

# Fundamentos de Software de Comunicaciones



# El lenguaje C

## Arrays, Cadenas, Estructuras y Punteros

# Contenidos

---

- Primer programa y salida de datos
- Arrays y estructuras
- Punteros y memoria dinámica
- Argumentos a funciones
- Punteros a funciones
- Punteros y arrays
- Cadenas de caracteres. Funciones
- Entrada de datos
- Main con argumentos
- Algunas funciones útiles

# Principales diferencias entre C y C++

- C es el lenguaje original con el que se creó Unix y la mayoría de dispositivos de comunicaciones se programan en él
  - Permite **acceso directo** a zonas de memoria
- Los lenguajes de C y C++ son muy compatibles
  - pero tienen distintas bibliotecas de funciones (y clases en el caso de C++)
  - Esta documentación está basada en el estándar **C99**. Ya hay un estándar C11 que lo acerca más a C++ (C++11/14)
- Principales diferencias al compilar:
  - ficheros .c (icon sintaxis C pura!) -> utiliza **gcc**
  - ficheros .cpp (con C++ y/o C) -> utiliza **g++**

# Primer ejemplo

```
1  #include <stdio.h>
2  ✓ int main()
3  {
4      printf("Hola\n");
5      return 0;
6  }
```

- Guardar en un fichero con extensión .c (por ejemplo: programa.c)
- Compilar con:
  - `gcc -o nombre_ejecutable programa.c -std=c99`
  - Opciones extra: `-Wall -WExtra`
- Ejecutar con:
  - `./nombre_ejecutable`

# Salida de datos en C

- Biblioteca de E/S:
  - `#include <stdio.h>`
- Salida por pantalla:

```
int x = 5;  
char hexa = 0xFF;  
char c = 'a';  
printf(" texto con comodines: %d %X %c \n", x, hexa, c);
```

# Salida de datos en C. Ejemplos

```
printf("Hola\n");  
printf("%s%c\n", "Hol", 'a');  
int x = 5;  
printf("%d %d\n", 35, x);  
char y = 0xFF; //notación hexadecimal  
printf("%x\n", y);  
printf("%p\n", &x); /*imprime la dirección de memoria de x*/
```

¿Qué sale por pantalla?

# Contenidos

---

- Primer programa y salida de datos
- **Arrays y estructuras**
- Punteros y memoria dinámica
- Argumentos a funciones
- Punteros a funciones
- Punteros y arrays
- Cadenas de caracteres. Funciones
- Entrada de datos
- Main con argumentos
- Algunas funciones útiles

# Los arrays de C

- Permiten la reserva de un número de posiciones consecutivas de memoria

- Declaración:

***tipo nombre[tamaño];*** //EL TAMAÑO ES **CONSTANTE**

- Inicialización:

***tipo nombre[T] = {c1, c2, ..., c\_n};***

- Siempre que T esté definido como **#define T 50**
- Ejemplo: ***int a[5] = {1, 2, 3};***

***tipo nombre[] = {c1, c2, ... c\_n};***

- Tamaño automático

NOTA: la asignación de un arrays a otro array es **ilegal** en C

```
char array1[50];
```

```
char array2[50];
```

```
array1 = array2; //ERROR (se verá la causa más adelante)
```

- Acceso a un elemento:

***nombre[índice]***

(donde índice está entre [0 , tamaño - 1])



# Resumen: arrays en C++ y en C

## ■ C++:

```
#include <tr1/array> // o <array> en C++11
const int TAM_ARRAY = 3;
//declaración con tamaño e inicialización
std::tr1::array<int, TAM_ARRAY> mi_array = {1,2,3};
//asignación elemento a elemento
mi_array[0] = 0;
//definición como tipo
typedef std::tr1::array<int, TAM_ARRAY> TipoArray;
TipoArray otro_array;
//consulta del tamaño del array (cuantos elementos tiene en uso)
otro_array.size();
```

## ■ C:

```
const int TAM_ARRAY = 3;
//declaración con tamaño e inicialización (¡de todas las casillas!)
int mi_array[TAM_ARRAY] = {1,2,3};
//declaración sin tamaño (lo resuelve el compilador)
int mi_array2[] = {1,2,3,4}; //este tiene cuatro elementos
//asignación elemento a elemento
mi_array[0] = 0;
//definición como tipo
typedef int TipoArray[TAM_ARRAY];
TipoArray otro_array;
//NO es posible consultar el tamaño ocupado de un array en C (lo cual es peligroso)
```

# Ejemplo de arrays de C

```
1  #include <iostream>
2  using namespace std;
3
4  int main(){
5      const int TAM_ARRAY = 10;
6      int arr[TAM_ARRAY];
7      int in;
8
9      for (int i=0; i < TAM_ARRAY ; ++i)
10         arr[i] = i; // inicializacion con valores
11
12     for (int i=0; i < TAM_ARRAY ; ++i)
13         cout << arr[i]; // impresion del array por pantalla
14
15     cout << "\nIntroduce el numero a buscar: " << endl;
16     cin >> in;
17
18     int j=0;
19     while( (j < TAM_ARRAY) && ( arr[j] != in) ){
20         j++;
21     }
22     if (j == TAM_ARRAY)
23         cout << "El numero " << in << "no esta en el array" << endl;
24     else
25         cout << "El numero " << in << "esta en la posicion " << j << endl;
26     return 0;
27 }
```

# Definición de estructuras

- ❑ Una estructura es una colección de variables (incluidas estructuras)

```
❑ struct nombre {  
    tipo nombre_campo;  
    tipo nombre_campo;  
    ...  
};
```

- ❑ En C/C++, una estructura puede utilizarse como una variable (asignación, paso como parámetro, valor de vuelta de una función)
- ❑ Las estructuras **no se pueden comparar**

# Tamaño de datos

- En C y C++ existe un operador unario ***sizeof*** que devuelve el número de bytes que ocupa una variable o un tipo de dato en memoria
  - La longitud la devuelve como natural positivo, en un tipo denominado ***size\_t***
- También se puede utilizar para calcular el tamaño en memoria de una estructura
  - **CUIDADO: SU VALOR PUEDE SER MAYOR QUE LA SUMA INDIVIDUAL DEL TAMAÑO DE SUS CAMPOS**

# Tamaño de datos: ejemplos

## ■ Ejemplos:

```
size_t longitud = sizeof(int);  
int var_entera; size_t longitud = sizeof(var_entera);
```

## ■ Tamaño de datos básicos:

- **char**: 1 byte
- **short**: 2 bytes
- **int**: 4 bytes
- **long**: 8 bytes
- **puntero**: 8 bytes (sistema de 64 bits)

# Tamaño de datos: ejemplos

```
struct X
{
    short s;
    int i;
    char c;
};
```

```
struct Y
{
    int i;
    char c;
    short s;
};
```

```
struct Z
{
    int i;
    short s;
    char c;
};
```

```
int main()
{
    struct X X;
    struct Y Y;
    struct Z Z;

    int sizeX = sizeof(X);
    int sizeY = sizeof(Y);
    int sizeZ = sizeof(Z);

    printf("X = %d; Y = %d, Z = %d\n", sizeX, sizeY, sizeZ);
}
```

■ ¿Qué sale por pantalla?

# Tamaño de datos: ejemplos

```
struct X
{
    short s;
    int i;
    char c;
};
```

```
struct Y
{
    int i;
    char c;
    short s;
};
```

```
struct Z
{
    int i;
    short s;
    char c;
};
```

## ■ Alineamiento en memoria y relleno (#)

○ struct X



○ struct Y



○ struct Z



# Tamaño de datos: alineamiento en memoria

## ■ Reglas de alineamiento de **structs**

- Antes de cada campo habrá el relleno necesario para que empiece en una dirección que es divisible por su tamaño
  - En un sistema de 64 bits como el nuestro, un `int` empieza en una dirección divisible por 4, `long` por 8 y `short` por 2.
- Los `char` son un tipo especial que puede ir en cualquier posición porque tienen tamaño 1
- El tamaño de un `struct` se alinea con respecto del tamaño de su campo más largo.

## ■ ¿Qué tamaño tendría estas estructuras?

```
struct Datos1
{
    short s;
    char  c;
};
```

```
struct Datos2
{
    long l;
    char c;
};
```



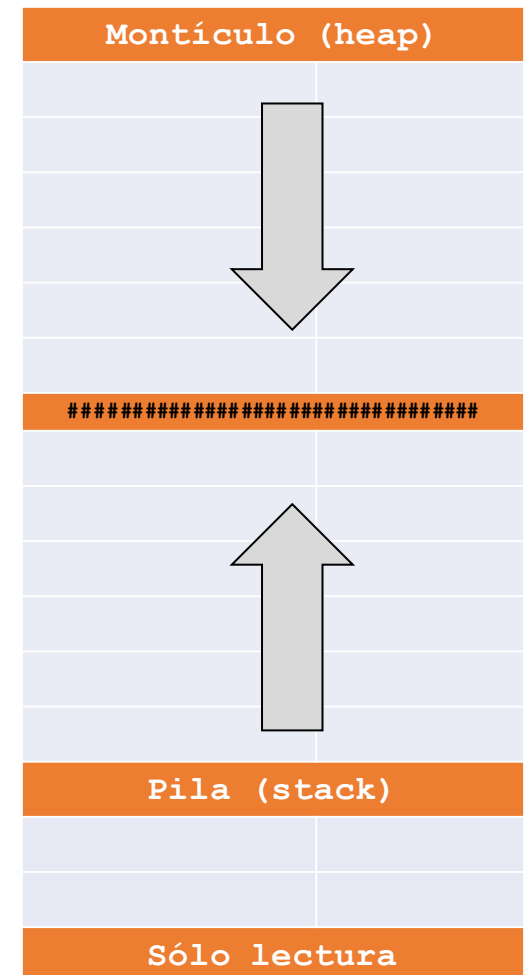
# Contenidos

---

- Primer programa y salida de datos
- Arrays y estructuras
- **Punteros y memoria dinámica**
- Argumentos a funciones
- Punteros a funciones
- Punteros y arrays
- Cadenas de caracteres. Funciones
- Entrada de datos
- Main con argumentos
- Algunas funciones útiles

# Repaso de punteros

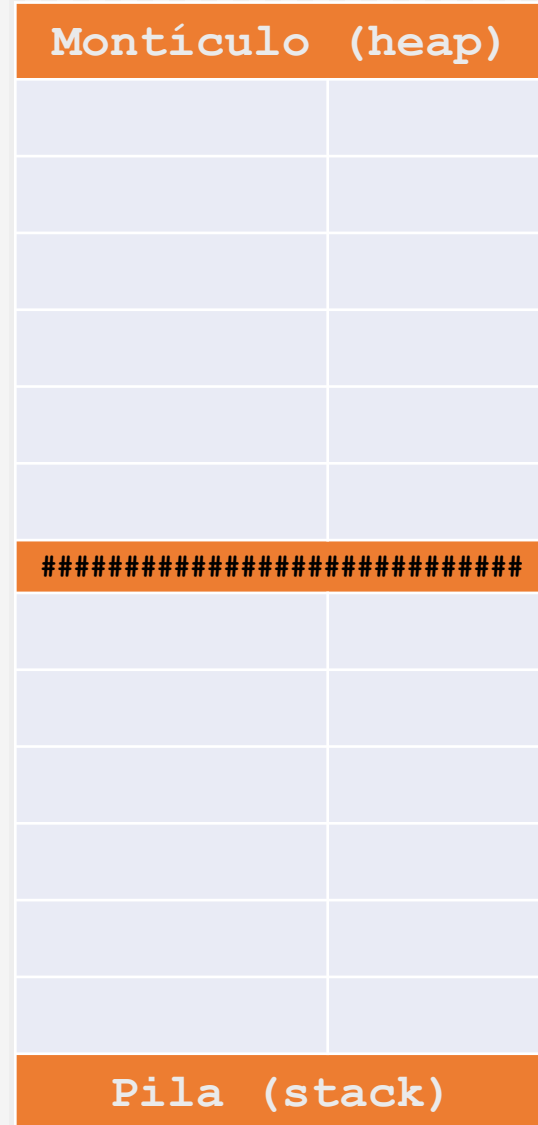
- La memoria de un proceso en Linux se organiza en segmentos
  - Data segment (heap o montículo)
    - Memoria dinámica
  - Stack (pila)
    - Memoria estática (tiempo de compilación)
  - Sólo lectura
    - Text segment
      - Código del programa
      - Nombres de las variables
      - ...
    - Bss segment



# Repaso de punteros

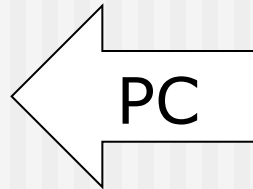
```
1 int x = 10;  
2 int *p;  
3 p = &x;  
4 *p = 20  
5
```

**p** guarda la dirección de memoria de **x**



# Repaso de punteros

```
1 int x = 10;
2 int *p;
3 p = &x;
4 *p = 20
5
```

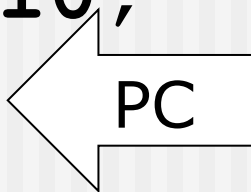


**p** guarda la dirección de memoria de **x**

Montículo (heap)	
#####	
p	#basura#
x	#basura#
Pila (stack)	

# Repaso de punteros

```
1  int x = 10;
2  int *p;
3  p = &x;
4  *p = 20
5
```



**p** guarda la dirección de memoria de **x**

Montículo (heap)	
#####	
p	#basura#
x	10

Pila (stack)
--------------

# Repaso de punteros


```
1 int x = 10;
```

```
2 int *p;
```

3 p = &x;

4 \* p = 20

5



PC

**p** guarda la dirección de memoria de **x**

Montículo (heap)	
#####	
p	#basura#
x	10
Pila (stack)	

# Repaso de punteros


```
1 int x = 10;
```

```
2 int *p;
```

3 p = &x;

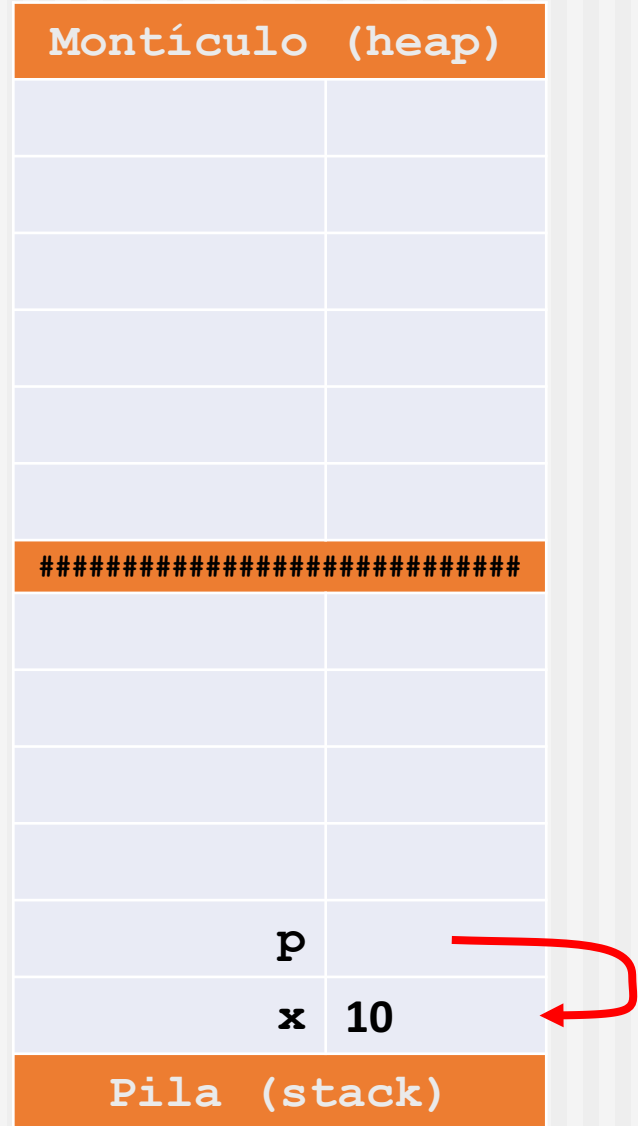
4 \* p = 20

5



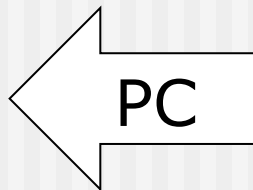
PC

**p** guarda la dirección de memoria de **x**

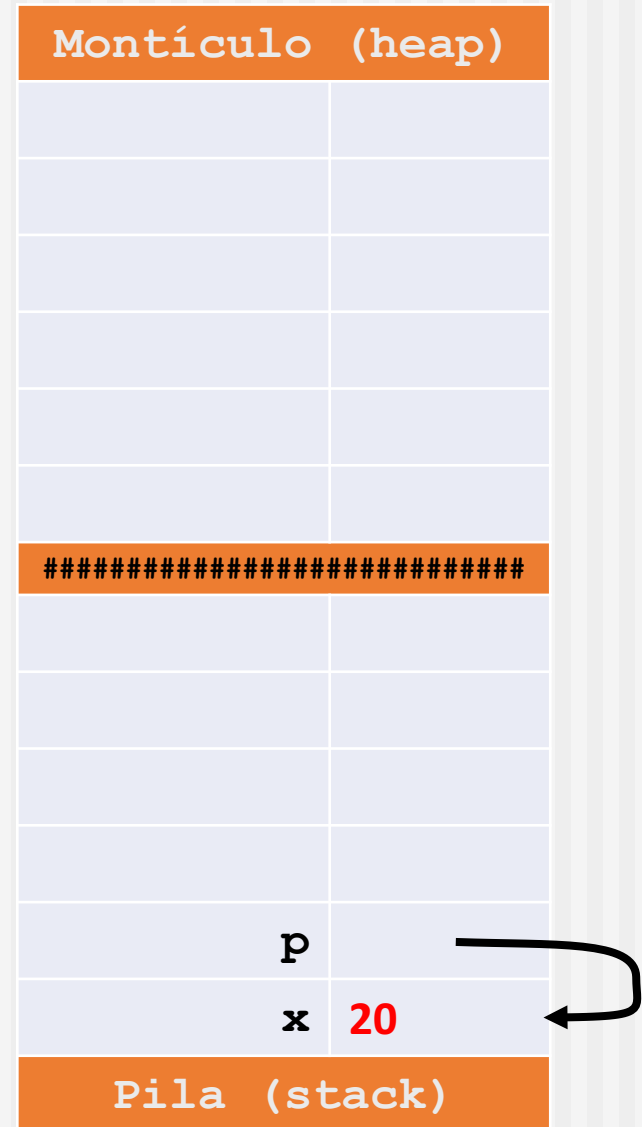


# Repaso de punteros

```
1  int x = 10;
2  int *p;
3  p = &x;
4  *p = 20
5
```



**p** guarda la dirección de memoria de **x**





# Repaso de punteros (III)

Declara un puntero a un entero

```
int x = 10;
```

```
int *p;
```

```
p = &x;
```

& es el operador **dirección**,  
que obtiene la dirección de x

```
*p = 20;
```

Es el operador **desreferencia**,  
que obtiene el valor apuntado por p

# Memoria dinámica en C

- **malloc()** es el equivalente en C a **new** en C++
- **free()** es el equivalente en C a **delete** en C++
- Ejemplo:

```
#include <stdlib.h>
/*se reserva memoria para un entero en el heap */
int * p = (int *)malloc(sizeof(int));
*p = 7;
free(p);
```

- Si se están utilizando **malloc()** y **free()** en C++, no hay que mezclar nunca **new** con **free** ni **malloc** con **delete**

# Memoria dinámica en C++ y C

## ■ C++

- new, new []
- delete, delete[]

○ o bien:

- #include <cstdlib>
- int \* p = (int \*)malloc(50\*sizeof(int))
- free(p);

## ■ C:

- #include <stdlib.h>
- int \* p = (int \*)malloc(50\*sizeof(int))
- //alternativa: int \* p = (int \*)calloc(50,sizeof(int));
- //si usas calloc el contenido de memoria se pone a cero!
- free(p);

# El operador flecha "->"

```
struct MisDatos{  
    int dato1;  
    int dato2;  
};  
  
MisDatos *p = new MisDatos;  
  
// Acceso a campos:  
(*p).dato1 = 5;  
  
// Es equivalente a:  
p->dato1 = 5;
```

# Heap (montón) vs. Stack (pila)

En el Heap / Reserva dinámica	En la Pila / Reserva automática
<pre>void f() {     MisDatos *p = (MisDatos *)         malloc(sizeof(MisDatos));     p-&gt;dato1 = 5;     //... }</pre>	<pre>void f() {     MisDatos p;     p.dato1 = 5;     //... }</pre>

¿Qué sucede cuando **p** queda fuera de alcance?

# Contenidos

---

- Primer programa y salida de datos
- Arrays y estructuras
- Punteros y memoria dinámica
- **Argumentos a funciones**
- Punteros a funciones
- Punteros y arrays
- Cadenas de caracteres. Funciones
- Entrada de datos
- Main con argumentos
- Algunas funciones útiles

# Argumentos pasados por valor

```
#include <stdio.h>

void doble(int i){
    i = i+i;
}

int main(){
    int x = 5;
    doble(x);
    printf("%d\n",x); //resultado: 5
    return 0;
}
```

# Argumentos pasados por referencia en C

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

```
void doble(int * i) {  
    *i = (*i)+(*i);  /* el i de esta funcion  
    es un puntero a la variable que le pasen  
    como argumento */  
}
```

```
int main() {  
    int x = 5;  
    doble(&x);  
    printf("%d\n",x); //resultado: 10  
    return 0;  
}
```



# Argumentos pasados por referencia (II)

```
#include <stdio.h>

void nuevoValor(int * i){
    i = (int *)malloc(sizeof(int));
    /*como el puntero se pasa por valor, su nuevo
    valor no se propaga fuera de la funcion*/
    *i = 7;
}

int main(){
    int x = 5;
    nuevoValor(&x);
    printf("%d\n",x); //resultado: 5
    return 0;
}
```

# Argumentos pasados por referencia (y III)

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

void nuevoValor(int ** i){
    *i = (int *)malloc(sizeof(int));
    /*ahora si puedo reasignar el puntero*/
    **i = 7; /*¡asignacion valida!*/
}

int main(){
    int *x;
    nuevoValor(&x);
    printf("%d\n", *x); //resultado: 7
    free(x); //ahora se puede liberar
    return 0;
}
```

# Contenidos

---

- Primer programa y salida de datos
- Arrays y estructuras
- Punteros y memoria dinámica
- Argumentos a funciones
- **Punteros a funciones**
- Punteros y arrays
- Cadenas de caracteres. Funciones
- Entrada de datos
- Main con argumentos
- Algunas funciones útiles

# Punteros a funciones

- Una función C no es una variable, pero
  - Se puede apuntar a la memoria donde comienza en el segmento de código
  - Se puede usar el puntero como se hace con el resto de variables
- Se define un puntero a un "tipo de funciones"
  - que tienen todas el mismo tipo de retorno
  - y los mismos argumentos de entrada
- Ejemplo:

```
void f() { /*cuerpo de la funcion*/ }
```

```
void (*p) () = f; /*p es el nombre de la variable de  
tipo puntero a una función que devuelve void y no  
tiene argumentos de entrada, por eso se puede  
asignar a f, que cumple este prototipo*/
```

# Punteros a funciones

```
int (*pf)(); /* puntero a función que devuelve un entero */
```

```
void (*pv)(); /* puntero a función que devuelve void */
```

```
int (*pf2)(int,float); /*puntero a una funcion que devuelva int y acepte como  
argumentos int y float*/
```

```
int (*pf3[2])(int); /* array de dos punteros a funciones que devuelvan int y acepten  
como argumentos int*/
```

```
int (*pv2[2][3])(); /*array de 2x3 punteros a funciones que devuelvan int */
```

## ■ Ejemplo:

```
int f(){ /*esta funcion no tiene argumentos y devuelve un entero*/  
    return 0;  
}
```

```
pf = f; //asignacion del puntero a funcion
```

```
pf(); //ejecución de la funcion! (es equivalente a poner directamente f());
```

# Contenidos

---

- Primer programa y salida de datos
- Arrays y estructuras
- Punteros y memoria dinámica
- Argumentos a funciones
- Punteros a funciones
- **Punteros y arrays**
- Cadenas de caracteres. Funciones
- Entrada de datos
- Main con argumentos
- Algunas funciones útiles

# Punteros y arrays

- El identificador de un array o cadena es un puntero constante a su primer elemento

```
int array1[20], array2[20];
```

```
int * p;
```

```
p = array1; //valido!
```

```
array2 = p; //no valido, array2 (y array1) son  
punteros constantes
```

- Se puede trabajar indistintamente con el operador [] o con aritmética de punteros:

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int numbers[5];
6      int * p;
7
8      p = numbers;    *p = 10;
9      p++;            *p = 20;
10     p = &numbers[2]; *p = 30;
11     p = numbers + 3; *p = 40;
12     p = numbers;    *(p+4) = 50;
13
14     for (int i = 0; i < 5; i++)
15         printf("%d,", numbers[i]);
16     printf("\n");
17 }
```

Montículo (heap)

#####

Pila (stack)

Sólo lectura

# arrays como argumentos en funciones

- No es eficiente pasar, por valor, datos de gran tamaño a una función, puesto que siempre se realiza una copia del contenido desde la memoria del contexto del llamante a la memoria del contexto de la función
- El compilador pasa los arrays a una función siempre como punteros
  - `void funcion(char array[])`  
*//es equivalente a*
  - `void funcion(char *array)`
- El programador debe hacer lo mismo si piensa mover grandes cantidades de datos entre funciones (por ejemplo, estructuras)
- Y en C no existen las referencias de C++: `f(tipo &var)`
  - solo existen prototipos de la forma: `f(tipo *var)`



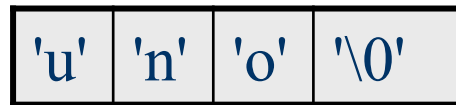
# Contenidos

---

- Primer programa y salida de datos
- Arrays y estructuras
- Punteros y memoria dinámica
- Argumentos a funciones
- Punteros a funciones
- Punteros y arrays
- **Cadenas de caracteres. Funciones**
- Entrada de datos
- Main con argumentos
- Algunas funciones útiles

# Cadenas de caracteres de C

- Declaración como arrays con tamaño  
`char nombre[tamaño];`
- O sin tamaño si se inicializan al declararlos  
`char cadena[] = {'u', 'n', 'o', '\0'};`  
`char cadena2[] = "uno"; //aquí el compilador pone el '\0'`



- El acceso se hace como un array normal, aunque existen también funciones especiales  
`cadena[0] = 'U';`
- La diferencia entre un array de C y una cadena es el **terminador '\0'**, que ocupa una posición, y permite entender el array de caracteres como texto

# Punteros y cadenas de caracteres

- Es habitual encontrar código donde se inicializa una cadena de caracteres como:

```
char * cadena = "hola";
```

- Hay que tener precaución porque en C/C++ dicha inicialización declara un **string literal**
- Según el manual, intentar modificar el contenido de **cadena**, e.g., **cadena[0] = 'H'**, resulta en un comportamiento indeterminado
- En el sistema de nuestra máquina virtual, y para el estándar C99, el siguiente código termina con una **violación de segmento**

```
char * cadena = "hola";  
  
cadena[0] = 'H';
```

# Ejemplo de manejo de cadenas

// copia una cadena de caracteres origen en otra destino

```
void copia(char destino[], const char origen[]){  
    int i=0;  
    while(origen[i] != '\0'){  
        destino[i] = origen[i];  
        ++i;  
    }  
    destino[i] = '\0';  
}
```

es equivalente a poner :  
char \* destino, const char \* origen

//pero en realidad se utiliza la función **strcpy(destino, origen)** que está en <string.h>

# Funciones de manejo de cadenas

- `#include <string.h>`
- Tamaño de una cadena: `strlen()`
- Copia una cadena en otra: `strcpy()`
- Concatena una cadena tras otra: `strcat()`
- Compara cadenas: `strcmp()`
- Busca una cadena dentro de otra: `strstr()`

# Longitud de una cadena

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

int main() {
    char cadena[] = "Hola mundo";
    printf("La cadena tiene %d
           caracteres\n", strlen(cadena));
    //resultado: 10 (no incluye en '\0')
    return 0;
}
```

# Copia de una cadena en otra

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

int main() {
    char destino[512];
    char origen[] = "Hola que hay";
    strcpy(destino, origen);
    /*el programador debe asegurarse de que
    hay sitio en el destino!*/
    printf("Las cadenas son: %s y
           %s\n", origen, destino);
    return 0;
}
```

# Concatenar una cadena con otra

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

int main() {
    char destino[512] = "Hola ";
    char origen[] = "que hay";
    strcat(destino, origen);
    /*el programador debe asegurarse de que
    hay sitio en el destino!*/
    printf("Resultado: %s\n", destino);
    //Resultado: Hola que hay
    return 0;
}
```



# Comparar dos cadenas

- `strcmp()` devuelve 0 si las cadenas son iguales, u otro valor si son distintas

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

```
int main(){
    char cad1[] = "Uno";
    char cad2[] = "Dos";
    if(strcmp(cad1, cad2) == 0){
        printf("Las cadenas son iguales\n");
    }else{
        printf("Las cadenas son distintas\n");
    }
    return 0;
}
```

# Buscar subcadenas en una cadena

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

int main() {
    char linea[] = "usuario@password";
    char * puntero = strstr(linea, "@");
    /*devuelve el puntero a la posicion de
    la cadena donde empieza "@" */
    *puntero = '\\0'; //elimino '@' por '\\0'
    /*ahora hay dos cadenas en la memoria */
    printf("%s %s\\n", linea, puntero+1);
    //Resultado: usuario password
    return 0;
}
```

# Contenidos

---

- Primer programa y salida de datos
- Arrays y estructuras
- Punteros y memoria dinámica
- Argumentos a funciones
- Punteros a funciones
- Punteros y arrays
- Cadenas de caracteres. Funciones
- **Entrada de datos**
- Main con argumentos
- Algunas funciones útiles

# Entrada de datos en C

## ■ Biblioteca de E/S:

- `#include <stdio.h>`

## ■ Lectura de teclado

```
int x;
```

```
scanf("%d", &x); //direccion donde está x
```

```
char cadena[512];
```

```
scanf("%s", cadena); //dirección de la cadena
```

```
/*OJO: scanf() es una función que utiliza punteros  
a variables*/
```

# Entrada de cadenas de caracteres

- La función **scanf()** tienen diversos inconvenientes
  - Ejemplo: `scanf("%s", cadena);`
  - Problema: cuando encuentra un delimitador (espacio, tabulador, etc.) corta la cadena → Si el usuario escribe "hola y adios" en el `scanf()` anterior, `cadena` vale sólo "hola".
  - Solución: `scanf("%[^\n]s", cadena);`
    - Hace que **scanf()** acepte todos los caracteres posibles hasta que encuentre un `'\n'`
- Alternativas
  - La función **fgets(cadena, MAXLINE, stdin)**
    - Función segura ante desbordamientos
    - Problema: no quita el `'\n'` del final → las cadenas tengan un carácter extra de longitud
  - La llamada al sistema **read(fd=0, cadena, MAXLINE)**
    - Llamada al sistema para leer de un descriptor de fichero: cuando `fd=0` es el descriptor de la entrada estándar, es decir, el teclado
    - Problemas:
      - No quita el `'\n'`
      - No pone el `'\0'`, con lo que devuelve **no se considera una cadena de caracteres**
    - Es la que usaremos por defecto

# Contenidos

---

- Primer programa y salida de datos
- Arrays y estructuras
- Punteros y memoria dinámica
- Argumentos a funciones
- Punteros a funciones
- Punteros y arrays
- Cadenas de caracteres. Funciones
- Entrada de datos
- **Main con argumentos**
- Algunas funciones útiles

# main() con argumentos

- Para crear la función principal main existen varias formas:

- Sin argumentos:

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

- Con argumentos (permite personalizar la ejecución):

```
int main(int argc, char *argv[]){  
    for(int i=0;i<argc; ++i)  
        printf("%s\n", argv[i]);  
}
```

- **argc** es el argumento que contiene el número de palabras (argumentos) que se pasan al ejecutable (incluyendo el nombre del programa)
- **argv** es un array de cadenas de caracteres, donde cada casilla contiene un argumento (argv[0] contiene el nombre del programa)
- Ejemplo: ./miprograma param1 param2
  - argc vale 3
  - argv es un array de 3 casillas. En argv[0] está la cadena "miprograma", en argv[1] está "param1" y en argv[2] está "param2"

# Contenidos

---

- Primer programa y salida de datos
- Arrays y estructuras
- Punteros y memoria dinámica
- Argumentos a funciones
- Punteros a funciones
- Punteros y arrays
- Cadenas de caracteres. Funciones
- Entrada de datos
- Main con argumentos
- **Algunas funciones útiles**



# Algunas funciones útiles

## ■ **#include <stdlib.h>**

- Para convertir cadenas de caracteres a números
- `atoi()`, `atof()`, `atol()`

## ■ Ejemplo:

```
char cadena[] = "37";  
int i = atoi(cadena);  
printf("%s = %d\n", cadena, i);
```

# Algunas funciones útiles (II)

## ■ **#include <ctype.h>**

- Esta cabecera declara un conjunto de funciones para detectar el tipo de un caracter (mayúscula, minúscula, texto alfanumérico, dígito...
  - `isupper()`, `islower()`, `isalpha()`, `isdigit()`...
- Devuelven 1 si el caracter es de este tipo y 0 en caso contrario

## ■ Ejemplo:

```
char cad[] = "a35";
if(isupper(cad[0]))
    printf("%c es mayuscula\n",cad[0]);
else if(islower(cadena[0]))
    printf("%c es minuscula\n",cad[0]);
if(isdigit(cad[1]))
    printf("En la primera posicion esta el digito
%d\n",cad[1]);
```

# Algunas funciones útiles (y III)

## ■ **#include <ctype.h>**

- También se incluyen algunas funciones para pasar caracteres a mayúscula o minúscula
  - `char toupper(char c)`
  - `char tolower(char c)`
  - Se les pasa un carácter y lo devuelven en máyúscula o minúscula, respectivamente

## ■ Ejemplo con tolower:

```
char str[]="Texto Prueba.\n";

for(int i=0; i < strlen(str); i++){
    printf("%c", tolower(str[i]));
}
printf("\n");
```