Fundamentos de Sistemas Operativos

Introducción al Lenguaje de Programación C

Universidad de Valladolid

Departamento de Informática

Curso 2023/2024

Contenidos

Introducción

- Creado en 1972 por Denis Ritchie.
- Lenguaje de propósito general.
- Inicialmente de nivel medio (entre alto nivel y ensamblador).
- Lenguaje compilado (compilar + enlazar).
- Modular (permite usar bibliotecas propias o estándar).
- (Demasiado) conciso.
- (Relativamente) sencillo de aprender.

Introducción

```c
#include <stdio.h>

int main()
{
    printf("Hola Mundo!\n");
    return 0;
}
```

- `stdio.h` es un fichero de cabecera. Un fichero de cabecera (`.h`) proporciona declaraciones de funciones, variables y constantes sin incluir sus definiciones completas.
- Un fichero de cabecera no es una biblioteca. Proporciona las **declaraciones** de las funciones y estructuras que un programa puede utilizar.
- Una biblioteca es un conjunto de código ya compilado y enlazado que contiene las **implementaciones** de esas funciones y estructuras.

Introducción

- **Compilar con `gcc`:**
  - Para compilar un programa en C con el compilador `gcc`:
    `gcc tu_archivo.c`
  - Crea un fichero ejecutable llamado `a.out`:
- **Opción -o:**
  - La opción `-o` permite especificar el nombre del archivo de salida (ejecutable) que se generará al compilar.
    `gcc tu_archivo.c -o mi_programa`
  - Crea un archivo ejecutable llamado `mi_programa`.
- **Ejecutar el ejecutable:**
  - Puede ejecutar el archivo basta con escribir su nombre precedido de ruta relativa o absoluta.
  - `./mi_programa`

Introducción

- Cada instrucción en C debe terminar con un punto y coma (;).
- Indica el fin de una expresión o declaración.
- Los comentarios en C se puede escribir de dos formas:

```
/* Comentarios de
varias
lineas*/

//comentario de una linea
```

Lenguaje C

Principales palabras clave.

auto	double	int	struct	break
else	long	switch	case	static
register	typedef	char	extern	return
union	const	float	short	unsigned
continue	for	while	void	default
~~goto~~	sizeof		do	if

Lenguaje C

Tipo de Dato	Versión	Tamaño en Bytes
`char`	Entero con signo	1
`short`	Entero con signo	2
`int`	Entero con signo	4
`long`	Entero con signo	8 (en sistemas de 64 bits)
`unsigned char`	Entero sin signo	1
`unsigned short`	Entero sin signo	2
`unsigned int`	Entero sin signo	4
`unsigned long`	Entero sin signo	8 (en sistemas de 64 bits)

- `char` se suele usar para representar caracteres.
- Los literales de caracteres se escriben entre comillas simples: `'a'`.

Lenguaje C

`Reales`

- Los número reales en C se representan con la norma IEEE754.

Tipo de Dato	Tamaño en Bytes
float	4
double	8

`Lógicos`

- El tipo `bool` representa valores lógicos.
- Puede tener dos valores: `true` o `false`.
- Requiere incluir `#include<stdbool.h>`

Lenguaje C

- Declaración de variables.
  `<tipo> <variable> , <variable>;`

```
main() {
    int entero1, entero2;
    char caracter1, c, car2;
    float real1, r2;
    double d1, d2;

    return 0;
}
```

Lenguaje C

- Declaración de constantes.
- Realmente son instrucciones el preprocesador.

```
#define PI 3.1416
#define TAMANO_MAX 200
//Se escriben antes del main
```

- Asignación.

```
int entero = 1, otro_entero;
float r1, r2 = 1.2e-5;
double d = 1.23456789;
char c = 'a', nueva_linea = '\n';

otro_entero = 12;
r1 = r2 / 0.5;
```

Lenguaje C

Operaciones aritméticas en C.

Suma	`a + b`	`a = a + 1`	`a += 1`
Resta	`c - d`	`c = c - d`	`c -= d`
Incremento	`a=a+1`	`a++`	`++a`
Decremento	`b=b-1`	`b--`	`--a`
Multiplicación	`e * f`	`e = e * 2`	`e *= 2`
División	`g / h`	`g = g / 10`	`g /= 10`
Módulo/Resto	`i % 14`	`i = i % 2`	`i % 2`

Lenguaje C

Los operadores incremento(++)/decremento(−−) no son equivalentes según su colocación.

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

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

Lenguaje C

Operaciones de Comparación.

Mayor	`j > k`
Mayor o Igual	`ll >= m`
Menor	`n < op`
Menor o Igual	`q <= r`
Igual	`s == t`
Distinto	`u != v`

Operaciones Lógicas.

AND	`w && y`
OR	`w || y`
Negación	`! z`

Lenguaje C

Ajuste o Cast:

- (<tipo_t>)<expresión>
  fuerza a que el tipo resultado de evaluar <expresión> sea de tipo tipo_t.
- Ejemplos:

```
res = (int) real / entero;
res2 = (float) entero / 2;
```

```
int main() {
  int a=1,b=2;
  float c;
  c = a/b;
  //c vale 0.0
}
```

```
int main() {
  int a=1,b=2;
  float c;
  c = (float)a/b;
  //c vale 0.5
}
```

Tipos de datos compuestos

- En C, un **array** es un tipo de datos compuesto homogéneo. Es una colección de elementos del mismo tipo, indexados por un número entero. Una **matriz** es un array multidimensional, es decir, un array de arrays, de esta forma se pueden crear matrices de más de 2 dimensiones.
- Los elementos de un array se acceden mediante índices, empezando desde 0.

```
int main(){
  int vector[5];
  float matriz[2][3];

  int vector2[5] = {1, 2, 3, 4, 5};
  int matriz2[2][3] = {{1, 2, 3}, {4, 5, 6}};

  vector2[0] = matriz2[1][2] + vector2[3];
}
```

Tipos de datos compuestos

- En C, un `struct` es un tipo de datos compuesto heterogéneo que permite combinar diferentes tipos de datos bajo un mismo tipo. A cada uno de los elmentos que lo forman se les denomina campos.

```c
int main() {
  struct Persona {
    char nombre[20];
    int edad;
  }; //definicion del struct Persona
  struct Persona persona1; //declaracion de la
    variable persona1 de tipo struct Persona
  persona1.edad = 25;
  strcpy(persona1.nombre, "Juan");
}
```

- El operador `.` permite acceder a los campos de una variables declarada como `struct`.

Tipos de datos compuestos

- Los tipos compuestos se pueden combinar para formar tipos más complejos.

```
struct Persona {
  char nombre[20];
  int edad;
};
struct Grupo {
  struct Persona varias_personas[10];
  char nombre_grupo[20];
};
```

Tipos de datos compuestos

- typedef se utiliza para asignar un nombre alternativo a un tipo de dato existente.
- Facilita la creación de tipos de datos personalizados con nombres más descriptivos.

```
typedef int Entero;
Entero numero = 42;

typedef float Punto[2];
Punto p;
p[1] = 1.5; p[2] = 2.5;

typedef struct {
    char nombre[20];
    int edad;
} Persona;

Persona persona1;
persona1.edad = 25;
strcpy(persona1.nombre, "Juan");
```

Estructuras condicionales

- En C, la estructura condicional `if` permite ejecutar un conjunto de acciones si una condición es verdadera.

```
if (condicion) {
    // Código a ejecutar si la condición es
verdadera
}
```

- Ejemplo:

```
int edad = 20;
if (edad >= 18) {
    printf("Eres mayor de edad.\n");
}
```

Estructuras condicionales

- La estructura de control `if-else` permite ejecutar un conjunto de acciones si una condición es verdadera y otras si es falsa.

```
if (condicion) {
    // Código a ejecutar si la condición es
verdadera
} else {
    // Código a ejecutar si la condición es falsa
}
```

```
int edad = 20;
if (edad >= 18 && edad == 20) {
    printf("Eres mayor de edad y tienes 20.\n");
} else {
    printf("No cumples ambas condiciones.\n");
}
```

Estructuras condicionales

- El `switch` es una estructura condicional que permite seleccionar entre múltiples opciones basadas en el valor de una expresión.

```
switch (expresion_entera) {
    case valor1:
        // Código si la expresión coincide con valor1
        break;
    case valor2:
        // Código si la expresión coincide con valor2
        break;
        ...
    default:
       // Código si no coincide con ningún case
}
```

Estructuras condicionales

- Ejemplo:

```
int opcion = 2;
switch (opcion) {
   case 1:
      printf("Opción 1 seleccionada.\n");
      break;
   case 2:
      printf("Opción 2 seleccionada.\n");
      break;
   default:
      printf("Opción no reconocida.\n");
}
```

Estructuras iterativas

- La estructura iterativa `while` se utiliza para repetir un conjunto de instruccines mientras una condición sea verdadera.

```
while (condicion) {
    // Código a repetir mientras la condición sea
     verdadera
}
```

- Ejemplo:

```
int contador = 0;
while (contador < 5) {
    printf("Contador: %d\n", contador);
    contador++;
}
```

Estructuras iterativas

- La estructura iterativa `do-while` se utiliza para repetir un conjunto de instrucciones al menos una vez, y luego seguir haciéndolo mientras una condición sea verdadera.

```
do {
    // Código a repetir al menos una vez
} while (condicion);
```

- Ejemplo:

```
int contador = 0;
do {
    printf("Contador: %d\n", contador);
    contador++;
} while (contador < 5);
```

Estructuras iterativas

- La estructura iterativa `for` se utiliza para repetir un conjunto de instrucciones un número específico de veces.

```
for (inicialización; condición; incremento/
    decremento) {
   // Código a repetir
}
```

- Ejemplo:

```
int i;
for (i = 0; i < 5; i++) {
    printf("Iteración %d\n", i);
}
```

Si una estructura `if`, `else`, `while` o `for` solo tiene una instrucción, no es necesario el uso de llaves.

```
int i;
for (i = 0; i < 5; i++)
    printf("Iteración %d\n", i);
```

Estructuras iterativas

- El ciclo de vida de una variable está determinado por el alcance en el que se declara.
- Cuando se declara una variable dentro de un bloque de código delimitado por llaves $\{\}$ (o, por ejemplo, en la inicialización de un bloque `for`), su alcance está limitado a ese bloque.
- Ejemplo:

```
int main() {
    int a = 10;
    if (a==10) {
        int b = 20;
        // a y b son visibles aquí
    }
    // a es visible aquí, pero b no lo es
    a = 5; //Esto es correcto
    b = 5; //Esto es un error, b ya no existe
}
```

Ciclo de Vida de las Variables

```
int main() {
    int j = 2;
    for (int i=0; i<10; i++) {
        j = j + 2;
    }
    j = 5; //Esto es correcto
    i = 5; //Esto es un error, i ya no existe
}
```

```
int var; //Esto es una variable global
//Es visible para todas las funciones declaradas a
    continuación
int main() {
    var = 4;
}
```

Entradas y salidas

La función `printf` en C se utiliza para imprimir texto formateado en el terminal.

```
int printf("<cadena control>" [, <argumentos>]);
```

- La cadena de control especifica el formato y determina el número de argumentos (mediante los descriptores de formato %?).
- Los argumentos son las variables o expresiones a escribir, separadas por comas (debe haber una por cada argumento (%?)).
- Devuelve el número de argumentos correctamente escritos. En la cadena de control pueden aparecer:
  - Constantes carácter o cadena, que aparecen como tales.
  - Constantes tipo carácter: \b, \n, \t, \', \".
  - Descriptores de formato, %?, que indican el formato de los argumentos, donde ? es uno de los siguientes:

Entradas y salidas

%c	Carácter sencillo
%d, %i	Entero
%e	Real en notación científica
%f	float
%lf	double
%g	El más corto de %e y %f
%o	Número octal
%x	Número hexadecimal
%s	Cadena de caracteres
%u	Decimal sin signo
%p	Puntero o dirección de memoria

Los descriptores se pueden especificar mediante `%m.n?` para determinar el número de dígitos enteros y decimales. Ejemplos:

- `%10d%10.5f%20s`

Entradas y salidas

```
int main() {
  int entero=5;
  float real=7.5;
  char caracter='a';
  printf("Un entero en una línea: %d \n", entero);
  printf("Dos enteros: %d, %d\n", entero, entero+5);
  printf("Varios %f tipos %c mezclados\n", real,
    caracter);

  printf("Cadena %s de caracteres.\n", "ejemplo cadena
    ");
}
```

¿Qué mostraría por el terminal ese código?

Entradas y salidas

```
int scanf("<cadena control>" [, <argumentos>]);
```

- **Cadena de control**: Igual que en `printf`.
- **Devuelve**: Número de argumentos leídos correctamente.
- Los argumentos son las variables o expresiones a leer.
- Por cada descriptor de formato (`%?`) en la cadena de control, debe haber un argumento (variable donde se almacenará ese elemento).
- Los argumentos que sean de tipos escalares (`int`, `float`, `double`, `char`), deben llevar delante el operador `&` (dirección), se verá el motivo más adelante con el uso de punteros.

Entradas y salidas

```
int entero, entero1, entero2, ent3;
char c;
scanf ("%d", &entero);     // lee un entero
scanf ("%d%d", &entero1, &entero2); // lee dos enteros
    , para que scanf distinga entre uno y otro hay que
     separarlos por un espacio al introducirlos

char cadena[20], string[40];
scanf ("%s", cadena);     // lee una cadena
scanf ("%d %c %s\n", &ent3, &c, string);
// lee un entero, ent3, y un carácter, c  y una cadena
    , string
// El string no lleva &, cuando se estudien los
    punteros, se descubrirá el motivo
```

scanf escanea la entrada hasta que encuentra un espacio o retorno de carro y continúa con la siguiente entrada o finaliza, por lo que no se pueden leer cadenas de caracteres con espacio. Para eso se puede usar fgets.

Punteros

- Todas las variables que se declaran en C se almacena en una zona de memoria denominada pila.
- Cada una de ellas tiene un dirección de memoria que permite localizarlas.
- De cada variable se conoce la primera dirección de memoria de la zona que ocupa.
- Aunque pueden ocupar más de una, por ejemplo, una variable de tipo `int`, normalmente ocupa 4 bytes.
- En C se asigna una dirección diferente a cada byte de la variable.

Punteros

```
int main() {
    int a = 155;
    float b = -25.1;
    char c = 'J';
}
```

Ejemplo de memoria

a	0xF000	10011011
	0xF001	00000000
	0xF002	00000000
	0xF003	00000000
b	0xF004	11001100
	0xF005	11001100
	0xF006	11001000
	0xF007	11000001
c	0xF008	01001010

En ese ejemplo, la dirección de memoria de a es `0xF000` y su valor está codificado en binario, dirección de b es `0xF004` y su valor está codificado con la norma IEEE754 y la de c es `0xF008` y su valor está codificado en ASCII.

Punteros

```
int main() {
    int a[3] = {1, 2, 3};
}
```

Ejemplo de memoria

A	[0]	0xA000	00000001
		0xA001	00000000
		0xA002	00000000
		0xA003	00000000
	[1]	0xA004	00000010
		0xA005	00000000
		0xA006	00000000
		0xA007	00000000
	[2]	0xA008	00000011
		0xA009	00000000
		0xA00A	00000000
		0xA00B	00000000

Los arrays se almacenan en memoria en direcciones contiguas, el operador [] permite indexar (*saltar*) al elemento correspondiente. Los *saltos* son del tamaño del tipo de datos. En el ejemplo, al ser un *array* de `int` de 32 bits, los saltos son de 4 en 4 bytes.

Punteros

- El operador unario & permite conocer la dirección de memoria de una variable o de una determinada posición de un array.

```
int a = 5;
float b[2] = {1.1, -2.1};

printf("La direccion de a es %p, su valor %d\n",&a,a);
printf("La de b[1] es %p y su valor %f\n",&b[1],b[1]);
```

Punteros

- Los punteros son variables que permiten almacenar **direcciones de memoria**.
- El puntero genérico es de tipo `void*`.
- Pero se pueden declarar punteros de tipo cualquier tipo (`int*`, `float*`, `double*`, etc).
- Si se escribe un tipo distinto a `void`, es porque se conoce que la dirección de memoria que se almacena en la variable de tipo puntero almacena ese tipo en memoria.
- Por ejemplo, una variable declarara como `int*` es una variable que contiene una dirección de memoria, y que si accede a esa dirección de memoria, los datos que se encuentren se interpretarán como un entero (`int`).
- Gracias al operador `&` se puede conocer la dirección de memoria de una variable y almacenarla en un puntero.

Punteros

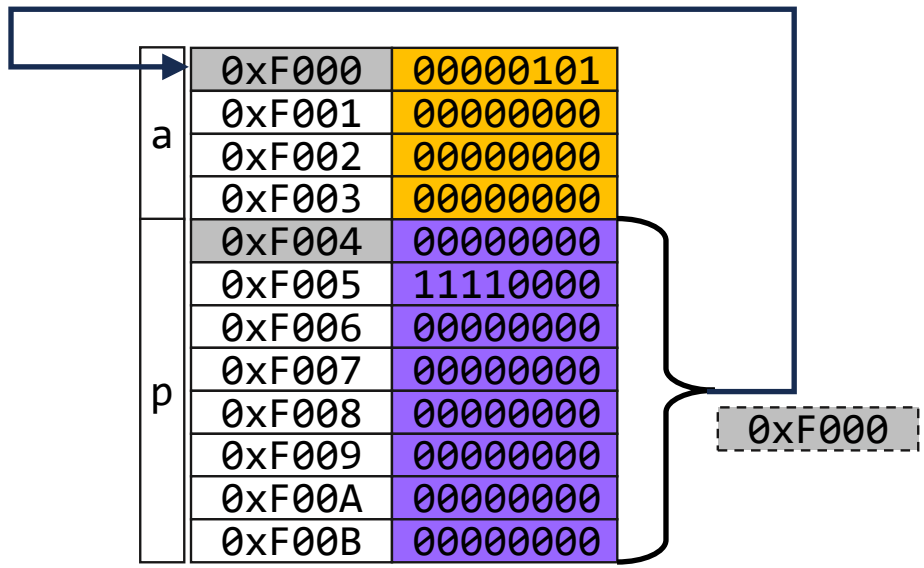
```
int a = 5;
int* p = NULL;
p = &a; // se asigna la
    direccion de memoria de
     la variable a la
    variable puntero p
```

a	0xF000	00000101
	0xF001	00000000
	0xF002	00000000
	0xF003	00000000
p	0xF004	00000000
	0xF005	11110000
	0xF006	00000000
	0xF007	00000000
	0xF008	00000000
	0xF009	00000000
	0xF00A	00000000
	0xF00B	00000000

- `NULL` es el puntero nulo o vacío. Es una buena práctica inicializar siempre los punteros a `NULL`.
- En un sistema de 64 bits los punteros ocupan 64 bits (8 bytes).
- `p` almacena la dirección de memoria de `a` en binario (`0xF000`)

Punteros

- Se les llama puntero (*pointer*) o indirección porque al almacenar una dirección de memoria es como si estuvieran apuntando a esa dirección.



Punteros

- Mediante el operador unario * se puede acceder o modificar el valor contenido en la dirección de memoria que almacena un puntero.
- Ese operador se usa con variables de tipo puntero, no tiene sentido usarlo con variables que no sean puntero.

```
int a = 5; //a vale 5
int *p = NULL;
p = &a;
printf("p contiene la dir %p, que apunta al valor %d\n
    ",p,*p);
*p = 6; //ahora a vale 6 ¿Por qué?
*p = *p + 1; // ahora vale 7
```

Punteros

- Un puntero también puede contener la dirección de memoria en la que empieza un array.
- En C, los *array* y los punteros se manejan de la misma manera (se puede usar el operador [] para indexar punteros).

```
int a[3] = {1,2,3}
int *p = NULL;
p = a; //a contiene la primera dirección de memoria
    del array
*p = 6; //ahora a[0] vale 6
p[1] = p[1] + 1; // ahora a[1] vale 3
```

Punteros

- Cuando un puntero contiene la dirección de memoria de un `struct`, se puede acceder a los campos del `struct` mediante el operador ->

```
typedef struct {
  double x;
  double y;
} Punto;
Punto s; *p = NULL;
s.x = 1.5;
s.y = 3.1;
p = &s;
p->x = 5.5; //Ahora s.x vale 5.5
```

Punteros

- Acceder a la dirección de memoria de un puntero con una dirección no válida provoca una violación de segmento (`segmentation fault`).
- El programador debe asegurarse que los punteros siempre contienen direcciones válidas.

```c
#include <stdio.h>
int main () {
    double *a;
    double b=2.1;
    double c;
    c = b + *a; //
     segmentation fault
    return 0;
}
```

```c
#include <stdio.h>
int main () {
    double *a=NULL;
    double b=2.1;
    double c;
    if (a!=NULL)
        c = b + *a;
    else
        printf("Error en un
     puntero.\n");
    return 0;
}
```

Funciones

- Las funciones en C se declaran precedidas del tipo que devuelven.
- Los parámetros se declaran precedidos de su tipo y de uno en uno.
- Las funciones de declaran antes del `main`.
- La palabra reservada `return` permite retornar el valor de la función.

```
double suma (double a, double b) {
   return a + b;
}

int main()
{
   double a=1.5, b=3.4, c;
   printf("%f + %f = %f\n",a,b,suma(a,b));

}
```

Funciones

- Las funciones se pueden declarar sin definir su cuerpo, siempre antes del `main`.
- Y después del `main` definir el cuerpo.

```
double suma (double a, double b); //declaración
int main()
{
   double a=1.5, b=3.4, c;
   printf("%f + %f = %f\n",a,b,suma(a,b));

}
double suma (double a, double b) { //definición
   return a + b;
}
```

Funciones

- Si una función no devuelve nada (procedimiento) se usa la palabra reservada `void`.

```
void hola_mundo() {
//también es correcto void hola_mundo(void)
   printf("Hola mudo\n");
}

int main()
{
   hola_mundo();
}
```

Funciones

- En C los parámetros **siempre se pasan por copia** (o valor), el paso por referencia requiere el uso de punteros.

```
void suma_uno(int a) {
    a = a + 1;
}

int main()
{
    int a=1; //a vale 1
    suma_uno(a);
    printf("a: %d\n",a); //a sigue valiendo 1
}
```

- La variable a de `suma_uno` y la variable a de `main` son dos variables diferentes, cada una está almacena en la pila de su correspondiente función y tiene una dirección de memoria diferente.

Funciones

- Para hacer el paso por referencia, hay que pasar la dirección de memoria de la variable o variables que se deseen modificar.

```
void suma_uno(int *a) { // se pasa una dirección de
    memoria
   *a = *a + 1;
  //Esto permite acceder y modificar el valor apuntado
     por la dirección de memoria que se ha pasado por
    parámetro
}

int main()
{
   int a=1; //a vale 1
   suma_uno(&a); // Ahora la función suma_uno espera
    una variable de tipo puntero, es decir, espera una
     dirección de memoria. Se le pasa la dirección de
    memoria de a (del main)
   printf("a: %d\n",a); //a vale 2
}
```

Funciones

- Los *arrays* siempre se pasan por referencia, puesto que lo que se pasa de un *array* es la dirección de memoria del primer elemento.

```
double suma_vector(double *v, int tam) {
//tambien se puede escribir:
//double suma_vector(double v[], int tam) {
   double suma=0;
   for (int i=0; i<tam; i++) {
      suma = suma + v[i];
   }
   return suma;
}
```

- Recuerda que en C, un *array* es una sucesión de datos contiguos en memoria. Si se necesita saber el tamaño de *array* en la función, hay que pasarlo como parámetro.

Funciones

```c
#define TAM 5

double suma_vector(double v[], int tam) {
    // tambien se puede escribir :
    // double suma_vector ( double v[], int tam) {
    double suma = 0;
    for (int i = 0; i < tam; i++)
    {
        suma = suma + v[i];
    }
    return suma;
}

int main() {
    double v[TAM] = {1.0, 2.0, 3.0, 4.0, 5.0};
    double suma = 0;
    suma = suma_vector(v,TAM);
    printf("La suma del vector es: %f\n",suma);

}
```

Funciones

- Los `struct` también se pasan por copia en C, para evitar duplicar estructuras de datos complejas y hacer un uso poco eficiente de la memoria se recomienda pasar los `struct` por referencia.

```
typedef struct {
   double x;
   double y;
} Punto;

void suma_puntos(Punto *a, Punto* b, Punto *c)
{
   c->x = a->x + b->x;
   c->y = a->y + b->y;
}
```

Funciones

```
typedef struct {
    double x;
    double y;
} Punto;

void suma_puntos(Punto *a, Punto* b, Punto *c) {
    c->x = a->x + b->x;
    c->y = a->y + b->y;
}

int main() {
    Punto d, e, f;
    d.x = 1.5;
    d.y = 4.5;
    e.x = 2.2;
    e.y = 3.5;
    suma_puntos(&d, &e, &f);
}
```