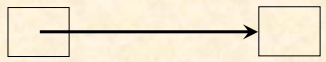
Tema 5: PUNTEROS

TEMA 5.
PUNTEROS

1. CONCEPTO DE PUNTERO.
2. DECLARACIÓN DE PUNTEROS EN C.
3. OPERADORES SOBRE PUNTEROS.
4. RESERVA DINÁMICA DE MEMORIA.
5. RELACIÓN ENTRE PUNTEROS Y ARRAYS.
6. PASO DE PARÁMETROS A FUNCIONES

1. Concepto de puntero

- ✓ **Un puntero es una variable que almacena la dirección en memoria de otro dato.**
  - × En esa posición de memoria tendremos almacenado un dato, del tipo que sea, al cual podremos acceder mediante una referencia al contenido del puntero.
- ✓ **APLICACIONES:**
  - × Devolución de varios datos desde una función a través de los parámetros.
  - × Reserva dinámica de memoria.
- ✓ **Se debe distinguir entre la *dirección* de memoria a la que apunta la variable puntero y el *contenido* de esa dirección.**



puntero (dirección de memoria)

contenido de la dirección

Lenguaje C: Punteros

2. Declaración de punteros en C

- ✓ **Se escribe un asterisco delante del identificador para indicar que se trata de una variable puntero.**

```
tipo  *identificador;
```

- ✓ **Ejemplos:**

```
int *pentero;
```
→ **pentero** es un puntero a un **int**

```
float *preal;
```
→ **preal** es un puntero a un **float**

```
short *pshort;
```
→ **pshort** es un puntero a un **short**

```
int *a, i;
```
→ **a** es un puntero a un **int**
i es un **int**

Lenguaje C: Punteros

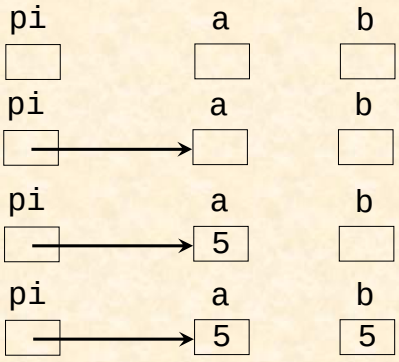
3. Operadores sobre punteros

- ✓ **Existen dos operadores unarios:**
  - × Operador *dirección* (&) ➔ da la dirección de un objeto.
  - × Operador *indirección* (*) ➔ permite acceder al contenido de la celda apuntada por un puntero.

- ✓ **Ejemplo:**

```
int *pi,a,b;

pi = &a;
a = 5;
b = *pi;
```



3. Operadores sobre punteros

EJEMPLO 1:

```
void main(void)
{
  int *pi,a,b;

  pi = &a;
  a = 5;
  b = *pi;
  printf("a = %d, b = %d\n", a, b);
  printf("*pi = %d\n", *pi);
  *pi = 10;
  printf("a = %d, b = %d\n", a, b);
  printf("*pi = %d\n", *pi);
  pi = &b;
  b = 15;
  printf("a = %d, b = %d\n", a, b);
  printf("*pi = %d\n", *pi);
}
```

Salida por pantalla:

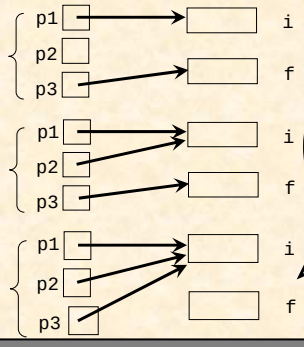
```
a = 5
b = 5
*pi = 5
a = 10
b = 5
*pi = 10
a = 10
b = 15
*pi = 15
```


3. Operadores sobre punteros

- ✓ **Una variable puntero puede apuntar a variables simples (*int, float, char, double*, etc.), *arrays*, funciones, o a otras variables puntero.**
  - × Es responsabilidad del programador que el puntero apunte exactamente al objeto que queremos referenciar.
- ✓ **Se puede asignar un puntero a otro, pero es muy conveniente que ambos apunten al mismo tipo de dato.**
- ✓ **Ejemplo:**

```
int *p1, *p2, i;
float *p3, f;

p3 = &f;
p1 = &i;
p2 = p1;
p3 = p2;
```



La última asignación provoca un *warning* ya que los punteros no apuntan al mismo tipo. Dependiendo del puntero con el que se acceda a esa dirección, se interpretará de forma distinta.

3. Operadores sobre punteros

- ✓ **Las variables puntero se pueden inicializar, igual que cualquier otra variable.**
- ✓ **Un valor inicial muy utilizado es el valor *NULL*.**
  - × *NULL* es una constante definida en el fichero de cabecera *stdlib.h*.
  - × Representa la dirección de memoria 0.
  - × Se utiliza para indicar que una variable puntero no apunta a ninguna dirección de memoria significativa.
  - × Si una variable puntero no se inicializa, o no se le asigna ningún valor, apuntará a una dirección indeterminada.
  - × Para la evaluación de expresiones lógicas, esta constante tiene valor falso.

```
int *p = NULL;
float *q = NULL;
```


3. Operadores sobre punteros

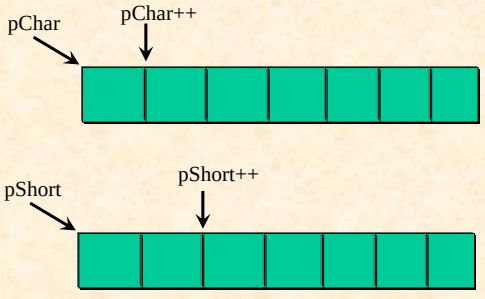
- ✓ **Aritmética de punteros**
  - × Suma o resta de un entero.

```
p+=1;
```

  - × Incremento o decremento.

```
short *pShort;
char *pChar;

pShort++;
pChar++;
```



4. Reserva dinámica de memoria

- ✓ **Según lo estudiado en el tema anterior, cuando declaramos un vector debemos indicar el tamaño para que se le asigne o reserve memoria (asignación o reserva *estática* durante la compilación).**
- ✓ **El uso de punteros permite realizar reserva *dinámica* de memoria.**
  - × La asignación *dinámica* nos permite reservar la memoria necesaria para la cantidad de variables que precisemos en función de la ejecución del programa. (Por ejemplo, en función de la cantidad de valores que desea introducir el usuario en el vector).
- ✓ **Para la asignación dinámica de memoria existe una función específica en C, llamada *malloc* cuyo prototipo es: (cabecera *stdlib.h*).**

```
void * malloc(int tam)
```

  - × *tam* es el tamaño de memoria (en bytes) que se quiere asignar o reservar.
  - × El tipo (*void* *) es un puntero genérico que luego se convertirá al puntero que nos interese.

4. Reserva dinámica de memoria

EJEMPLO 1:

```
// asignación estática de un vector de 50 elementos
char texto[50];

// declaración de un puntero (no asignamos memoria para almacenar valores)
char *ptexto;
int n;

printf("Introduzca n: ");
scanf("%d", &n);
ptexto = (char *) malloc(n);      // Asigna dinámicamente n bytes
if (ptexto==NULL)
    printf("Error en la asignación de memoria\n");
else
   free(ptexto);   // libera la memoria asignada cuando ya no se necesita
```

La función *free* sirve para liberar la memoria asignada dinámicamente

4. Reserva dinámica de memoria

- ✓ **Si se quiere reservar memoria para otro tipo de datos que no sea *char*, se puede usar el operador *sizeof* para obtener el tamaño del tipo de datos**

  *tamaño = (número de elementos) * (tamaño del tipo)*

- ✓ **Ejemplos**
  - × Reserva de memoria dinámica para 50 enteros

```
int *pi;
pi = (int *) malloc(50*sizeof(int));
```

  - × Reserva de memoria dinámica para 100 flotantes

```
float *pi;
pi = (float *) malloc(100*sizeof(float));
```


5. Relación entre punteros y *arrays*

- ✓ **Los punteros y los *arrays* están muy relacionados: el identificador de un *array* es en realidad un puntero al elemento 0 del *array***

  char cad[100]; ⇒ cad == &cad[0]

- ✓ **Ejemplos:**

```
int a[10]
int *pa;

pa= &a[0] ;   ⇔  pa=a;
*pa ⇔ a[0]
*(pa+i) ⇔ a[i]
pa[i] ⇔ a[i]

char mensaje[]="Fin del Programa";     // Vector
char *pmensaje="Fin del Programa";     // Puntero a una constante
```

6. Paso de parámetros a funciones

- ✓ **Como ya sabemos, existen dos tipos de paso de parámetros a funciones: por valor y por dirección**
- ✓ **Paso por valor:**
  - × No se modifica el valor de la variable original
  - × Se utiliza cuando únicamente nos interesa el valor de una determinada expresión que no precisa mantener posibles modificaciones dentro de la función
  - × Los parámetros por valor se comportan como variables dentro de la función
- ✓ **Paso por dirección:**
  - × El parámetro que se pasa es la *dirección* de memoria de la variable, y por tanto el valor de la variable original se puede modificar
  - × Puede usarse para conseguir que una función devuelva más de un valor

6. Paso de parámetros a funciones

EJEMPLO 1:

```
int Funcion(int a, int *b)
{
  *b += 5;
  return(a + *b);
}

void main(void)
{
  int c=0, d=2, e=7;

  /* antes de la llamada: c==0, d==2, e==7 */
  c = Funcion(d, &e);

  /* despues de la llamada: c==14, d==2, e==12 */
  printf("c=%d, d=%d, e=%d\n", c,d,e);
}
```

Salida por pantalla:

```
C=0, d=2, e=7
C=14, d=2, e=12
```

d se pasa por valor,
e se pasa por dirección