

(Versión de fecha 27 de octubre de 2024)

## 1 CASTING

### CASTING DE TIPOS

$10/4 \rightarrow 2$  División entera

$10.0/4 \rightarrow 2.5$  División double

double media = 10/4 ; // media es 2.00

---

int suma = 10 ;

double media = suma / 4.0 ; // media es 2.50

---

int suma = 10 ;

int num\_el = 4 ;

double media = suma / num\_el ; // media es 2.00

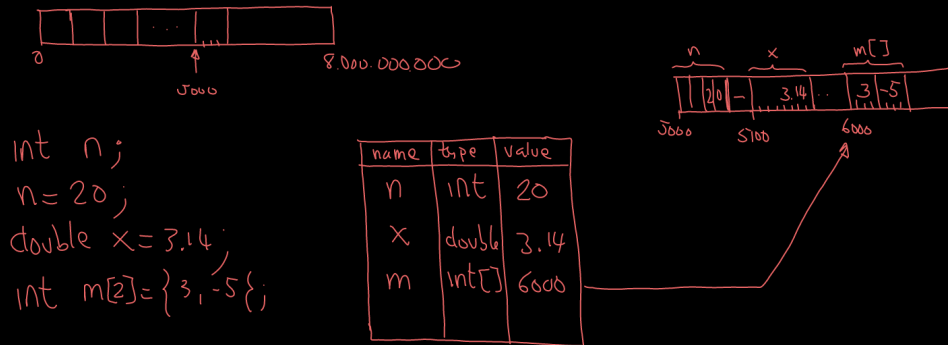
double media = (double)suma / num\_el ; // media es 2.50

## 2 PUNTEROS

(Ver las diapositivas 16 a 21 del tema 3 del curso)

### PUNTEROS

Puntero = variable que guarda la dirección de memoria de un valor y sabe de que tipo de dato es ese valor.



### Declarar puntero:

`tipo * nombre;` → varias opciones equiv.

- respecto al tipo: `int* ptr;`
- en medio: `int * ptr;`
- respecto al n. var: `int *ptr;`

### Asignar puntero

```
int n = 10;  
int* ptr;  
ptr = &n;  
double x = 3.14;  
double* ptr2 = &x;
```

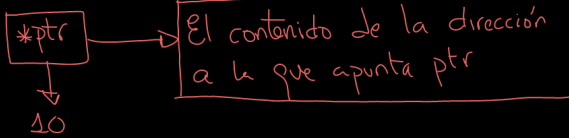
n	int	10
ptr	int*	3500
x	double	3.14
ptr2	double*	7000



## Desreferenciar puntero

```
int n = 10;
```

```
int* ptr = &n;
```



```
int n = 10;
```

```
int* ptr = &n;
```

```
printf("%d\n", *ptr); // imprime 10
```

```
*ptr = 20; // n vale 20
```

---

```
int n[] = {3, 1};
```

`n` es un puntero al primer elem. del array

`n ≡ &n[0]`

**Definición:** los punteros son un tipo especial de variables que guardan la dirección de memoria de algún valor y saben de qué tipo de datos es el valor al que *apuntan*.

**Declaración de un puntero:**

```
tipo_datos * nombre_puntero;
```

## Criterios para escribir el asterisco en la declaración de los punteros

Hay tres formas de escribir el asterisco en la declaración de los punteros:

- Pegado al tipo de datos.
- Entre el tipo de datos y el nombre de la variable puntero.
- Pegado al nombre de la variable puntero.

A mí me gusta escribirlo pegado al tipo de datos, por ejemplo:

```
int* ptr;
```

En la instrucción anterior, mi cabeza lee: *Puntero a número entero ptr*. De esta forma, mi cabeza interpreta que la variable *ptr* es del tipo de datos *puntero a entero*.

En cualquier caso, las tres formas son equivalentes. Lo que si debes hacer es ser consistente. Si utilizas una de las formas, mantén ese criterio en todo el programa y, siempre que declares un puntero, usa ese criterio.

### Ejemplos:

```
int* ptr_1;    // Declara el puntero ptr_1 para apuntar a valores del tipo int
double* ptr_2; // Declara el puntero ptr_2 para apuntar a valores del tipo double
```

### Asignación de valor a punteros:

```
variable_puntero = &nombre_variable;
```

El signo & se podría leer como *la dirección de memoria de*. Así, lo que se hace es asignar al puntero la dirección de memoria de la variable a la que se quiere que apunte el puntero. La variable debe ser del mismo tipo de datos del que se declaró que apuntaba el puntero.

### Ejemplos:

```
int n = 10;    // Declara y asigna la variable entera n
int* ptr;      // Declara el puntero a enteros ptr
ptr = &n;      // Asigna a ptr la dirección de memoria donde guarda su valor la variable n
```

### Desreferenciación de punteros:

Se denomina *desreferenciar* un puntero a obtener el valor contenido en la dirección de memoria a la que apunta. Esto es, suponga el ejemplo anterior en el que el puntero *ptr* apuntaba a la variable *n*. Desreferenciar *ptr* consiste en obtener el valor de *n*.

Para desreferenciar un puntero se pone un asterisco delante del nombre del puntero.

```
*ptr : Valor contenido en la dirección de memoria a la que apunta ptr
```

La desreferenciación funciona en los dos sentidos: puede servir para obtener el valor al que apunta un puntero o para modificar dicho valor.

### Ejemplos

```
int n = 10;    // Declara y asigna la variable entera n
int* ptr = &n; // Declara el puntero a enteros ptr y le asigna la dirección de n
printf("%d \n", *ptr); // Imprime el valor al que apunta ptr. Imprime 10
*ptr = 20;     // Asigna 20 al contenido apuntado por ptr. Ahora n vale 20
```

El siguiente ejemplo muestra algunos usos de punteros. En la imagen de la derecha se pueden ver los resultados del programa.

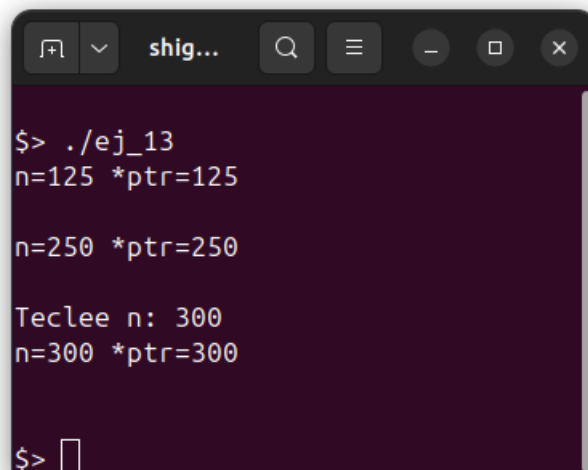
### Ejemplo 1 Utilización de punteros

```
#include <stdio.h>

int main() {
    int n = 125;
    int* ptr = &n;
    printf("n=%d *ptr=%d \n\n", n, *ptr);

    *ptr = 250;
    printf("n=%d *ptr=%d \n\n", n, *ptr);

    printf("Teclee n: ");
    scanf("%d", &n);
    printf("n=%d *ptr=%d \n\n", n, *ptr);
}
```



```
$> ./ej_13
n=125 *ptr=125

n=250 *ptr=250

Teclee n: 300
n=300 *ptr=300

$> 
```

### El nombre de los arrays es un puntero

Hay que saber que, cuando se utiliza un array, el nombre del array equivale a un puntero que apunta al primer elemento del array. Por ejemplo, imagine que se declara una array con la siguiente expresión:

```
double coords[] = 1.5, -2.5;
```

A todos los efectos, el nombre del array equivale a un puntero al primer elemento:

$$coords \equiv \&coords[0]$$

Por ejemplo, es posible acceder al primer elemento del array desreferenciando el nombre del array como si fuera un puntero:

```
double valores[] = {3.14, 2.5};
printf("%.2f \n", *valores); // Imprime 3.14
```

### Para saber más: variables y valores

Cuando se declara un puntero y se le asigna la dirección de memoria en la que una variable guarda su valor, lo que se tiene es dos variables (la variable original y el puntero) apuntando a un mismo valor. En memoria solo hay un valor, pero se dispone de dos variables mediante las que se puede acceder a dicho valor.

Es una forma más de apreciar la diferencia que existe entre el concepto de variable y el concepto del valor que guarda dicha variable.

### 3 ESTRUCTURAS

(Ver las diapositivas 22 a 25 del tema 3 del curso)

#### ESTRUCTURAS

```
struct Libro {  
    int id;  
    double precio;  
};  
struct Libro libro1;  
libro1.id = 127;  
libro1.precio = 3.14;
```

```
struct P2D {  
    double x;  
    double y;  
};  
struct P2D p1;  
p1.x = 10.0;  
p1.y = 32.5;  
printf("%f %f\n", p1.x, p1.y);  
scanf("%lf", &(p1.y));
```

En el lenguaje C, las *estructuras* son un tipo de datos compuesto que permite agrupar bajo un mismo nombre de variable varios valores que no tienen por qué ser del mismo tipo.

Cuando se declara una estructura lo que se hace es crear un nuevo tipo de datos. Luego, será posible declarar variables del nuevo tipo para utilizarlas en el programa.

En el siguiente código, se declara una estructura llamada *Libro* que incluye una variable del tipo *int* llamada *id* y una variable del tipo *double* llamada *precio*:

```
struct Libro {  
    int id;  
    double precio;  
}
```

Una vez que se ha declarado una estructura, se pueden crear variables del nuevo tipo de datos:

```
struct Libro libro1;
```

Observa que la declaración anterior sigue la forma general de la declaración de variables:

```
tipo_de_datos nombre_de_variable ;
```

solo que, en este caso, el tipo de datos se llama *struct Libro*.

A cada una de las variables que forman parte de la definición de una estructura se les denomina *campos* de la estructura. Así, los campos de la estructura *Libro* que se ha definido en los párrafos anteriores son: *id*, del tipo entero y *precio*, del tipo *double*.

Para acceder al valor de los campos de las variables del tipo estructura, se utiliza la nomenclatura *punto*:

```
nombre_variable.nombre_campo
```

Declaración de un nuevo tipo de datos: el tipo **struct Libro**

```
struct Libro {  
    int id;  
    double precio;  
}
```

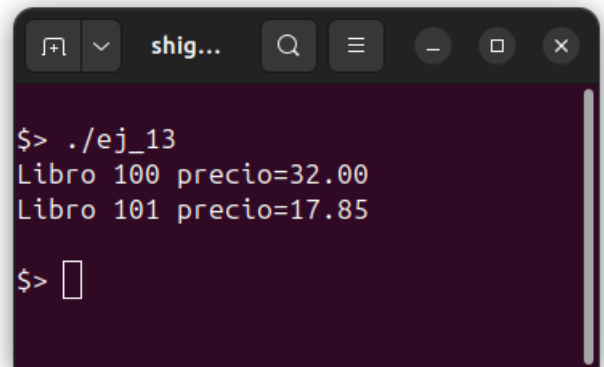
La estructura **Libro** tiene dos campos:

- El campo **id** del tipo **int**
- El campo **precio** del tipo **double**

El siguiente ejemplo de código crea la estructura **Libro**. A continuación, crea dos variables del tipo *struct Libro*, llamadas *libro\_1* y *libro\_2* y asigna valores a sus campos. Por último, muestra en pantalla el valor de los campos de cada una de las variables:

### Ejemplo 2 Ejemplo de uso de estructura

```
#include <stdio.h>  
  
struct Libro {  
    int id;  
    double precio;  
};  
  
int main() {  
  
    struct Libro libro_1, libro_2;  
  
    libro_1.id = 100;  
    libro_1.precio = 32.0;  
    libro_2.id = 101;  
    libro_2.precio = 17.85;  
  
    printf("Libro %d precio=%.2f \n",  
        libro_1.id, libro_1.precio);  
    printf("Libro %d precio=%.2f \n",  
        libro_2.id, libro_2.precio);  
}
```



```
$> ./ej_13  
Libro 100 precio=32.00  
Libro 101 precio=17.85  
$> 
```

### Importante

Observa un detalle muy importante del código del ejemplo anterior: la declaración de la estructura *Libro* se ha hecho fuera de la función *main()*.

Es la forma habitual de hacerlo. Así, el nuevo tipo de datos estará disponible para cualquier función que pueda existir en el programa. Si la declaración se hace dentro de una función concreta, por ejemplo dentro de *main()*, el nuevo tipo solo estaría disponible dentro de la función en la que se ha declarado.

### Uso del operador de asignación con estructuras.

Es posible copiar una estructura en otra utilizando el operador de asignación, el signo igual. El siguiente ejemplo crea una estructura llamada *Punto* que consta de dos campos del tipo *double*: *x* e *y*. A continuación, se crea una variable del tipo *struct Punto* llamada *p1* y se asigna valor a sus campos. Se crea un segundo punto, *p2*, y se impone que sea igual al primero. Por último, se imprimen los valores *x* e *y* del segundo punto:

### Ejemplo 3 Operador de asignación y estructuras

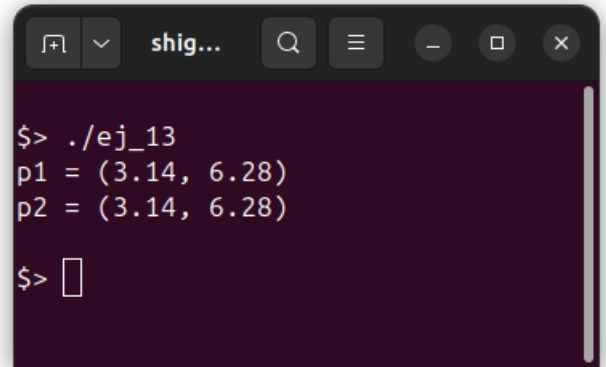
```
#include <stdio.h>

struct Punto {
    double x;
    double y;
};

int main() {
    struct Punto p1;
    p1.x = 3.14;
    p1.y = 6.28;

    struct Punto p2 = p1;

    printf("p1 = (%.2f, %.2f)\n", p1.x, p1.y);
    printf("p2 = (%.2f, %.2f)\n", p2.x, p2.y);
}
```



```
$> ./ej_13
p1 = (3.14, 6.28)
p2 = (3.14, 6.28)

$> 
```

El operador de asignación copia los valores de los campos, pero cada variable, en este caso *p1* y *p2*, tiene su propio juego de valores *x* e *y*. Si, por ejemplo, se modifica el valor de la *x* en *p1*, el valor de la *x* en *p2* no se verá afectado:

### Ejemplo 4 Cada variable tiene sus campos

```
#include <stdio.h>

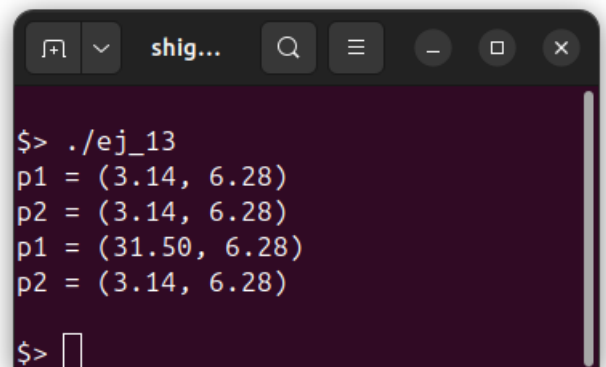
struct Punto {
    double x;
    double y;
};

int main() {
    struct Punto p1;
    p1.x = 3.14;
    p1.y = 6.28;

    struct Punto p2 = p1;

    printf("p1 = (%.2f, %.2f)\n", p1.x, p1.y);
    printf("p2 = (%.2f, %.2f)\n", p2.x, p2.y);

    p1.x = 31.5; // Cambia p1.x, pero no p2.x
    printf("p1 = (%.2f, %.2f)\n", p1.x, p1.y);
    printf("p2 = (%.2f, %.2f)\n", p2.x, p2.y);
}
```



```
$> ./ej_13
p1 = (3.14, 6.28)
p2 = (3.14, 6.28)
p1 = (31.50, 6.28)
p2 = (3.14, 6.28)

$> 
```

**Los operadores de comparación no se pueden aplicar a estructuras, aunque sí a sus campos.**

En el ejemplo anterior, no se podría evaluar si son iguales los campos de *p1* y *p2* utilizando el operador de comparación `==` directamente con los nombres de variables:

`p1 == p2` ⇒ **¡No tiene sentido!**

Si se quisiera comprobar si los valores de los campos *x* e *y* de las variables *p1* y *p2* son iguales, habría que utilizar una expresión como la siguiente:

`(p1.x == p2.x) && (p1.y == p2.y)`

Observa el uso del operador lógico AND, `&&`. La expresión anterior se podría leer como: *La x de p1 es igual que la x de p2 Y SIMULTÁNEAMENTE, la y de p1 es igual que la y de p2.*



## Uso de punteros con los campos de las estructuras.

Es posible declarar punteros que apunten a una variable del tipo estructura. Lo estudiaremos en la segunda parte del curso. También es posible declarar punteros que apunten a los campos de una variable del tipo estructura.

El siguiente ejemplo, crea una variable del tipo *struct Punto* y solicita al usuario los valores de sus campos *x* e *y*. A continuación, crea un puntero que apunta al campo *x* y lo utiliza para realizar operaciones con él.

### Ejemplo 5 Punteros a campos de estructuras

```
#include <stdio.h>

struct Punto {
    double x;
    double y;
};

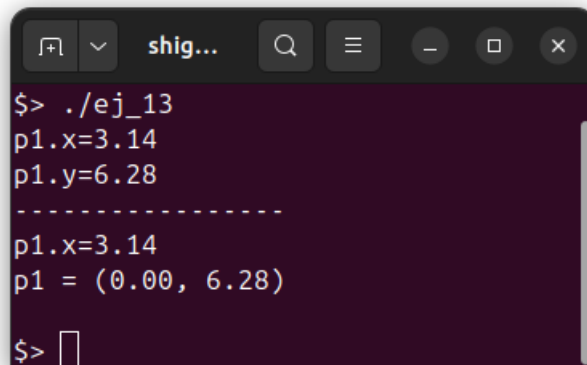
int main() {
    struct Punto p1;

    printf("p1.x=");
    scanf("%lf", &(p1.x));
    printf("p1.y=");
    scanf("%lf", &(p1.y));
    printf("-----\n");

    double* ptrx = &(p1.x);

    printf("p1.x=%.2f\n", *ptrx);

    *ptrx = 0.0;
    printf("p1 = (%.2f, %.2f)\n", p1.x, p1.y);
}
```



```
$> ./ej_13
p1.x=3.14
p1.y=6.28
-----
p1.x=3.14
p1 = (0.00, 6.28)
$> 
```

En el ejemplo anterior, observa que el puntero se declara del tipo de datos del campo, no de la estructura. Observa también que se usa la misma sintaxis para referirse al campo en la instrucción *scanf()* que en la asignación del puntero: *&(p1.x)*.

### Observación

En el ejemplo anterior, se han utilizado paréntesis en las expresiones *&(p1.x)*, para referirse a la dirección de memoria del campo *p1.x*.

En realidad, se podrían haber omitido los paréntesis y se habría obtenido el mismo resultado, por tener precedencia el operador punto sobre el operador *&*.

Pero es buena práctica mantener los paréntesis para que quede clara la intención del código: el operador *dirección de memoria de*, *&*, se aplica al campo *p1.x*.

## 4 TIPOS DE DATOS DEFINIDOS POR EL USUARIO: *TYPEDEF*

(Ver las diapositivas 26 a 31 del tema 3 del curso)

Es posible definir un nombre para un tipo de datos concreto. Sería algo así como declarar un *alias* para referirse a un tipo de datos y poder utilizar en el programa el alias en los lugares donde se quiera usar el tipo de datos al que se refiere. La utilización de nombres personalizados para tipos de datos facilita la lectura de los programas.

Para la creación de alias, se utiliza la instrucción *typedef*. La sintaxis general de la instrucción *typedef* es la siguiente:

```
typedef tipo_de_datos nombre_alias;
```

Por ejemplo, imagina que quieres definir un tipo de datos llamado *MES* para referirse a variables que guardan el número de orden del mes en determinado programa. Podrías hacerlo así:

```
typedef int MES;
```

A partir de ese momento, se podrían crear variables del tipo *MES*, por ejemplo:

```
MES enero = 1;
```

### Observación

A todos los efectos, la variable *enero* del ejemplo anterior, es un número entero. La utilización de *typedef* no impide que alguien pudiera asignar el valor 13 o cualquier otro número entero a una variable del tipo *MES*.

Para conseguir un tipo de variable que solo admitiera valores entre 1 y 12 se pueden utilizar otros mecanismos, por ejemplo las *enumeraciones*, que no estudiamos en este curso.

### Utilización de *typedef* con arrays.

Puede ser útil definir tipos de datos personalizados para ciertos arrays. En el caso de los arrays, la declaración de un tipo de datos personalizados utiliza una sintaxis un poco especial. Por ejemplo, si se quisiera declarar un tipo llamado *Punto2D* para arrays de *double* tamaño 2, se haría de la siguiente forma:

```
typedef double Punto2D[2];
```

El siguiente ejemplo de código utiliza el tipo *Punto2D*:

### Ejemplo 6 *typedef* con arrays

```
#include <stdio.h>

typedef double Punto2D[2];

int main() {

    Punto2D p1 = {1.0, -1.0};

    printf("p1 = (%.2f, %.2f)\n", p1[0], p1[1]);

}
```

## Observación

En el ejemplo anterior, observa que la declaración del tipo de datos *Punto2D* se ha hecho fuera de la función *main()*, como hacíamos con la declaración de las estructuras, para que tenga un alcance global y se pueda utilizar en cualquier parte del programa.

Ésta es la forma habitual de proceder con las declaraciones de tipos personalizados.

Observa también que los corchetes y el tamaño del array se ponen en el nombre del nuevo tipo *Punto2D*, no en la cláusula *double*.

### Utilización de *typedef* con estructuras.

La utilización de *typedef* con estructuras es muy habitual, hasta el punto de que las estructuras se suelen definir siempre con *typedef* y raramente se usa la definición directa de la estructura. El motivo fundamental es que facilita mucho la lectura y escritura del código.

La sintaxis para definir un tipo de datos personalizado asociado con una estructura es la siguiente:

```
typedef struct {  
    ...  
    campos  
    ...  
} NOMBRE_NUEVO_TIPO;
```

Una vez declarado el nuevo tipo de datos, se puede utilizar por su nombre, sin necesidad de añadir la *cláusula struct*, como sucede cuando la estructura se declara directamente. Este detalle es quizás el que hace que las estructuras se suelen declarar siempre utilizando *typedef*.

El siguiente ejemplo utiliza esta técnica para declarar y usar una estructura llamada *Libro*. Compáralo con el ejemplo equivalente que se usó en el Apartado 3:

### Ejemplo 7 Utilización de *typedef* con estructuras

```
#include <stdio.h>  
  
typedef struct {  
    int id;  
    double precio;  
} Libro;  
  
int main() {  
    Libro libro_1, libro_2;  
  
    libro_1.id = 100;  
    libro_1.precio = 32.0;  
    libro_2.id = 101;  
    libro_2.precio = 17.85;  
  
    printf("Libro %d precio=%.2f \n",  
        libro_1.id, libro_1.precio);  
    printf("Libro %d precio=%.2f \n",  
        libro_2.id, libro_2.precio);  
}
```

## Criterios de nombres para estructuras y tipos de datos personalizados

En general, se considera una buena práctica de programación dar nombre a las estructuras y a los tipos de datos personalizados utilizando el criterio *PascalCase*, esto es, una o más palabras con la primera letra mayúscula y sin guiones bajos para separar las palabras.

En los apuntes del curso verás que, a veces, se utiliza una letra *t* como primer carácter del nombre del tipo, para indicar que se trata de un tipo de datos personalizado.

Este criterio de nombres para las variables se denomina *notación húngara* y se popularizó hace años, cuando los lenguajes permitían pocos caracteres para los nombres de variables y los editores de texto no ofrecían las ayudas que proporcionan actualmente.

Hoy en día la notación húngara está en desuso, si bien su uso depende fundamentalmente del estilo de programación de cada uno. Echa un vistazo al artículo que hay al respecto en la wikipedia:

[https://es.wikipedia.org/wiki/Notación\\_húngara](https://es.wikipedia.org/wiki/Notación_húngara)