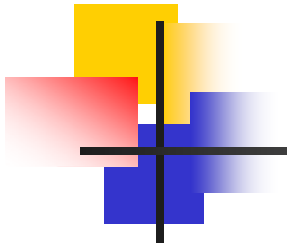


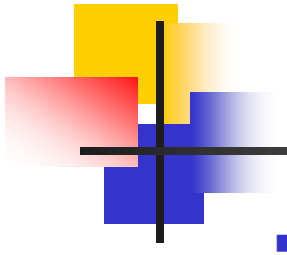
Índice

- Introducción a los lenguajes de programación.
  - Tratamiento automático de la información
  - Esquema de un computador
  - Concepto de programa. Niveles de abstracción de un programa.
- Repaso de C:
  - Tipos de variables.
  - Declaracion de variables
  - Sentencias de control
  - Funciones. Paso de argumentos a una función.
  - Punteros, operaciones con punteros, punteros y arrays.
  - Asignación dinámica de memoria.
    - Memoria en tiempo de compilacion.
    - Memoria en tiempo de ejecución.

Introducción a los lenguajes de programación

- Informática: Tratamiento automático de la información.
- Dados unos datos de entrada, se realiza una serie de operaciones sobre ellos y obtenemos unos datos de salida.
- Nos centraremos en la representación de ese algoritmo usando el lenguaje de programación C.

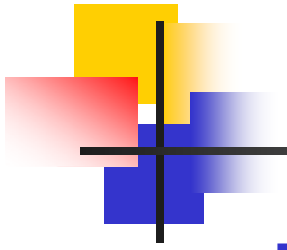
Datos de Entrada → **Algoritmo** → Datos de Salida



Introducción a los lenguajes de programación

- Nuestro objetivo es codificar el algoritmo para que se ejecute sobre un computador utilizando un lenguaje de programación.
- Definiremos:

- Computador: Lo definiremos como un sistema capaz de almacenar y procesar información. En este, podemos distinguir normalmente una serie de elementos:
- Procesador o CPU: Encargado de realizar operaciones aritméticas y lógicas y de coordinar el funcionamiento de los demás componentes.
- Memoria principal o RAM: Componente del ordenador donde se guardan los datos y los programas que la CPU está utilizando. Volátil, el contenido desaparece al apagar la alimentación.
- Disco duro: Elemento de almacenamiento masivo de la información. En soporte magnético generalmente. En el disco duro se almacenan los programas. De ahí se pueden cargar en memoria principal antes de ser ejecutados.
- Periféricos: Pantalla, ratón, teclado... elementos de interfaz humana.



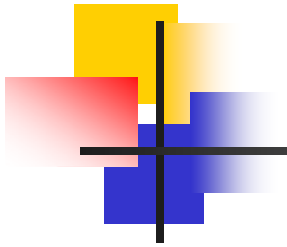
Introducción a los lenguajes de programación

- Codificación de un algoritmo, formas de representar o codificar un algoritmo:
  - Mediante **lógica cableada**. Los circuitos lógicos está diseñados para realizar unas determinadas operaciones. Ej. calculadoras. Ventaja: gran rapidez. Desventaja: No es modificable.
  - Mediante **lógica programada**: Se utiliza una CPU. Programa: Esta unidad es capaz de procesar una serie de instrucciones de manera secuencial (consecutiva) para procesar la información. El objetivo que perseguimos es generar ese conjunto de instrucciones para que el procesador realice la tarea que se desea.

Introducción a los lenguajes de programación

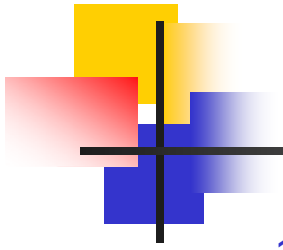
- Vamos a distinguir diferentes **niveles de abstracción**:
  - **Lenguaje máquina:** Nativo del procesador. Códigos binarios especiales que representan la instrucción que debe realizar la CPU. Claramente, codificar en lenguaje máquina un algoritmo puede llegar a ser tedioso y llevar mucho tiempo.
  - **Lenguaje ensamblador**: Es un nivel superior de abstracción, se utilizan nemónicos asociados a la instrucción del procesador. Ejemplo. Juego de instrucciones de los procesadores Intel Pentium. El ensamblador es un programa encargado de traducir el lenguaje ensamblador a lenguaje de máquina del procesador y es es específico para cada tipo de procesador. En general, los programas realizados en ensamblador son difíciles de leer y de encontrar los errores. Como ventaja frente a los lenguajes de programación: La CPU hace exactamente lo que se le indica, y el código puede llegar a ser muy rápido.

Opcode, código máquina	Instrucción	Descripción
04 Hex 000 100	ADD AL,imm8	Sumar el byte (imm8) y el contenido del registro AL, y guardar el resultado en AL.
05 Hex 000 101	ADD AX,imm16	Sumar la palabra (imm16) y el contenido del registro AX y guardar el resultado en AX.
…		



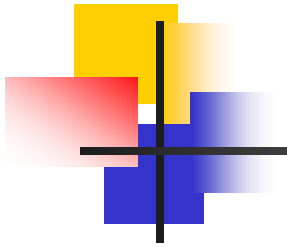
Introducción a los lenguajes de programación

- **Lenguaje de alto nivel:** Son un salto hacia adelante en abstracción. El lenguaje está orientado a la resolución de problemas y intenta independizarse del procesador. Ejemplos: Lenguaje C, FORTRAN, PASCAL.
- Características:
  - Son más cercanos al lenguaje natural. P. e. repite esto mientras la variable i sea menor de 100. while(i<100){ ...}.
  - Se utilizan una serie de palabras reservadas para este propósito. Ej. **while**, **for**, **int**, **long**, **float**, **struct** ...
  - Se utiliza una serie de sentencias de control que nos permiten definir de manera más fácil el programa. La utilización de identificadores para las variables nos permite obviar la dirección en la que se esta se almacena.
  - En este caso el compilador es el encargado de traducir el fichero del programa (.c, .cpp) en un programa ejecutable (.exe en Windows). Este proceso viene complementado con una etapa de *linkado*, que enlaza con las librerías que hayamos utilizado. P. e. librería <stdio.h>, que han sido compiladas previamente.



Índice

1. Introducción a los lenguajes de programación.
   - Tratamiento automático de la información
   - Esquema de un computador
   - Concepto de programa. Niveles de abstracción de un programa.
2. Repaso de C:
   - Tipos de variables.
   - Declaracion de variables
   - Sentencias de control
   - Funciones. Paso de argumentos a una función.
   - Punteros, operaciones con punteros, punteros y arrays.
   - Asignación dinámica de memoria.
     - Memoria en tiempo de compilacion.
     - Memoria en tiempo de ejecución.



Repaso de lenguaje C

1) **Conocer los recursos** que nos ofrece el lenguaje C y C++ ... gramática del lenguaje.

Tipos de datos: char, int, long

Control de flujo: if, if... else, for, while

Funciones de librería estándar: printf, scanf...

2) Pero, por ejemplo, conocer la gramática inglesa no equivale a saber hablar inglés. Es necesario adquirir el **hábito** de realizar programas que tengan las siguientes características:

Rapidez: La llamada al programa resuelve el problema en un tiempo finito. Además, se deberá tener en cuenta el tiempo necesario para la programación.

Claridad: El programa esté escrito de forma clara. Generalmente se añaden comentarios que describen los pasos realizados.

Modularidad: El programa debe estar separado en funciones, cada una de ellas con un cometido en particular.

Portabilidad: El programa pueda ser compilado y ejecutado en diferentes computadores, funcionando con diferentes sistemas operativos (p.e. Windows y Linux).

3) **Tiempo + trabajo**

Repaso de lenguaje C: Tipos básicos

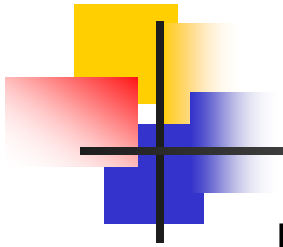
- **Tipos de datos**: El lenguaje C nos permite utilizar una serie de datos básicos (primitivos). La selección del tipo de dato a utilizar dependerá de las necesidades del problema y del tipo de datos a almacenar.

Tipo de dato	Descripción	Tam.memoria
int	Entero	2 bytes
char	Caracter	1 byte
unsigned int	Entero positivo	2 bytes
float	Número real	4 bytes
double	Número real de doble precision	8 bytes
...	...	...

C permite definir tipos de datos más complejos a partir de estos tipos básicos →Estructuras.

Diferente rango, dependiendo del tamaño. P.e. int: 216→[0-65535] unsigned y [-32768 32767] signed

El tamaño de cada tipo de dato en memoria depende del procesador, del compilador y del S.O. Ya que existen diferentes compiladores para procesadores diferentes.



Repaso de lenguaje C

Declaración de variables:

- Antes de utilizar una variable en C se debe declarar, lo que hace que se reserve un área en la memoria para almacenar un tipo de dato. Lo siguiente declara una variable de tipo entero que se llama num.

```
int num;
```

- El valor inicial de la variable es aleatorio. Deberemos asignarle un valor antes de utilizarla.

```
num = 42;
```

- Se pueden declarar al tiempo varias variables de tipo entero.

```
int num, Num=0, num_2, 2num;
```

Repaso de lenguaje C: Operadores

Operador Aritmético	Nombre	Descripción
=	Asignación	Var. A la izquierda toma valor de la derecha
+	Adición	Suma dos operandos
-	Sustracción	Resta dos operandos
*	Producto	Multiplica dos operandos
/	División	Divide dos operandos
%	Resto	Devuelve el resto de la división entera

Operador Comparación	Nombre	Descripción
==	Igualdad	A==B
!=	Desigualdad	A!=B
>	Mayor que	A > B
<	Menor que	A<B
>=	Mayor o igual	A>=B
<=	Menor o igual	A<=B
&&	AND	AND lógico
||	OR	OR lógico

Repaso de lenguaje C: Bifurcaciones

```
if(expresion)
{
    sentencia;
}

if(expresion)
{
    sentencia_1;
}
else
{
    sentencia_2;
}
```

```
if(expresion1)
{
    sentencia_1;
}
else if(expresion2)
{
    sentencia_2;
}
else if(expresion3)
{
    sentencia_3;
}
else
{   sentencia_3;
}
```

```
switch(expresion)
{
    case 1: sentencia_1;
        break;
    case 2: sentencia_2;
        break;
     ...
    case N: sentencia_N;
        break;
     default: sentencia;
        break;
}
```

Repaso de lenguaje C: Bucles

```
while(expresion)
{
    sentencia;
}
```

```
for(inicializacion; expresion_control; actualizacion)
{
    sentencia;
}
```

```
inicializacion;
while(expresion_control)
{
    sentencia;
    actualizacion;
}
```

Repaso de lenguaje C: Break, continue, goto

```
int i = 0;
while(i<100)
{
    printf(“%d\n”, i);
    i++;
    if(i==50)
        break;
}
printf(“Acabado el bucle while”);
```

```
int i = 0;
while(i<100)
{
      printf("%d %d %d ", i, i/2, i%2);
      if(i%2){
          i++;
          printf("\n");
          continue;
       }
     printf("Es par\n");
     i++;
}
```

Repaso de lenguaje C: Errores comunes

```
int i = 0;
while(i<100)
{
    printf(“%d\n”, i);
    i++;
    if(i=50)
        break;
}
printf(“Acabado el bucle while”);
```

```
int i = 0;
while(i<10)
{
    if(i=!0)
        printf("%d\n", i);
    i++;
}
printf("Acabado el bucle while");
```

1
2
3
4
5 ...

1
1
1
1
1 ...

Repaso de lenguaje C

```
void main(void)
{
        int x, y;
        printf(“\nIntroduzca x”);
        scanf(“%d”, &x);
        printf(“\nIntroduzca y”);
        scanf(“%d”, &y);
        if(x>y)
                printf(“\nEl maximo es x=%d”, x);
        else
                printf(“\nEl maximo es y=%d”, y);

        printf(“\nIntroduzca x”);
        scanf(“%d”, &x);
        printf(“\nIntroduzca y”);
        scanf(“%d”, &y);
        if(x>y)
                printf(“\nEl maximo es x=%d”, x);
        else
                printf(“\nEl maximo es y=%d”, y);
}
```

Repaso de lenguaje C: Funciones

- El código anterior es bastante largo. También, si queremos extender su funcionalidad es difícil de modificar.
- Las funciones permiten dividir un programa en modulos, más pequeños y manejables (subprogramas o funciones que son llamadas por el programa principal.)

Ventajas:

- **Modularización**: A Cada función se le asigna una misión concreta. No debe tener un número de líneas excesivo. Recibe unos parámetros de entrada, realiza una serie de cálculos y ofrece unos resultados.
- **Reutilización de código**: Cada función puede llamarse muchas veces desde un programa. Con lo que se reduce el número de líneas de código del programa. Además, se puede reutilizar en diferentes programas.

A la hora de escribir un programa en C, deberemos dividir el problema general en problemas más pequeños, y programarlos separadamente en funciones. La división de un programa en funciones no es única y depende del estilo de cada persona.

Repaso de lenguaje C: Funciones

```
#include <stdio.h>

void HallarMaximo(void); //Declaración, prototipo

void main(void)
{
	HallarMaximo(); //Llamada
	HallarMaximo();
}

void HallarMaximo(void) //Definición, implementación
{
	int x, y;
	printf("\nIntroduzca x");
	scanf("%d", &x);
	printf("\nIntroduzca y");
	scanf("%d", &y);
	if(x>y)
		printf("\nEl maximo es x=%d", x);
	else
		printf("\nEl maximo es y=%d", y);
}
```

Repaso de lenguaje C: Funciones, ejemplos

```
#include <stdio.h>

//Declaración,
   prototipo

void
   HallarMaximo(void)
   ;
```

```
void HallarMaximo(void) //Definición, implementación
{
      int x, y;
      printf("\nIntroduzca x");
      scanf("%d", &x);
      printf("\nIntroduzca y");
      scanf("%d", &y);
      if(x>y)
            printf("\nEl maximo es x=%d", x);
      else
            printf("\nEl maximo es y=%d", y);
}
```

Repaso de lenguaje C: Duración y visibilidad de las variables

En C/C++ vamos a distinguir:

- **Variables locales**: El especificador es ***auto*** (por defecto). Cada variable auto se crea al comenzar a ejecutarse el bloque y deja de existir al finalizar el bloque {...}. Las variables locales solo son visibles dentro del bloque en el que fueron declaradas. El valor inicial no está definido.

- **Variables globales**: Se declaran fuera de cualquier bloque y son visibles desde todo el código dentro del fichero C. Si en algún bloque se define una variable local con el mismo nombre, prevalece la local. Al crearse son inicializadas a cero.

- **Variables estáticas**: Especificador ***static***. La visibilidad es como las variables locales. Al crearse se inicializan a cero, pero mantienen su valor al finalizar el bloque en el que fueron definidas.

- **Variables de tipo registro**: Si se declara una variable como ***register***, se le indica al compilador que almacene esa variable en un registro del procesador.

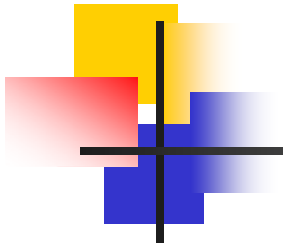
Repaso de lenguaje C: Duración y visibilidad de las variables

```
int func1(float a, float b); //Ojo a, b ??
int func2(int , int); //OK!

float x, y;
int ntot, k;
void main(void)
{
      int i, j, k=2;
      float a=2, b=2;

      func1(a,b);
      i=func1(a,x);
      j=func2(a,b);
      printf("Llamadas totales: %d", ntot);
     printf("Llam func1, func2: %d %d", i, j);
      printf("Valor x: %f", x);
      printf("Valor y: %f", y);
}
```

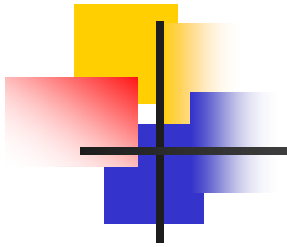
```
int func1(float c, float d)
{
      float c; //!!¿?
      printf("\nfunc1, valor de k: %d", k);
      static int n;
      x = c*d;
      n++;
      ntot++;
      return n;
}
int func2(int c, int d)
{
      float y;
      static int n;
      y= (float)c/d;
      n++;
      ntot++;
      return n;
}
```

```
func1, valor de k: 0
func1, valor de k: 0
Llamadas totales: 3
Llamadas func1: 2
Llamadas func2: 1
Valor x: 8.000000
Valor y: 0.000000
```

Repaso de lenguaje C: Duración y visibilidad de las variables



- Las variables globales pueden servir para comunicar ciertos valores entre funciones diferentes, sin tener que pasar explícitamente el parámetro a cada función.
- No obstante, es fácil que el programador declare otra variable local con el mismo nombre, enmascarando la global y, seguramente, produciendo un fallo en el funcionamiento del programa.
- Utilizar variables locales genera un código más fácil de leer generalmente.



Paso de argumentos a una función

- C permite pasar la dirección de una variable como argumento de una función.
- Esto posibilita cambiar la variable dentro del cuerpo de la función.
- Paso de argumentos a una función:
  - Paso por valor: Se le pasa el valor de la variable. Se altera una copia de la variable, no la variable original.
  - Paso por dirección: El contenido de la dirección puede ser cambiado. Se cambia la variable original.
- El paso por dirección permite a una función devolver más de un dato.

Ejemplo

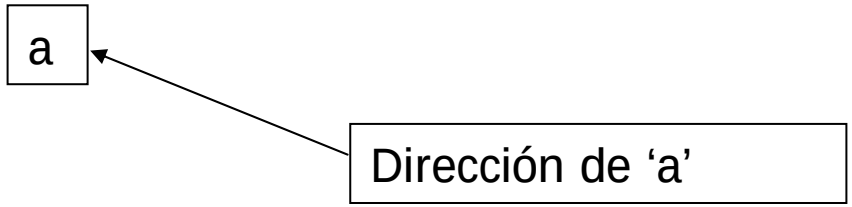
```
void func1(int x, int y); //prototipo de la función
void func2(int *px, int *px);

#include <stdio.h>

void main( void )
{
      int u = 3;
      int v = 4;

      func1(u, v);

      printf("\n u=%d, v=%d", u, v);

      func2(&u, &v);

      printf("\n u=%d, v=%d", u, v);
}
```

```
void func1(int a, int b)
{
    a = 0;
    b = 0;
    return;
}
void func2(int *pa, int *pb)
{
    *pa = 0;
    *pb= 0;
    return;
}
```

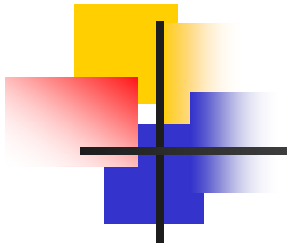
Salida:
u = 3, v = 4
u = 0, v = 0

Repaso de lenguaje C: Punteros

- Puntero: Variable que almacena la dirección de una variable en memoria.
  - Son de gran utilidad en C.



- Se debe distinguir entre la dirección de memoria a la que apunta la variable puntero y el contenido de esa dirección.
- En esa zona de memoria tendremos almacenado un dato.
- El número de bytes que ocupa una variable en memoria depende de su tipo. P.e. un caracter ASCII ocupa un byte. El compilador se encarga de estos detalles.



Repaso de lenguaje C: Declaracion de punteros

- Declaración de un puntero.

tipo-dato *pvar;

pvar: es el nombre de la variable puntero.

tipo-dato: es el tipo de variable al que puede apuntar (int, float, uint...)

Repaso de lenguaje C: Punteros, operadores

- Declaración de un puntero:

```
int *pentero;
```

- Se escribe un * delante del nombre del identificador (pentero).
- Se utilizan dos operadores con los punteros:
  - Operador dirección: & Da la dirección de un objeto.
  - Operador indirección: * Permite acceder al contenido de una dirección a la que apunta un puntero.
- Ejemplo:

```
int a, b; //Variable de tipo entero
int *pa; //Puntero a un entero

a = 5;
pa = &a; // pa apunta a la variable a.
b = *pa; // b vale 5
```

- & es el operador dirección. Opera sobre cualquier tipo de variable.
- * es el operador indirección. Solo puede actuar sobre operandos que sean punteros (variables puntero).

Repaso de lenguaje C: Ejemplo

```
#include <stdio.h>

void main( void )
{
    int u = 3, v;
    int *pu; //puntero a entero
    int *pv; //puntero a entero

    pu = &u;
    v = *pu;
    pv = &v;
    printf("\nu=%d &u=%X  pu=%X *pu =  %d", u, &u, pu, *pu);
    printf("\nv=%d &v=%X  pv=%X *pv =  %d", v, &v, pv, *pv);
}
```

Salida:

```
u = 3      &u = ff56      pu = ff56      *pu = 3
v = 3      &v = f67A      pv = f67A      *pv = 3
```

Repaso de lenguaje C: Punteros y arrays

- Recordemos que un array o vector es una agrupación de varias variables del mismo tipo de datos que se almacenan de forma consecutiva en memoria.

  float y[20];

- Debemos recordar que el nombre de un array es la dirección de su primer elemento.

```
int x[3]; //array de 3 enteros
int *puntero;

x[0]=10;
x[1]=20;
x[2]=30;

puntero = x;                    //Equivalentes
puntero = &x[0];

printf("\n%d", *puntero);
```

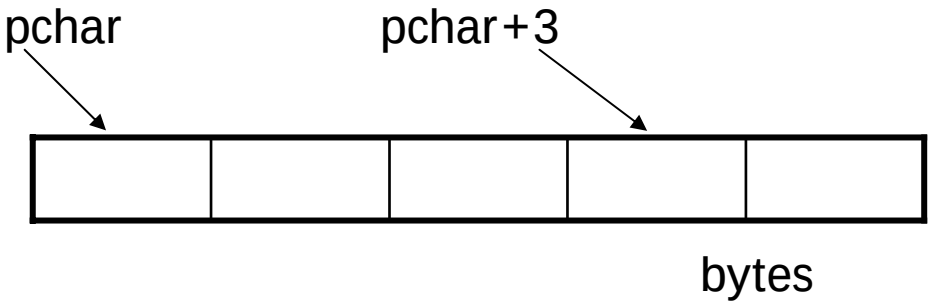
Repaso de lenguaje C: Aritmética de punteros

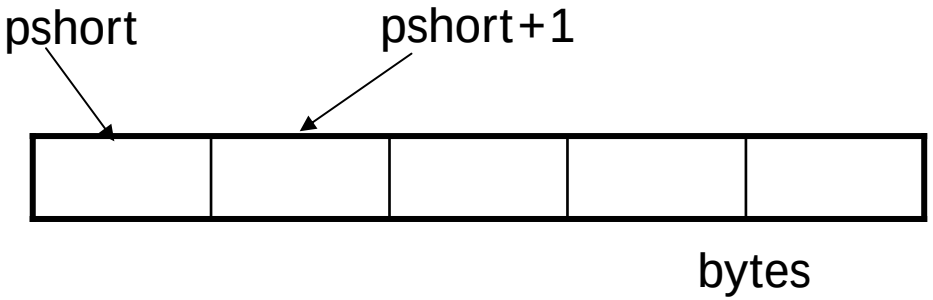
- Aritmética de punteros:
  - Suma o resta de un entero.

```
char *pchar;
pchar=pchar+3;
```

  - Incremento o decremento.

```
short *pshort;
pshort++;
```





- El número de bytes que ocupa un tipo de dato en memoria depende del compilador y de la máquina.

Repaso de lenguaje C: Punteros y arrays

- Podemos acceder a los elementos del array de la siguiente manera:

```
int x[3]; //array de 3enteros
int *puntero;

x[0]=10;
x[1]=20;
x[2]=30;

puntero = x;                    //Equivalentes
puntero = &x[0];

*puntero ←→ x[0]                //Equivalentes
 *(puntero+i) ←→ x[i]
puntero[i] ←→ x[i]

prinft("\n%d, %d, %d, %d, %d, %d", *puntero,  *(puntero+1), *(puntero+2), puntero[0], puntero[1], puntero[2]);

Salida:

10 20 30 10 20 30
```

Repaso de lenguaje C: Arrays de caracteres:

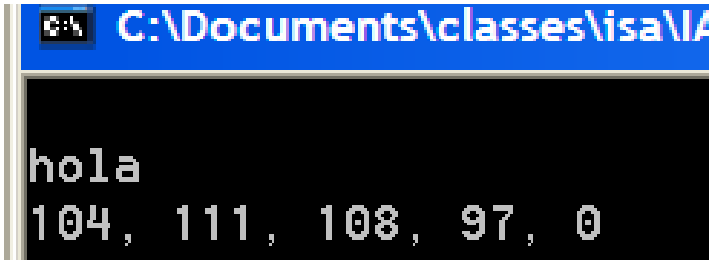
- Un array de caracteres se debe entender como un array de elementos de tipo char.

  char cade[20]=“hola";
- Para que el array de caracteres sea válido, su último elemento debe ser el carácter nulo ‘\0’, que es el valor 0 en código ASCII. El compilador hace esto automáticamente.

h	o	l	a	‘\0’

- En cada uno de los elementos del array se almacena el código ASCII. Lo podemos ver de la siguiente manera:

```
char cade[20]="hola";
printf("\n%s", cade);
printf("\n%d, %d, %d, %d, %d", cade[0],cade[1],
    cade[2], cade[3], cade[4]);
```



Repaso de lenguaje C: Asignación dinámica de memoria

- Cuando declaramos un array reservamos una cantidad fija de memoria.

```
float x[50]; // Array de 50 float
char cadena[100];//array de caracteres
```

- ¿Y si necesitamos una cantidad de memoria variable?
- La gestión dinámica de memoria se realiza mediante la función malloc (stdlib.h):

```
void *malloc(tamaño)
```

tamaño: es el tamaño en bytes de la memoria que se quiere reservar.

void *: malloc devuelve un puntero a un tipo genérico.

- Uso:

```
float *x; // declaramos un puntero
x = (float *) malloc(50*sizeof(float));
//reservamos memoria para 50 float
```

Ejemplo

```
#include <stdio.h>
#include <stdlib.h>
int main (void)
{
    int i, num;
    int* buffer;

    printf(“Cuantos enteros desea almacenar?");
    scanf("%d", &num);

    buffer = (int*)malloc(num*sizeof(int)); //reservamos la memoria

    if(buffer==NULL) //Si no se ha podido reservar la memoria, malloc devuelve
      return -1;     // el puntero NULL

    for(i=0; i<num; i++) {
      printf(“Elemento %d: %d\n", i);
      scanf(“%d", &buffer[i]);
    }

    free(buffer);//Liberamos la memoria reservada cuando no la necesitamos
    return 0;
}
```

Repaso de lenguaje C: Lo que no podemos hacer

- No reservar memoria:

```
float *px;
scanf("%f", px);
```

Explicación: px apunta a (contiene) una dirección de memoria aleatoria. La función scanf guarda un dato en esa zona de memoria. Deberemos poner:

```
float x;
scanf("%f", &x);
```

- Como norma general debemos recordar que declarar un puntero no nos reserva memoria.
- Normalmente debemos comprobar si la memoria se ha reservado:

```
buffer = (int*)malloc(num*sizeof(int)); //reservamos la memoria

if(buffer==NULL) //Si no se ha podido reservar la memoria, malloc devuelve NULL
      return -1;
```

Repaso de lenguaje C: Lo que no podemos hacer

- Errores al tratar con cadenas de caracteres:

```
char *cadena;
cadena = (char*)malloc(20*sizeof(char));  //o bien
char cadena[20];
```

¿Cuál de las siguientes es válida?

```
scanf("%s", &cadena);
scanf("%s", *cadena);
scanf("%s", cadena);
```

- Errores genéricos de variables y punteros. ¿Qué ocurre si hacemos?:

```
float x;
scanf("%f", x);
```

Respuesta: El compilador nos dará un error. Espera como segundo parámetro una dirección de memoria y le estamos pasando un número en coma flotante.

```
int y;
scanf("%d", y);
```

Respuesta: Ahora no hemos tenido suerte. La variable y es un entero, del mismo tamaño que una dirección de memoria, con lo que el compilador no se queja. Al ejecutar el código, scanf intenta guardar los datos en una zona de memoria, que no se le ha asignado.