
int x, y, *ip, z[10] ;
```

```
x = 1 ;
ip = &x ;
y = *ip ;
*ip = 0 ;
ip = &z[0] ;
*ip = 7 ;
```

`ip` apunta a `x`

`y` toma el valor `1`

`x` vale `0`

`ip` apunta ahora a `z[0]`

`z[0]` vale `7`

1. Introducción

Toda variable puntero debe ser inicializada previamente antes de su uso. Así el siguiente ejemplo NO es correcto:

```
int v, *pv ;
v = 8 ;
*pv = v ;
```

```
int v, *pv ;
v = 8 ;
pv = &v ;
```

¿Es lo que busco?

1. Introducción

En los ejemplos vistos hasta ahora, NO se ha creado dinámicamente una variable, sino que se ha hecho que un puntero apunte a un lugar predefinido.

Las variables en cuestión se van a llamar *dinámicas* porque se crean en ejecución. Para eso necesitamos algún mecanismo que reserve la memoria según se necesite.

1. Introducción

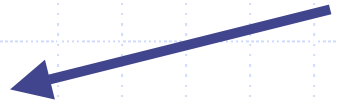
Este mecanismo es la función:

```
void * malloc (numero_bytes_memoria)
```

Está definida en el archivo de cabecera **<stdlib.h>**. Veamos cómo calcular el número de bytes en memoria necesarios.

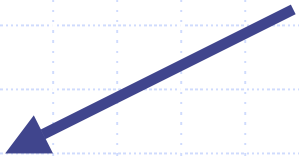
1. Introducción

```
#include <stdlib.h>

main() {
        int *dato_simple;
        dato_simple = (int *) malloc(sizeof(int));
        *dato_simple = 89;
        printf("El valor del puntero es: %d\n",*dato_simple);
        free(dato_simple)
        return;
}
```



1. Introducción

```
#include <stdlib.h>

struct persona {
        float peso;
        int edad;
};

main() {
        struct persona *p1, *p2;
        p1 = (struct persona *) malloc(sizeof(struct persona));
        p2 = (struct persona *) malloc(sizeof(struct persona));

        p1->peso = 80.9;
        p1->edad = 20;

        (*p2).peso = 74.3;
        (*p2).edad = 23;

        return;
}
```



2. Listas enlazadas

¿Qué ocurre cuando queremos generar una estructura dinámica de la que no conocemos el tamaño exacto?

Supongamos que tenemos un archivo de un número no determinado de personas y tenemos que sacar un listado alfabético de sus componentes.

2. Listas enlazadas

La estructura podría ser:

```
struct persona {
  char nombre[50] ;
  int edad ;
} ;
```

2. Listas enlazadas

Manipular el archivo directamente no se puede. Parece lógico volcar la información a una estructura intermedia ...

¿Cuál? ¿Un vector?

"No se puede" porque no se qué tamaño tiene.

Aparece la necesidad del uso de una lista enlazada.

2. Listas enlazadas

En el caso de usar una lista enlazada, necesitamos "algo" que nos permita enganchar toda la estructura ...

Ese algo es un puntero interno en la estructura.

```
struct persona {
  char nombre[50] ;
  int edad ;
  struct persona * sig ;
} ;
```

2. Listas enlazadas

Vamos a trabajar con una *lista enlazada* simple. En este caso, por ejemplo, la información es un entero. La estructura es:

```
struct reg {
  int dato ;
  struct reg * sig ;
} ;
```

2. Listas enlazadas

Operaciones sobre listas enlazadas:

1. Crear una lista vacía
2. Ver si un elemento se encuentra o no en la lista
3. Recorrer la lista completa
4. Insertar en cabeza de la lista
5. Insertar al final de la lista
6. Borrar el primer elemento de la lista
7. Borrar el último elemento de la lista

2. Listas enlazadas

1. Creación de la lista vacía:

```
struct reg * lst ;
/* Declaración del puntero externo */

void crea (struct reg **) ;
/* Prototipo de la función */

crea (&lst) ;
/* Llamada a ejecución */
```

2. Listas enlazadas

El código de la función es:

```
void crea ( struct reg ** lis) {
      *lis = NULL ;
 }
```

2. Listas enlazadas

2. Decir si un elemento cualquiera está en la lista:

```
int pertenece (struct reg *, int) ;
```

/ Prototipo de la función */*

```
if (! pertenece (lst, i)) …
```

/ Ejemplo de uso */*

2. Listas enlazadas

El código de la función es:

```
int pertenece (struct reg *li, int i) {
      struct reg * actl ;
      int enc ;
      actl = li ;
      enc = 0 ;
      while (actl != NULL &&  ! enc) {
            enc = (actl->dato == i) ;
            actl = actl->sig ;
      }
      return enc ;
}
```

2. Listas enlazadas

3. Realizar alguna operación sobre todos los elementos de la lista, por ejemplo sacarlos por pantalla:

```
void imprime (struct reg *) ;
```

/ Prototipo de la función */*

```
imprime (lst) ;
```

/ Ejemplo de uso */*

2. Listas enlazadas

```
void imprime (struct reg * lis) {

        while (lis != NULL) {
                printf ("%d\n", lis->dato) ;
                lis = lis->sig ;
        }
}
```

2. Listas enlazadas

4. Insertar un elemento en la cabeza de la lista:

```
void inscab (struct reg **, int) ;
```

/ Prototipo de la función */*

```
inscab (&lst, i) ;
```

/ Ejemplo de uso */*

2. Listas enlazadas

```
void inscab (struct reg ** lis, int i) {

    struct reg * p ;

    p = (struct reg *)malloc(sizeof(struct reg));
    p->sig = *lis ;
    p->dato = i ;
    *lis = p ;
}
```

2. Listas enlazadas

5. Insertar un elemento al final de la lista:

```
void insfin (struct reg **, int) ;
```

/ Prototipo de la función */*

```
insfin (&lst, i) ;
```

/ Ejemplo de uso */*

2. Listas enlazadas

```
void insfin (struct reg ** lis, int i) {

 struct reg *p ;

 if (*lis != NULL)
      insfin (&((*lis)->sig), i) ;
 else {
    p = (struct reg *) malloc(sizeof(struct reg));
    p->sig = NULL ;
    p->dato = i ;
    *lis = p ;
 }
}
```

2. Listas enlazadas

6. Borrar el primer elemento de la lista:

```
void borrarpri (struct reg ** lis) {

     struct reg * aux ;

     if (! lista_vacia (*lis) {
          aux = *lis ;
          *lis = *lis->sig ;
          free (aux) ;
     }
}
```

2. Listas enlazadas

7. Borrar el último elemento de la lista:

```
void borrarult (struct reg ** lis) {
   struct reg * aux1, * aux2 ;
   if (! lista_vacia (*lis))
      if (*lis->sig == NULL) {
         free (*lis) ;
         *lis = NULL;
      }
      else {
         aux1 = *lis ; aux2 = NULL;
         while (aux1->sig != NULL)
            { aux2 = aux1; aux1 = aux1->sig ; }
         aux2->sig = NULL ;
         free (aux1) ; }
}
```

2. Listas enlazadas ordenadas

Tenemos dos opciones principales:

- Crear la lista y después ordenarla.
- Crearla ya ordenada.

Respecto a la ordenación de la lista, siempre hay que usar un método de ordenación, pero, al hacerlo (2 opciones):

Movemos sólo la información de los nodos.

Movemos los nodos.

2. Listas enlazadas ordenadas

Veamos el caso de crear la lista de forma ordenada. Será la forma habitual de resolverlo. Los métodos a usar serán prácticamente los mismos que el de creación de listas.

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

struct reg {
   int dato ;
   struct reg * sig ;
} ;

main() {

   struct reg *lst, *q;
   int i;

   void crea (struct reg **) ;
   int pertenece (struct reg *, int, struct reg **);
   void imprime (struct reg *) ;
   void insfin (struct reg **, int) ;
   void inscab (struct reg **, int) ;
```

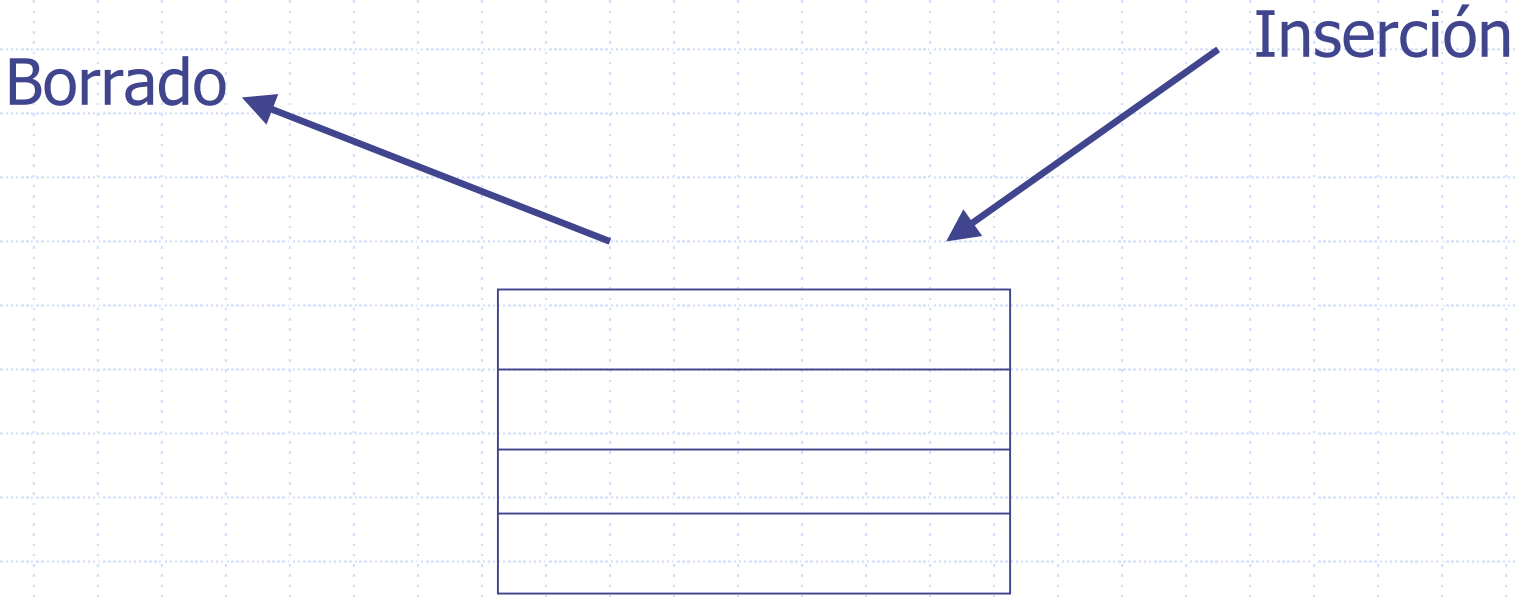
```
        crea (&lst) ;
        printf ("Dame el valor del entero: ") ;
        scanf ("%d", &i) ;
        while (i != 50) {
              if (! pertenece (lst, i, &q))
                    if (q == NULL)
                          inscab (&lst, i) ;
                    else
                          inscab (&(q->sig), i) ;
              printf ("Dame el valor del entero: ") ;
              scanf ("%d",&i) ;
        }
        imprime (lst) ;
}
```

```
int pertenece (struct reg *l, int i, struct reg **q)
   int enc ;
   struct reg *p ;

   p = l ;
   *q = NULL ;
   enc = 0 ;
   while (p != NULL && ! enc) {
      enc = (p->dato >= i) ;
      if (! enc)  {
         *q = p ;
         p = p->sig ;
      }
   }
   if (p == NULL && *q == NULL) return enc ;
   else if (p == NULL) return enc ;
      else if (p->dato == i) return enc ;
         else return (! enc) ;
}
```

```
void inscab (struct reg **lis, int i) {

   struct reg *p;

   p = (struct reg *) malloc(sizeof(struct reg)) ;
   p->sig = *lis ;
   p->dato = i ;
   *lis = p ;
}
```

3. Pilas

Una pila es una estructura de datos en la que inserciones y borrados se realizan por el mismo extremo.



3. Pilas

Para representar esta estructura de datos usaremos una lista enlazada donde inserciones y borrados se hagan por el mismo extremo.

En el caso de las pilas a las operaciones se les llama *push* y *pop* (apilar y desapilar) respectivamente.

```
#include <stdio.h>
#include <stdlib.h>
  struct pila {
     int valor ;
     struct pila *sig ;
  } ;
main() {
  struct pila *pl ;
  int i ;
  void crea (struct pila **) ;
  void push (struct pila **, int) ;
  int pop (struct pila **) ;

  crea (&pl) ;
  for (i=0; i<5; i++)
     push (&pl, i) ;
  for (i=0; i<5; i++)
     printf ("Valor: %d\n", pop (&pl)) ;
}
```

3. Pilas

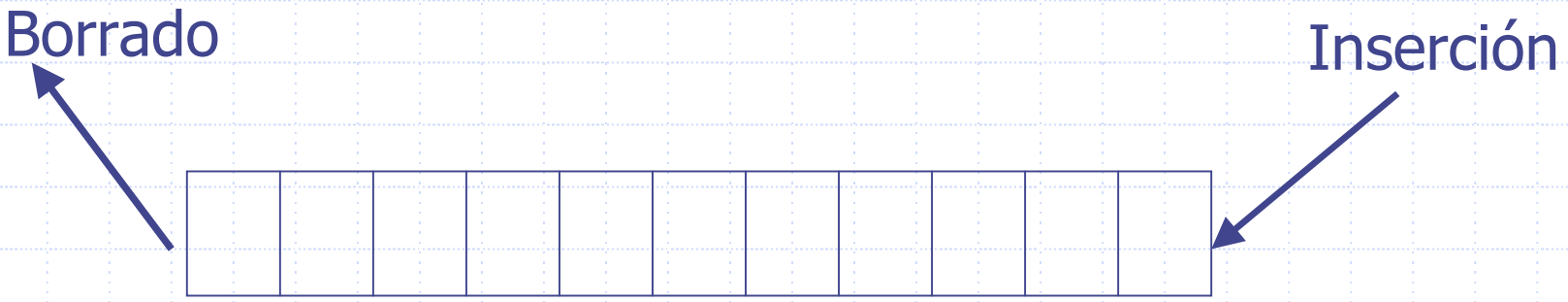
```
void crea ( struct pila **pl) {
   *pl = NULL ;
}

void push (struct pila **pl, int i) {
   struct pila *p ;

   p = (struct pila *) malloc(sizeof(struct pila)) ;
   p->sig = *pl ;
   p->valor = i ;
   *pl = p ;
}
```

3. Pilas

```
int pop (struct pila **pl) {
        struct pila *p ;
        int v ;

        p = *pl ;
        *pl = (*pl)->sig ;
        v = p->valor ;
        free(p) ;

        return v ;
}
```

4. Colas

Es una estructura de datos en la que las inserciones se hacen por un extremo y los borrados por el otro.



4. Colas

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

struct nodo {
        int valor ;
        struct nodo *sig ;
} ;

struct cola {
        struct nodo *primero ;
        struct nodo *ultimo ;
} ;
```

4. Colas

```
main() {
        struct cola cl ;
        int i ;

        void crea (struct cola *) ;
        void ins (struct cola *, int) ;
        int extrae (struct cola *) ;

        crea (&cl) ;
        for (i=0; i<5; i++)
              ins (&cl, i) ;
        for (i=0; i<5; i++)
              printf("Valor: %d\n", extrae (&cl)) ;
}
```

4. Colas

```
void crea (struct cola *cl) {

                cl->primero = NULL ;
                cl->ultimo = NULL ;
}
```

4. Colas

```
void ins (struct cola *cl, int i) {
   struct nodo *n ;

   n = (struct nodo *) malloc(sizeof(struct nodo)) ;
   n->valor = i ;
   n->sig = NULL ;
   if (cl->primero == NULL) {
         cl->primero = n ;
         cl->ultimo = n ;
   }
   else {
         cl->ultimo->sig = n ;
         cl->ultimo = n ;
   }
}
```

4. Colas

```
int extrae (struct cola *cl) {

    struct nodo *n ;
    int v ;

    n = cl->primero ;
    cl->primero = (cl->primero)->sig ;
    v = n->valor ;
    free (n) ;
    if ((cl->primero) == NULL)
          cl->ultimo = NULL ;

    return v ;
}
```