

Universidad Nacional de Ingeniería
Facultad de Ciencias

Introducción a la Programación (CC-102)

Sesión 9: Punteros - I

¿Qué es un PUNTERO?

Un puntero es un objeto que **apunta** a otro objeto. Es decir, una variable cuyo valor es la **dirección de memoria** de otra variable.

Las **direcciones de memoria** dependen de la arquitectura del ordenador y de la gestión que **el sistema operativo** haga de ella.

¿Qué es un PUNTERO?:

No hay que confundir una dirección de memoria con el contenido de esa dirección de memoria.

```
int x = 25;
```

Dirección		1502	1504	1506	1508		
...	...	25	

La **dirección** de la variable x (&x) es 1502

El **contenido** de la variable x es 25

Declaración de variables puntero

Se declara como todas las variables y donde su identificador va precedido de un asterisco (*):

```
int *punt;
```

punt Es una variable puntero a una variable de tipo entero.

```
char *car;
```

car Es un puntero a variable de tipo carácter.

```
float *ptr;
```

```
float *mat[5]; // ...
```

Un puntero
tiene su
propia
dirección de
memoria:

&punt

&car 4

Declaración de variables puntero

Es decir: hay tantos tipos de punteros como tipos de datos, aunque también pueden declararse punteros a estructuras más complejas (funciones, struct, ficheros...) e incluso punteros vacíos (void) y punteros nulos (NULL).

Ejm. - Declaración de variables puntero:

```
char dato;      //variable que almacenará un carácter.
```

```
char *punt;     //declaración de puntero a carácter.
```

```
float *x;       //declaración de puntero a real
```

```
Personas *y;    //declaración de puntero a estructura
```

```
FILE *z;        //declaración de puntero a archivo
```


Operadores de Punteros

Existen dos operadores:

► Operador de dirección: **&** Representa la dirección de memoria de la variable que le sigue:

`&fnum` representa la dirección de `fnum`.

► Operador de contenido o indirección: ***** Permite acceder al contenido de la variable situada en la dirección de memoria que se especifica en el operando.

`*punt` es el contenido de la dirección de `punt`

Operadores de Punteros

Ejemplo de operadores:

```
float altura = 26.92, *apunta;  
apunta = &altura; //inicialización del puntero  
printf("\n%.2f", altura);    //salida 26.92  
printf("\n%.2f", *apunta);  //salida 26.92
```

No se debe confundir el operador ***** en la declaración del puntero: `int * p;`

Con el operador ***** en las instrucciones:

```
*p = 27;
```

```
printf("\nContenido = %d", *p);
```

Operadores de Punteros

Veamos con un ejemplo en C la diferencia entre todos estos conceptos

Es decir: `int x = 25, *ptrint;`

`ptrint = &x;`

`ptrint` apunta a la dirección de la variable `x`.

`*ptrint` es el valor de la variable (`x`), es decir 25.

`ptrint` tiene su propia dirección: `&ptrint`

Operadores de Punteros

Veamos con otro ejemplo en C la diferencia entre todos estos conceptos

```
int main(void) {  
    int a, b, c, *p1, *p2;  
    void *p;  
    p1 = &a;  
    *p1 = 1;  
    p2 = &b;  
    *p2 = 2;  
    p1 = p2;  
    *p1 = 0;  
}
```

Operadores de Punteros

```
p2 = &c;
```

```
*p2 = 3;
```

```
printf("a=%d b=%d c=%d\n", a, b, c);
```

```
p = &p1;
```

```
p1 = p2;
```

```
*p1 = 1;
```

```
printf("a=%d b=%d c=%d\n", a, b, c); // Paso 13. ¿Qué se imprime?
```

```
return 0;
```

```
}
```

Operadores de Punteros

Veamos con otro ejemplo en C la diferencia entre todos estos conceptos

```
int main(void) {  
  
    int a, b, c, *p1, *p2;  
  
    void *p;  
  
    p1 = &a;           // Paso 1. La dirección de a es asignada a p1  
    *p1 = 1;           // Paso 2. p1 (a) es igual a 1. Equivale a a = 1;  
    p2 = &b;           // Paso 3. La dirección de b es asignada a p2  
    *p2 = 2;           // Paso 4. p2 (b) es igual a 2. Equivale a b = 2;  
    p1 = p2;           // Paso 5. Asigno un puntero a otro  p1 = p2  
    *p1 = 0;           // Paso 6. b = 0  
}
```

Operadores de Punteros

`p2 = &c;` // Paso 7. La dirección de `c` es asignada a `p2`

`*p2 = 3;` // Paso 8. `c = 3`

`printf("a=%d b=%d c=%d\n", a, b, c);` // Paso 9. ¿Qué se imprime?

`p = &p1;` // Paso 10. `p` contiene la dirección de `p1`

`p1 = p2;` // Paso 11. `p1` apunta a `p2`;

`*p1 = 1;` // Paso 12. `c = 1`

`printf("a=%d b=%d c=%d\n", a, b, c);`// Paso 13.¿Qué se imprime?

`return 0;`

`}`

Inicialización de punteros(I)

`< Almacenamiento > < Tipo > * < Nombre > = < Expresión >`

Si `<Almacenamiento>` es `extern` o `static`, `<Expresion>` deberá ser una expresión constante del tipo `<Tipo>` expresado.

Si `<Almacenamiento>` es `auto`, entonces `<Expresion>` puede ser cualquier expresión del `<Tipo>` especificado.

Ejemplos:

- 1) La constante entera 0, NULL (cero) proporciona un puntero nulo a cualquier tipo de dato:

```
int *p;
```

```
p = NULL; //actualización
```


Inicialización de punteros(II):

2) El nombre de un array de almacenamiento static o extern se transforma según la expresión:

a) `float mat[12];`

`float *punt = mat;`

b) `float mat[12];`

`float *punt = &mat[0];`

3) Un "cast" puntero a puntero:

`int *punt = (int *) 123.456;`

Inicializa el puntero con el entero.

Inicialización de punteros(III):

4) Un puntero a carácter puede inicializarse en la forma:

```
char *cadena = "Esto es una cadena";
```

5) Se pueden sumar o restar valores enteros a las direcciones de memoria en la forma: (aritmética de punteros)

```
static int x;
```

```
int *punt = &x+2, *p = &x-1;
```

6) Equivalencia: Dos tipos definidos como punteros a objeto P y puntero a objeto Q son **equivalentes** sólo si P y Q son del **mismo tipo**. Aplicado a matrices:

```
nombre_puntero = nombre_matriz;
```

PUNTEROS Y ARRAYS

Sea el array de una dimensión:

```
int mat[ ] = {2, 16, -4, 29, 234, 12, 0, 3};
```

en el que cada elemento, por ser tipo int, ocupa dos bytes de memoria.

Suponemos que la dirección de memoria del primer elemento, es 1500:

&mat[0] es 1500

&mat[1] será 1502

&mat[7] será 1514

PUNTEROS Y ARRAYS

PUNTEROS Y ARRAYS

```
int mat[ ] = {2, 16, -4, 29, 234, 12, 0, 3};
```

En total los 8 elementos ocupan 16 bytes.

Podemos representar las direcciones de memoria que ocupan los elementos del array , los datos que contiene y las posiciones del array en la forma:

Dirección	1502	1504	1506	1508	1510	1512	1514	
	2	16	-4	29	234	12	0	3
Elemento	mat[1]	mat[2]	mat[3]	mat[4]	mat[5]	mat[6]	mat[7]	

PUNTEROS Y ARRAYS

Dirección	1502	1504	1506	1508	1510	1512	1514
2	16	-4	29	234	12	0	3

Elemento mat[1] mat[2] mat[3] mat[4] mat[5] mat[6] mat[7]

El acceso podemos hacerlo mediante el índice:

$x = \text{mat}[3] + \text{mat}[5]; \quad // \quad x = 29 + 12$

para sumar los elementos de la cuarta y sexta posiciones.

Como hemos dicho que podemos acceder por posición y por dirección: ¿Es lo mismo $\&\text{mat}[0]$ y mat ?

Y $\&\text{mat}[1] = \text{mat}++$?

PUNTEROS Y ARRAYS

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

```
#include <conio.h>
```

```
int mat[5]={2, 16, -4, 29, 234, 12, 0, 3}, i; //declaradas como globales
```

```
void main() {
```

```
    printf("\n%d", &mat[0]);    //resultado: 1500 (dirección de mem)
```

```
    printf("\n%p", mat);        //resultado: 1500 ( " " " " " )
```

```
    i++;                        //i=1
```

```
    printf("\n%p", mat+i);      //resultado: 1502 ( " " " " " )
```

```
    printf("\n%d", *(mat+i));    //resultado: 16 (valor de mat[1] o valor
```

```
    getch(); }                  //en la dirección 1502
```

PUNTEROS Y ARRAYS

Comprobamos con un ejemplo: [lo](#)

Parece deducirse que accedemos a los elementos del array de dos formas:

- mediante el subíndice.
- mediante su dirección de memoria.

Elemento mat[1] mat[2] mat[3] mat[4] mat[5] mat[6] mat[7]

2	16	-4	29	234	12	0	3
---	----	----	----	-----	----	---	---

PUNTEROS

Analizando las direcciones de memoria del array:

**Dirección
del elemento**

1

**Dirección del
octavo elemento**

`&mat[0]` `&mat[1]` `&mat[2]` `&mat[3]` `&mat[4]` `&mat[5]` `&mat[6]` `&mat[7]`

2	16	-4	29	234	12	0	3
---	----	----	----	-----	----	---	---

`mat` `mat+1` `mat+2` `mat+3` `mat+4` `mat+5` `mat+6` `mat+7`

**Puntero a la
dirección del
elemento 1**

**Incremento en una
unidad int (dos bytes)**

`mat++`

PUNTEROS Y ARRAYS

De lo anterior se obtienen varias conclusiones:

- Es lo mismo **&mat[0]** que **mat**, **&mat[2]** que **mat + 2**
- Para pasar de un elemento al siguiente, es lo mismo:

```
for(i=0; i<8; i++)  
    printf("&mat [%d] = %p", i, &mat[i]);
```

que el código:

```
for(i=0; i<8; i++)  
    printf("mat + %d = %p", i, mat + i);
```

A esta forma de desplazarse en memoria se le llama

Aritmética de punteros

PUNTEROS Y ARRAYS

