

(Versión de fecha 27 de octubre de 2023)

*Un puntero es un tipo de datos cuyo valor es una dirección de memoria.*

*Los punteros son una herramienta muy importante del lenguaje C. Otros lenguajes disponen de otras variables similares. En el caso del lenguaje Java, por ejemplo, se dispone de las denominadas referencias. Las referencias de Java no son iguales a los punteros de C, son una abstracción superior, debido en parte al carácter interpretado del lenguaje Java.*

*La utilización correcta de los punteros proporciona al lenguaje C una potencia y una proximidad al lenguaje máquina difícil de superar. Pero su utilización incorrecta también ha sido históricamente una de las mayores fuentes de errores en los programas. Problemas como los punteros colgados (dangling pointers) pueden dar lugar a errores graves y, a veces, muy difíciles de detectar. Empresas como Microsoft o Google afirman que más del 70 % de los errores graves de sus productos desarrollados en C se deben al manejo incorrecto de la memoria.*

*En este documento se va a explicar el funcionamiento de los punteros.*

## 1 DEFINICIÓN

Los punteros en C son un tipo de datos pensado para guardar direcciones de la memoria. En general, la dirección de memoria que guarda un puntero corresponde a la dirección de memoria en la que está guardado un valor de algún tipo de datos. Se dice que el puntero *apunta* a dicho valor o a dicha posición de memoria. Las variables del tipo puntero, además de guardar la dirección de memoria de un valor, guardan también el tipo de datos de dicho valor.

Es habitual, que el valor hacia el que apunta un puntero sea el de alguna variable ya existente, con lo que también es habitual decir que el puntero apunta a la variable o que el puntero guarda la dirección de memoria de una variable<sup>1</sup>.

Los punteros se relacionan con el concepto de *referencia indirecta*. Una variable es una referencia directa a un valor: el contenido de la variable es el valor. En cambio, un puntero es una referencia indirecta a un valor: el contenido del puntero tiene la dirección de memoria en donde está guardado el valor.

## 2 DECLARACIÓN DE PUNTEROS

Para declarar una variable del tipo puntero se necesita indicar el tipo de datos al que va a apuntar dicho puntero. La sintaxis consiste en añadir un asterisco entre el tipo de datos al que apuntará el puntero y el nombre de la variable puntero que se está declarando:

```
tipo_datos * nombre;
```

El asterisco se puede poner pegado al tipo de datos, separado de éste por un espacio, o pegado al nombre de la variable del puntero, las tres formas de hacerlo son válidas. Las siguientes líneas de código serían tres formas equivalentes de declarar una variable del tipo *puntero a entero* llamada *p*:

<sup>1</sup>El concepto *variable* es diferente del concepto *valor*. Las variables se utilizan para guardar valores, pero son conceptos diferentes, aunque a veces se utilizan indistintamente.

```
int* p;  
int * p;  
int *p;
```

### 3 ASIGNACIÓN DE VALOR A LOS PUNTEROS

La forma habitual de asignar valor a un puntero es el operador `&`. Para ello, hay que poner el nombre de una variable existente, del mismo tipo de datos con el que se ha declarado el puntero, precedido sin espacios por el símbolo del operador. El operador `&` se podría leer como *la dirección de memoria de*. La sintaxis es:

```
&nombre_variable;
```

En el siguiente ejemplo, se declara una variable del tipo *double* llamada *x* con un valor de 3.14. A continuación, se declara un puntero a *double* llamado *px* y se le asigna como valor la dirección de memoria de la variable *x*:

```
double x = 3.14;  
double* px = &x;
```

El contenido de un puntero es una dirección de memoria y para imprimirla con instrucciones del tipo *printf()*, se utiliza la especificación de formato *%p*. Dicho formato escribe la dirección a la que apunta el puntero en formato hexadecimal. La siguiente línea de código se podría utilizar para imprimir la dirección de memoria a la que apunta el puntero *px* que se ha definido en el ejemplo anterior.

```
printf("%p \n", px);
```

Hay que tener en cuenta que la dirección de memoria que se muestre no será la misma en distintos ordenadores o, incluso dentro del mismo ordenador, tampoco tiene por qué ser la misma en ejecuciones sucesivas del mismo programa. En la parte izquierda del Ejemplo 1 se muestra el código de la creación de dos variables, una de tipo entero llamada *N* y una de tipo *double* llamada *x*; además, se crean dos punteros *ptr* y *ptr2*, cada uno de ellos apuntando a una de las variables. En la parte derecha del mismo ejemplo se muestra la salida de pantalla de dos ejecuciones sucesivas del programa. Observe que, en cada caso, la dirección de memoria que se ha asignado a los punteros *px* es diferente.

#### Ejemplo 1 Ejemplo de impresión en pantalla de un puntero

```
#include <stdio.h>  
  
int main() {  
    int N = 25;  
    int* ptr = &N;  
  
    double x = 3.14;  
    double* ptr2 = &x;  
  
    printf("N= %d    ptr= %p \n", N, ptr);  
    printf("x= %.2f  ptr2= %p \n", x, ptr2);  
  
    return 0;  
}
```

```
N= 25    ptr= 0x7ffe33767c7c  
x= 3.14  ptr2= 0x7ffe33767c80
```

```
N= 25    ptr= 0x7ffd7128068c  
x= 3.14  ptr2= 0x7ffd71280690
```

La Figura 1 muestra un esquema ficticio de cómo podrían estar guardadas en memoria las variables y punteros del ejemplo anterior. Las direcciones de memoria se han puesto a partir de la posición 1000, para simplificar su escritura en la imagen. Observe que cada puntero guarda el valor de la dirección de memoria en la que está guardada la variable hacia la que apunta.

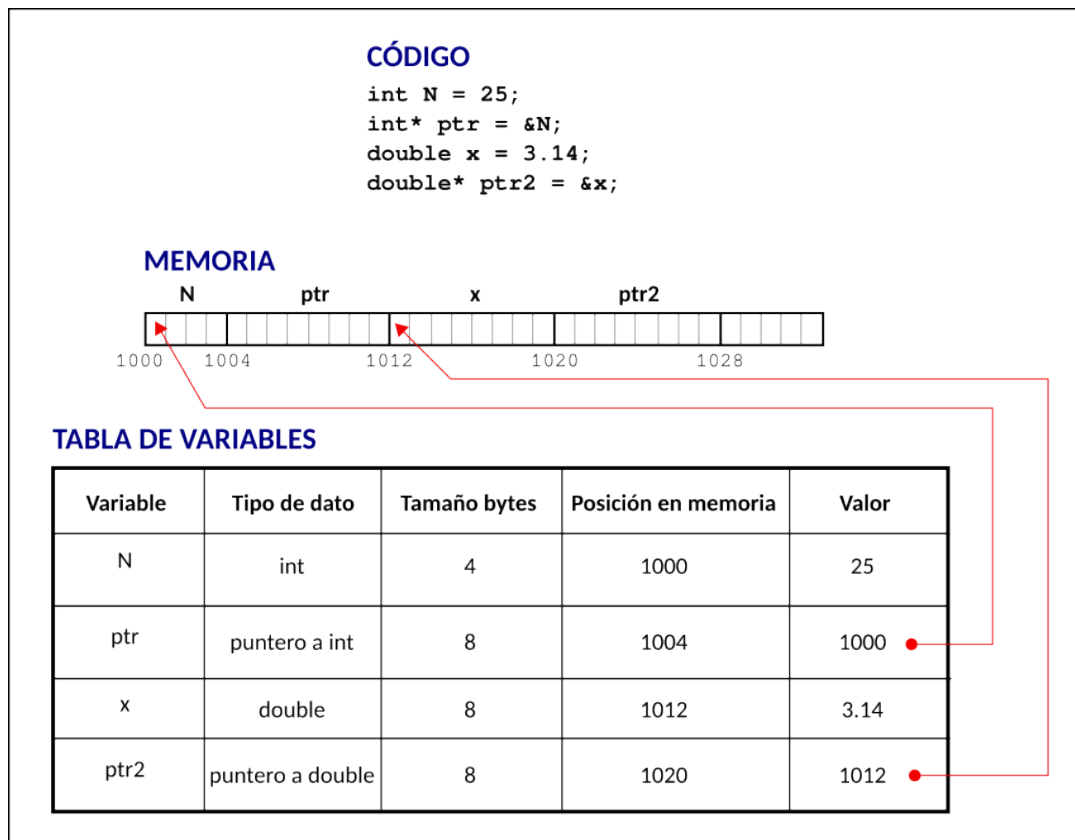


Figura 1: Esquema de almacenamiento y valores de variables y punteros

Al declarar un puntero, no conviene dejarlo sin apuntar a alguna dirección de memoria. Se puede utilizar la macro *NULL*, que está definida en *stddef.h* como la dirección de memoria 0.

Al asignar la dirección 0, se evita que el valor basura que habría en el puntero si no se inicializa pudiera contener alguna dirección de memoria que diera lugar a errores. Además, un puntero declarado *NULL* se puede chequear para ver si el puntero es nulo o tiene asignada ya una dirección. Si se deja sin asignar, la comparación podría indicar que tiene asignada dirección, cuando en realidad es basura.

De esta forma, siempre que se inicialice un puntero conviene hacerlo de manera similar a la del siguiente ejemplo:

```
double* ptr = NULL;
```

#### Para saber más: tamaño de los punteros

El tamaño de las variables puntero es *size\_t*, que corresponde al tipo *long unsigned int* cuyo tamaño depende del procesador. Se puede imprimir en pantalla mediante el siguiente código:

```
printf("%d \n", sizeof(size_t));
```

También se puede utilizar un puntero a cualquier tipo de datos pues, para un mismo ordenador, todos los punteros tienen el mismo tamaño. Por ejemplo, se podría usar el tamaño de un puntero *char\**:

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

En el ordenador del autor, que tiene un procesador de 64 bits, los punteros tienen un tamaño de 8 bytes.

## Para saber más: el sistema de numeración hexadecimal

El sistema hexadecimal es un sistema de numeración posicional de base 16 ( $2^4$ ). Utiliza 16 símbolos:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F

La A hexadecimal es el número 10 decimal, la B el 11, y así hasta la F que es el 15. Se pueden utilizar en mayúsculas o minúsculas. Cuando se está escribiendo un número hexadecimal se suele poner una *x* delante, para especificar que es hexadecimal; también se usa *0x* delante.

El sistema hexadecimal actual fue introducido en el ámbito de la computación por primera vez por IBM en 1963. Se utiliza mucho para simplificar la escritura de números binarios, pues cada dígito equivale a un número binario de 4 bits. Así, un número hexadecimal de dos dígitos puede representar números decimales entre el 0 y el 255, o lo que es lo mismo, un *byte*.

La interpretación decimal de un número hexadecimal se hace de manera similar a como se hace con los números decimales: cada dígito equivale a su valor multiplicado por la potencia de 16 correspondiente a su posición, empezando en cero por la derecha. Así, se tiene:

$$A = 10 \times 16^0 = 10$$

$$1A = 1 \times 16^1 + 10 \times 16^0 = 26$$

$$b3 = 11 \times 16^1 + 3 \times 16^0 = 179$$

$$ff = 15 \times 16^1 + 15 \times 16^0 = 255$$

$$FFFF = 15 \times 16^3 + 15 \times 16^2 + 15 \times 16^1 + 15 = 65535$$

## 4 DESREFERENCIAR UN PUNTERO

Como se ha dicho, cada puntero guarda la dirección de memoria en la que hay guardado determinado valor y el tipo de datos de dicho valor. Con estos datos, a partir del puntero es posible saber el valor que hay guardado en dicha posición de memoria. A la operación de obtener el valor que hay guardado en la posición de memoria a la que apunta un puntero se le denomina *desreferenciar* el puntero.

Se podría hacer una analogía de los punteros con los números de página en el índice de un libro. Se podría decir que los números de página en el índice son punteros a dichas páginas. En ese sentido, ir a la página concreta señalada en el índice sería como desreferenciar el puntero.

Para desreferenciar un puntero se utiliza un asterisco escrito delante del nombre de la variable puntero. En el siguiente código se declaran una variable entera y un puntero que apunta al valor de dicha variable. A continuación, se imprime en pantalla el valor guardado en la variable desreferenciando el puntero.

```
int N = 243;
int* ptr = &N;

printf("%d \n", *ptr); // Imprime 243
```

Sí, está usted pensando lo correcto: el operador desreferenciar utiliza el mismo símbolo que el operador utilizado para declarar un puntero. Seguramente, si se hubiera escogido otro símbolo para la operación desreferenciar, trabajar con punteros sería más sencillo y no habría dado lugar a su consideración histórica como objetos un poco crípticos y difíciles de manejar. Pero el daño ya está hecho.

Cada uno debe buscarse sus trucos mentales para distinguir bien cuándo se trata de una operación de desreferenciar y cuándo se trata de la declaración de un puntero. En cualquier caso, cuando el asterisco va a continuación del nombre de un tipo de datos, se está declarando un puntero y, cuando el asterisco va delante del

nombre de una variable del tipo puntero, sin que haya antes el nombre de un tipo de datos, se está accediendo al valor al que apunta el puntero, se está desreferenciando el puntero. Se podría razonar en el sentido de que la expresión `*ptr` se lee como *valor al que apunta el puntero* y, por tanto, la expresión `int *ptr` indica que el valor al que apunta el puntero es un entero.

Aunque, al declarar un puntero, el asterisco se puede escribir pegado al tipo de datos, separado de él por un espacio, o pegado al nombre de la variable puntero, personalmente siempre escribo el asterisco de la declaración de los punteros pegado al tipo de datos y los asteriscos de la desreferenciación pegados al nombre de la variable puntero. Eso me ayuda a distinguir la operación de que se trata en cada momento. En mi caso, la expresión `int*` la leo mentalmente como *tipo de datos puntero a entero*, la expresión `int* ptr` la leo como *ptr es un puntero a entero* y la expresión `*ptr` (sin tipo de datos delante) la leo como el *valor al que apunta el puntero*.

## 5 COMPARACIÓN DE PUNTEROS

El operador de igualdad `==` y el operador `!=` (no igual) se pueden utilizar con punteros. Se compararán las direcciones de memoria a las que apuntan los punteros. Dos punteros serán iguales si apuntan a la misma dirección de memoria, como muestra el siguiente ejemplo:

### Ejemplo 2 Igualdad de punteros

```
#include <stdio.h>

int main() {
    double x = 3.14;
    double y = 6.28;

    double* ptr_1 = &x;
    double* ptr_2 = &x;
    double* ptr_3 = &y;

    printf("%d\n", ptr_1 == ptr_2); // Imprime 1, true
    printf("%d\n", ptr_1 == ptr_3); // Imprime 0, false

    return 0;
}
```

Los operadores mayor, menor y similares no suele tener sentido utilizarlos con punteros pues, a priori, no se sabe en que orden ha colocado las variables en memoria el compilador.

También se pueden comparar punteros desreferenciados. En ese caso, la comparación corresponderá a la de los valores a los que apuntan los punteros:

### Ejemplo 3 Comparación de valores desreferenciados

```
#include <stdio.h>

int main() {
    double x = 3.14;
    double* ptrx = &x;
    double y = 6.28;
    double* ptry = &y;

    printf("%d\n", *ptrx < *ptry); // Imprime 1, true
    printf("%d\n", *ptrx > 0);    // Imprime 1, true

    return 0;
}
```

## 6 ARITMÉTICA DE PUNTEROS. PUNTEROS Y ARRAYS

Con los punteros, se pueden utilizar los operadores suma (+), resta (−) y los operadores incremento (++) o decremento (−−). Pero cada unidad que se suma o que se reste dará lugar a sumar o restar tantas posiciones de memoria como sea el tamaño del dato al que apunta el puntero.

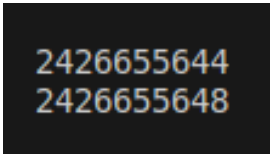
Por ejemplo, si un puntero *ptr* está declarado que apunta a un número del tipo *int*, que ocupa 4 bytes, al sumar una unidad al puntero se obtendrá una dirección de memoria 4 bytes mayor que la original. El objetivo sería que, al sumar una unidad al puntero, quede apuntando a la primera posición de memoria libre tras él. El Ejemplo 4 muestra un ejemplo con un puntero a enteros. Observe en la figura de la derecha del cuadro que la dirección de memoria aumenta en 4 bytes.

### Ejemplo 4 Incremento en una unidad de un puntero a entero

```
#include <stdio.h>

int main() {
    int n = 10;
    int* ptr = &n;
    printf("%u \n", (unsigned int)ptr);
    printf("%u \n", (unsigned int)(ptr+1));

    return 0;
}
```



Esta característica es muy útil cuando se trabaja con *arrays*. Cuando se declara un *array* unidimensional, el nombre de la variable es un puntero al elemento inicial del *array*. Por ello, para declarar un puntero al elemento inicial de un *array* de números enteros de nombre *A*, las dos formas siguientes son equivalentes:

```
int* ptr = A;
int* ptr = &A[0];
```

El lector puede comprobar este extremo con el código del Ejemplo 5.

### Ejemplo 5 El nombre de un array es un puntero

```
#include <stdio.h>

int main() {
    int A[3] = {1, 2, 3};
    int* ptr_1 = &A[0];
    int* ptr_2 = A;
    printf("%p %p \n", ptr_1, ptr_2); // Imprime dos veces la misma dirección de memoria
    printf("%d\n", ptr_1==ptr_2);    // Imprime 1, true, A es equivalente a &A[0]
    return 0;
}
```

Utilizando la aritmética de punteros se pueden recorrer todos los elementos del *array*. Sea por ejemplo un *array* *A* de números enteros. Se cumplen las siguientes expresiones:

A	es un puntero a la dirección del primer elemento del array, o sea, $A == \&A[0]$
*A	es el valor del primer elemento del array, o sea, $*A == A[0]$
A+1	es un puntero a la dirección del segundo elemento del array, o sea, $(A+1) == \&A[1]$
*(A+1)	es el valor del segundo elemento del array, o sea, $*(A+1) == A[1]$
A+i	es un puntero al elemento i del array, o sea, $(A+i) == \&A[i]$
*(A+i)	es el valor del elemento i del array, o sea, $*(A+i) == A[i]$

Observe que, para desreferenciar el elemento  $i$  del *array*, hay que poner  $*(A + i)$ , poniendo entre paréntesis la operación de aritmética de punteros  $(A + i)$ . Una vez calculada la dirección de memoria  $(A + i)$ , se aplica el operador de desreferenciar, el asterisco.

El Ejemplo 6 muestra los elementos de un *array* unidimensional utilizando bucles. Observe que se utilizan dos bucles equivalentes para acceder a los elementos del *array*: el primero utilizando el nombre del *array* como puntero y el segundo creando un puntero al primer elemento del *array*.

### Ejemplo 6 Acceso a los elementos de un *array* utilizando punteros

```
#include <stdio.h>

int main() {
    int A[3] = {1, 2, 3};
    // Utilizando el nombre del array como puntero
    for(int i=0; i<3; i++) {
        printf("%d ", *(A+i));
    }
    printf("\n");
    // Creando un array al primer elemento de array
    int* ptr = &A[0];
    for(int i=0; i<3; i++) {
        printf("%d ", *(ptr+i));
    }
    printf("\n");

    return 0;
}
```

### Para saber más: punteros y arrays de dos dimensiones

En el caso de los *arrays* de dos dimensiones hay que interpretar el *array*, no como una matriz de filas y columnas, sino como un *array* de *arrays*. Por ejemplo, un *array* de  $2 \times 3$ , pongamos  $A[2][3]$ , hay que interpretarlo como 2 *arrays* unidimensionales de 3 elementos. De esta forma,  $A[0]$  es un puntero al *array* de la primera fila y  $A[1]$  es un puntero al *array* de la segunda fila.

El Ejemplo 9 muestra como mostrar en pantalla un *array* de dos dimensiones, utilizando punteros para recorrer los elementos. Observe que, aunque se ha declarado un puntero, se podría haber omitido sustituyendo la expresión  $*(ptr + j)$  por la expresión  $A[i] + j$ .

### Ejemplo 7 Recorrer un *array* 2D utilizando punteros

```
#include <stdio.h>

int main() {
    int A[2][3] = { {1, 2, 3}, {4, 5, 6} };
    int* ptr = NULL;
    for(int i=0; i<2; i++) {
        for(int j=0; j<3; j++) {
            ptr = A[i];
            printf("%d ", *(ptr+j)); // Se podría haber hecho A[i]+j
        }
        printf("\n");
    }
    return 0;
}
```

Incluso se podría utilizar un único puntero a la primera fila, de la siguiente manera:

#### Ejemplo 8 Recorrer un *array* 2D utilizando un sólo puntero

```
#include <stdio.h>

int main() {
    int A[2][3] = { {1, 2, 3}, {4, 5, 6}};

    int* ptr = A[0];
    for(int i=0; i<2; i++) {
        for(int j=0; j<3; j++) {
            printf("%d ", *(ptr+3*i+j));
        }
        printf("\n");
    }
    return 0;
}
```

Se puede ver que los punteros aportan mucha flexibilidad a la hora de recorrer los elementos de un *array* pero, una vez más, la potencia que proporcionan los punteros es un arma de doble filo que puede dar lugar a errores muy difíciles de depurar.

La utilización de los punteros con *arrays* de dos dimensiones da pie a la introducción de los *arrays* de punteros. Observe la siguiente declaración:

```
int* ptr[2];
```

La declaración anterior podría servir para declarar un *array* de 2 punteros. Cada elemento del *array* es un puntero a enteros. Utilizando un *array* de punteros para las filas de la matriz del Ejemplo 9, el código se podría escribir:

#### Ejemplo 9 Recorrer un *array* 2D utilizando punteros

```
#include <stdio.h>

int main() {
    int A[2][3] = { {1, 2, 3}, {4, 5, 6}};

    int* ptr[2] = {NULL};
    for(int i=0; i<2; i++) {
        ptr[i] = A[i];
        for(int j=0; j<3; j++) {
            printf("%d ", *(ptr[i]+j));
        }
        printf("\n");
    }
    return 0;
}
```

En el ejemplo anterior, observe la manera de inicializar en *NULL* todos los punteros del *array* de punteros y cómo se asigna el valor del puntero de fila a los punteros del *array* de punteros dentro del bucle.

## 7 PUNTEROS *VOID\**

Hay veces en las que el tipo de datos al que tendrá que apuntar un puntero no es conocido en el momento de la declaración. En esos casos, se puede utilizar el tipo de puntero *void\**.

```
void* ptr = NULL;
```



Los punteros *void\** pueden guardar una dirección de memoria, pero no tienen asignado ningún tipo de datos, por lo que no pueden desreferenciar el valor almacenado en dicha dirección. A los punteros *void\** se les puede asignar la dirección de memoria de cualquier tipo de dato:

```
void* ptr = NULL;
int n = 10;
ptr = &n;
```

En un puntero *void\**, aunque se le haya asignado una dirección de memoria, el compilador no infiere el tipo de datos del puntero. Para desreferenciar su contenido hay que hacer *casting* del tipo de puntero:

```
void* ptr = NULL;
int n = 10;
ptr = &n;
int h = *(int*)ptr;
```

La expresión de *casting* queda más clara escribiendo unos paréntesis adicionales:

```
int h = *((int*)ptr);
```

Lo que se hace es desreferenciar un puntero *void\** que se ha moldeado a puntero a entero.

El código del Ejemplo 10 ayuda a mostrar el funcionamiento del *casting* a tipo de puntero:

#### Ejemplo 10 Casting de punteros *void\**

```
#include <stdio.h>

int main() {
    void* ptr = NULL;

    int n = 10;
    ptr = &n;
    printf("%d \n", *(int*)ptr);

    double x = 3.14;
    ptr = &x;
    printf("%f \n", *(double*)ptr);

    return 0;
}
```

Los punteros *void\** son muy diferentes de los punteros de un tipo concreto que se inicializan con *NULL*. Un puntero declarado de un tipo concreto inicializado con *NULL* conoce el tamaño del tipo de datos al que apunta. Un puntero *void\** puede apuntar a una dirección, pero sigue sin conocer el tamaño del tipo de datos al que apunta, salvo que se haga *casting*.

## 8 PUNTEROS A ESTRUCTURAS

Los punteros pueden apuntar a valores de cualquier tipo de datos, en particular, un puntero puede apuntar a un tipo de estructura. En el código del Ejemplo 11, se declara una estructura *Punto2D* y se crea una instancia de dicha estructura, llamada *P1*, con unos valores concretos; a continuación, se declara un puntero del tipo *puntero a struct Punto2D*, llamado *ptr*, y se le asigna la dirección de *P1*; por último, se accede al valor de los campos de *P1* desreferenciando el puntero *ptr*:

## Ejemplo 11 Ejemplo de puntero a estructura

```
#include <stdio.h>

int main() {
    // Se define el tipo struct Punto2D
    struct Punto2D {
        double x;
        double y;
    };
    // Se crea una instancia de struct Punto2D
    struct Punto2D P1;
    P1.x = 3.0;
    P1.y = 2.5;
    // Se accede a los campos de P1
    printf("P1 = (%.2f, %.2f)\n", P1.x, P1.y);
    // Se crea el puntero ptr apuntando a P1
    struct Punto2D* ptr= &P1;

    // Se accede a los campos de P1 desreferenciando ptr
    printf("P1 = (%.2f, %.2f) \n", ( *ptr).x, ( *ptr).y);

    return 0;
}
```

Observe que para acceder a los campos, desreferenciando el puntero, es necesario utilizar los paréntesis antes del operador *punto* de acceso al campo. Esto es debido a que el operador *punto* tiene precedencia sobre el operador asterisco de la desreferenciación y, si no se hiciera así, se interpretaría que el campo *x* es de la variable *ptr*, no de la variable *\*ptr* a la que apunta el puntero. Por tanto para desreferenciar un campo de la estructura a la que apunta un puntero, se debe seguir la sintaxis:

`(*nombre_puntero).nombre_campo`

Esta utilización de los punteros para desreferenciar campos de estructuras es muy frecuente, por lo que se ha creado un operador específico que permite desreferenciar directamente los campos de una estructura, utilizando el operador `->`, con la siguiente sintaxis:

`nombre_puntero->nombre_campo`

Esta sintaxis es mucho más simple que la anterior y es la que se suele utilizar. En el Ejemplo 11, la impresión de los campos se podría haber hecho:

```
printf("P1 = (%.2f, %.2f) \n", ptr->x, ptr->y);
```

Observe que el operador `->` incluye la desreferenciación y el acceso al campo.