LECCIÓN 3: Punteros

María de la Paz Guerrero Lebrero
Curso 2014 / 2015
Grado en Matemáticas
maria.guerrero@uca.es

Índice

- Introducción
- Concepto de puntero
- Operadores de punteros
- Aritmética de punteros
- Punteros como argumento de funciones
- Memoria dinámica
- Ejercicios

Introducción

- ¿Qué lenguajes los usan?
- ¿Por qué se usan?

¿Qué lenguajes los usan?

- Los punteros son muy útiles en programación, la mayoría de los lenguajes permiten su manipulación (C, C++,...).
- Sin embargo, los punteros suelen ser difíciles de utilizar para los programadores nóveles y para cualquier programador que deba depurar una aplicación.
- En nuevos lenguajes de alto nivel, los punteros se han tratado de abstraer (Java, Eiffel, C#, ...).

¿Por qué se usan?

- La razón de ser principal de los punteros reside en manejar datos alojados en la zona de memoria dinámica.
  - Datos elementales (int, char, float,...)
  - Estructuras
  - Funciones
- Se usan principalmente porque optimizan el código a ejecutar.

Concepto de puntero

- Definición
- Manejo de la memoria
- Declaración
- El valor NULL en los punteros
- Ejemplos

Definición

- Un puntero es una variable que contiene una dirección de memoria.
- Normalmente, esa dirección es la posición de otra variable en la memoria.
- Si una variable contiene la dirección de otra variable, entonces se dice que la primera variable apunta a la segunda.

Manejo de la memoria

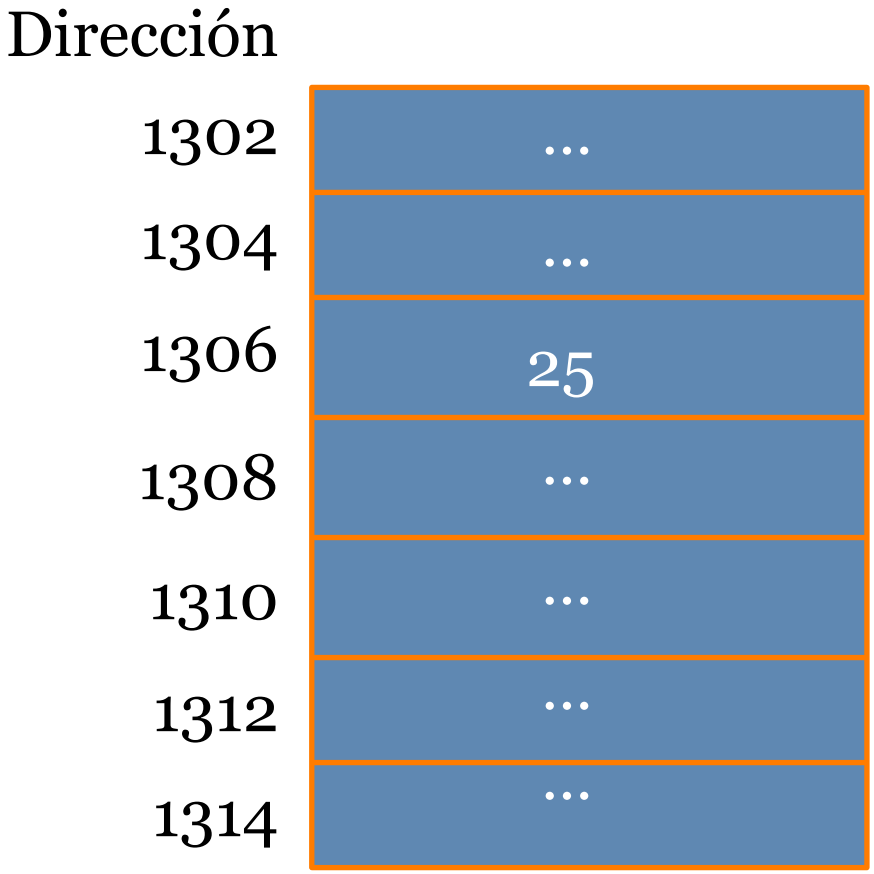
- Las direcciones de memoria dependen de la arquitectura del ordenador y de la gestión que el sistema operativo haga de ella.
- En lenguaje ensamblador se debe indicar numéricamente la posición física de memoria en que queremos almacenar un dato.
- Por ello, este lenguaje depende de la máquina en la que se aplique.

Manejo de la memoria

- En C no debemos, ni podemos, indicar numéricamente la dirección de memoria, si no que utilizamos una etiqueta que conocemos como variable (direcciones de memoria).

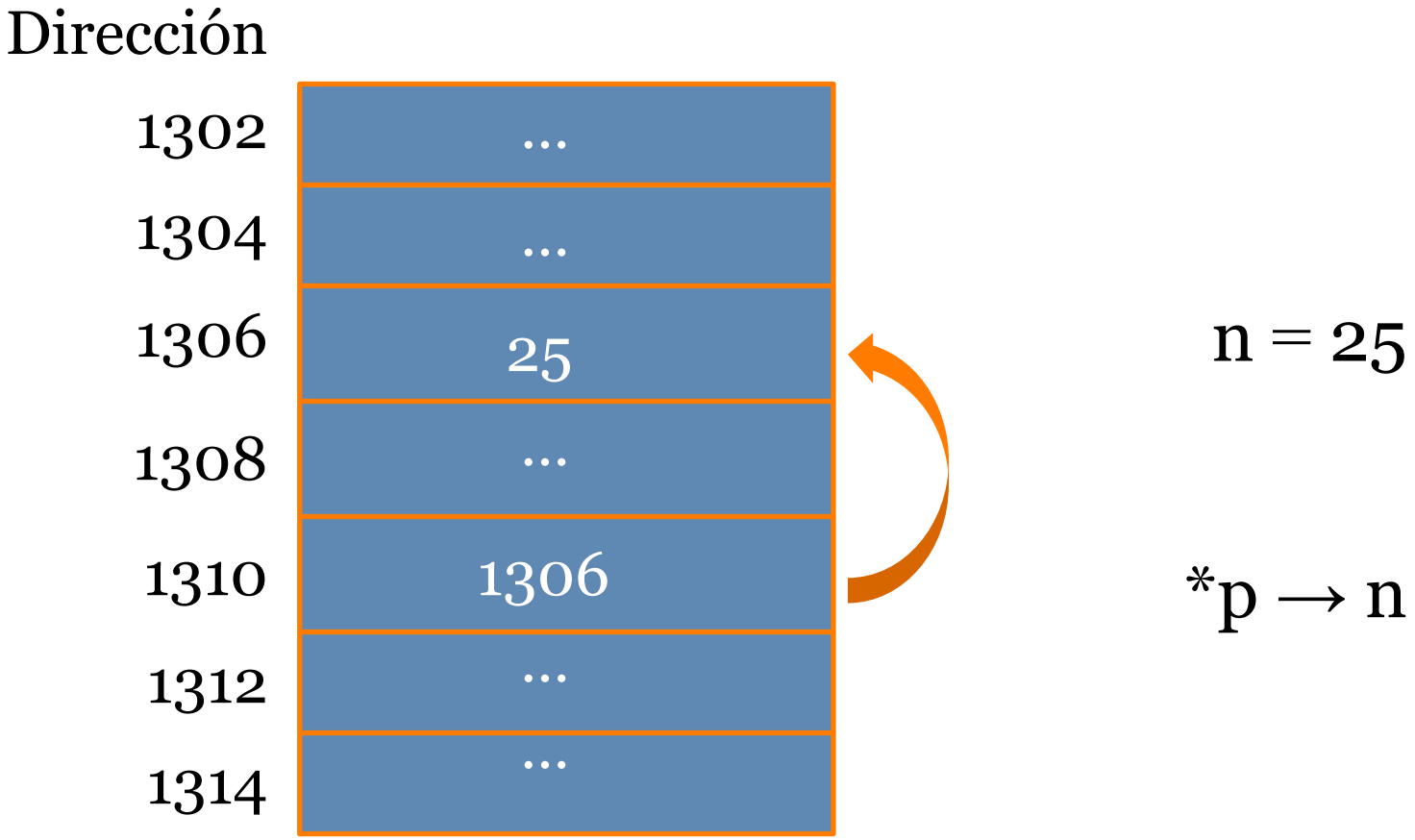
- Lo que nos interesa es almacenar un dato, y no la localización exacta de ese dato en memoria.

Manejo de la memoria



int n = 25;

Manejo de la memoria



Declaración

- Una declaración de un puntero consiste en un tipo base, un * y el nombre de la variable.
- La forma general es:

```
tipo * var_nombre;
```

- Donde tipo es cualquier tipo válido y var_nombre es el nombre de la variable que se ha declarado como puntero.

Declaración

- El tipo base del puntero define el tipo de variables a las que puede apuntar.
- Técnicamente, cualquier tipo de puntero puede apuntar a cualquier dirección de la memoria.
- Sin embargo, toda la aritmética de punteros está hecha en relación a sus tipos base, por lo que es **importante declarar correctamente el puntero**.

El valor NULL en los punteros

- Uno de los principales errores que se pueden cometer cuando se programa en C es utilizar punteros que no apuntan realmente a nada.
- Para evitar esto, es importante que un puntero **siempre** apunte a una variable.
- Cuando no es posible saber a qué variable tiene que apuntar el puntero, debe utilizarse un valor especial, NULL, como valor inicial de los punteros.

Ejemplos

`int *punt;`	• Declaración de un puntero a tipo entero.
`char *car;`	• Declaración de un puntero a tipo carácter.
`float * num;`	• Declaración de un puntero a tipo flotante
`float *mat[5];`	• Declaración de un puntero a un vector de tamaño 5 de tipo flotante

Operadores de punteros

- Operador de dirección (&)
- Operador de contenido (*)
- Ejemplo

Operador de dirección (&)

- Se aplica como prefijo a una variable.
- Devuelve la dirección de memoria en la que se encuentra almacenada la variable.

```
float n = 3.6;
float *ptr = &n;
```

Operador de contenido (*)

- Se aplica como prefijo a un puntero.

- Devuelve el valor contenido en la dirección de memoria almacenada en el puntero.

```
float n = 3.6;
float *ptr = &n;
n = 2.8;
printf("El valor de ptr es %f\n", *ptr);
```

Operador de contenido (*)

- No podemos confundir el operador * en la declaración del puntero:
  - int * ptr;
- Con el operador de contenido que se utiliza en las instrucciones:
  - *ptr = 5;
  - printf("Contenido de ptr = %d\n", *ptr);

Ejemplo

```
int main()
{
        int a,b, c, *p1, *p2;

        p1 = &a;
       *p1 = 1;
        p2 = &b;
       *p2 = 2;
        p1 = p2;
       *p1 = 0;
        p2 = &c;
       *p2 = 3;
        printf("%d %d %d\n", a, b, c);
}
```

Ejemplo

```
#include <stdio.h>
int main()
{
   int a,b, c, *p1, *p2;

   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. El valor del p1 = p2

 *p1 = 0;          // Paso 6. b = 0
```

Ejemplo

```
    p2 = &c;        // Paso 7. La dirección de c es asignada a p2
   *p2 = 3;         // Paso 8. c = 3
    printf("%d %d %d\n", a, b, c);        // Paso 9. ¿Qué se imprime?
}
```

Ejercicios

Ejercicios

1. Dadas las siguientes definiciones y asignaciones, ilustra cómo quedaría la memoria después de ejecutarlas.

```
int x, *p1, *p2;
*p1 = NULL;
*p2 = NULL;
x = 15;
p1 = &x;
p2 = p1;
```

Ejercicios

2. Encuentra el error en el siguiente código en C:

```
#include <stdio.h>
main(){
  float x = 55.4;
  int *p;
  p = &x;
  printf("El valor correcto es: %f\n", x);
  printf("Valor apuntado: %f \n", *p);
}
```

Puntero de distinto tipo que variable a la que apunta

Ejercicios

3. Encuentra el error en el siguiente código en C:

```
#include <stdio.h>
main(){
  int i, *p;
  i = 50;
  *p = i;
  printf("El valor de i es %i \n", i);
  printf("El valor de *p es %i \n", *p);
}
```

¿A dónde apunta p? Falta & antes de la i.

Ejercicios

4. Encuentra el error en el siguiente código en C:

```
#include <stdio.h>
  main(){
  int i, *p;
  i = 50;
  p = i;
  printf("El valor de i es %i \n", i);
  printf("El valor de *p es %i \n", *p);
}
```

P es un puntero, se está asignando una dirección de memoria.

Ejercicios

5. Se necesita almacenar información sobre animales para gestionar una clínica veterinaria. Para comenzar, se supondrá que sólo hay un animal. Del animal es necesario conocer:
   - Tipo de animal (perro, gato, etc.)
   - Nombre
   - Edad
   - Nombre del dueño

Implementa un programa que pida los datos necesarios del animal utilizando la siguiente variable de tipo puntero:

```
animal *a;
```

Aritmética de punteros

- Introducción
- Suma
- Resta
- Comparación
- Asignación

Introducción

- Existe una cierta aritmética que es aplicable a los punteros, y que difiere con respecto a la aritmética vista para los enteros o reales.

- Las operaciones que se pueden realizar son:
  - Suma
  - Resta
  - Comparación
  - Asignación

Suma

- Sumar un valor entero *x* a una variable de tipo puntero significa desplazarnos *x* posiciones hacia delante en memoria.

int * v;

0	1	2	3	4	5	6
23	34	12	56	67	45	89
*v	*(v+1)	*(v+2)	*(v+3)	*(v+4)	*(v+5)	*(v+6)

- Para realizar este desplazamiento, también podemos utilizar el operador de autoincremento (v++);

Resta

- Restar un valor entero *x* a una variable de tipo puntero significa desplazarnos *x* posiciones hacia detrás en memoria.

- También se puede utilizar el operador de autodecremento (v--);

Comparación

- Comparar (==, !=) dos punteros significa evaluar las direcciones de memoria a las que apuntan.

```
int main()
{
    int uno, dos, *p_uno, *p_dos;

    p_uno = &uno;
    p_dos = &dos;

    if(p_uno == p_dos)
       printf("Apuntan a la misma dirección\n");
    else
       printf("NO apuntan a la misma dirección\n");
    system("PAUSE");

}
```

Asignación

- Al igual que el resto de tipos de variables, a los punteros también se les puede asignar (=) contenido, el cual será, normalmente, una dirección de memoria.

```
char c = 'a';
char *ptr = &c;
```

Ejercicios

Ejercicios

1. Dadas las siguientes definiciones y asignaciones, ilustra cómo quedaría la memoria después de ejecutarlas.

```
int A = 1, B = 2, C = 3, *P1, *P2;
P1 = &A;
P2 = &C;
*P1 = (*P2)++;
P1 = P2;
P2 = &B;
*P1 - = *P2;
```

```
++*P2;
*P1 *= *P2;
A = ++*P2 * *P1;
P1 = &A;
*P2 = *P1 /= *P2;
```

Punteros como argumentos de funciones

- Ámbito de las variables
- Paso de parámetros

Ámbito de las variables

- Principalmente en C las actúan en dos ámbitos:
  - **Locales:** Aquellas cuyo tiempo de vida finaliza cuando lo hace la función en la que se declaró. Su uso está restringido únicamente en dicha función.
  - **Globales:** Aquellas cuyo tiempo de vida finaliza con el fin del programa. Su uso está disponible en todas las funciones que componen dicho programa.

Ámbito de las variables

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

int global;

int main()
{
    int local_m;

    local_m = 3;
    escribir_valor();
    global = 5;
    printf("Local = %d.   Global = %d\n", local_m, global);
    system ("PAUSE");
}
```

Ámbito de las variables

```
//No se puede tener acceso a la variable “local_m” de la función main()

void escribir_valor()
{
    int local_e;

    local_e = 8;
    global = 6;
    printf(“Local = %d.   Global = %d\n”, local_e, global);
    system (“PAUSE”);
}
```

Paso de parámetros

- Existen dos formas de pasar los argumentos a una función:
  - **Por valor:** El valor que se le pasa a la función es una copia del original. No se modifica el valor en origen.
  - **Por referencia:** El valor que se le pasa como parámetro a la función es un puntero a la dirección de memoria de la variable. Se puede modificar el valor en origen.

Paso de parámetros

- Ejemplo de paso de parámetros por valor:

```
#include <stdio.h>

int Suma(int a, int b)
{
        int resultado;

        resultado = a + b;

        return resultado;
}
```

```
int main()
{
        int v1, v2, r;
        printf("Introduzca dos valores: ");
         scanf("%d %d", &v1, &v2);

         r = suma(v1, v2);

        printf("La suma es : %d\n", r);

        system("PAUSE");
}
```

Paso de parámetros

- Ejemplo de paso de parámetros por referencia:

```
#include <stdio.h>

void Suma(int *a, int *b, int *r )
{
        *r = *a + *b;
}
```

```
int main()
{
        int v1, v2, r = 0;
        printf("Introduzca dos valores: ");
        scanf("%d %d", &v1, &v2);

        suma(&v1, &v2, &r);

        printf("La suma es : %d\n", r);

        system("PAUSE");
}
```

Memoria dinámica

- malloc()
- free()
- calloc()
- realloc()
- Ejemplo

malloc()

- Es una de las funciones de C para reserva de memoria dinámica.
- Su prototipo es:

```
void *malloc(size_t size);
```

- Reserva *size* bytes en memoria. Si la reserva es correcta devuelve un puntero al bloque de memoria, si no, devuelve el puntero NULL

free()

- Cuando se reserva memoria dinámica ésta persiste hasta que finalice el programa o hasta que el programador libere dicho bloque de memoria mediante la función free().

- Su prototipo es:

```
void free(void* ptr);
```

Ejemplo: malloc() - free()

```
int main()
{
      int *v, i;

      v = (int*) malloc(10 * sizeof(int));

      for(i = 0; i<10; i++)
         v[i] = 0;

     free(v);
     system("PAUSE");
}
```

calloc()

- Similar a malloc() pero además inicializa el bloque de memoria con valor cero.

- Su prototipo es:

```
void *calloc(size_t num_eltos, size_t tam_eltos);
```

- Reserva un bloque de memoria de num_eltos x tam_eltos cuyo valor inicial es 0.

Ejemplo: calloc() - free()

```
int main()
{
        int *v, i;

        v = (int*) calloc(10, sizeof(int));

        free(v);
       system("PAUSE");
}
```

realloc()

- Aumenta el tamaño del bloque de memoria reservado inicialmente por malloc() o calloc().

- Su prototipo es:

```
void *realloc(void *ptr_ini, size_t new_tam);
```

Ejemplo: realloc() - free()

```
int main()
{
        int *v, i;

        v = (int*) calloc(10, sizeof(int));

        v = (int*)realloc(v, 20 * sizeof(int));

        free(v);
        system("PAUSE");
}
```

Ejemplo

```
int main()
{
	int **m, filas, col;

	printf("Introduzca las filas y las columnas: ");
	scanf("%d %d", &filas, &col);

	m = InicializarMatriz(filas, col);
	//...
	LiberarMatriz(m);
	system("PAUSE");
}
```

Ejemplo

```
int** Inicializar(int f, int c)
{
      int **m, i;

      m = (int**)malloc(f * sizeof(int*)); //Inicializamos filas
      for(i = 0; i < f; i++)
         m[i] = (int*)malloc(c * sizeof(int));  //Inicializamos
                                                  columnas

      return m;
}
```

Ejemplo

```
void Liberar(int **m, int f)
{
   int i;

   for(i = 0; i < f; i++)
     free(m[i]);

   free(m);
}
```

Ejercicios

Ejercicios

1. ¿Qué se imprime a la salida del siguiente programa?

```
int main() {
    int *p, *q;
    p=(int *)malloc(sizeof(int));
   *p=5;
    q=(int *)malloc(sizeof(int));
   *q=8;free(p);
     p=q; free(q);
    q=(int *)malloc(sizeof(int));
   *q=6;
    printf("p = %d, q = %d\n", *p, *q);
    return 0;
}
```

Ejercicios

2. Sea NOTAS una matriz de 3x10 que almacena las calificaciones de 10 alumnos de una clase en cada una de las 3 asignaturas que cursan. Se pide:
    1. Diseñar una función que reciba las calificaciones obtenidas por los alumnos en una determinada asignatura y devuelva la calificación media.
    2. Realizar un programa que inicialice la matriz y llame a la función anterior para calcular la media de las tres asignaturas.

Ejercicios

3. Implementa un programa que ordene las líneas de un texto leído desde la entrada estándar, donde cada línea tiene diferente longitud.