9.2. ¿Qué son los punteros?

Un puntero no es más que una **dirección de memoria**. Lo que tiene de especial es que normalmente un puntero tendrá un tipo de datos asociado: por ejemplo, un "puntero a entero" será una dirección de memoria en la que habrá almacenado (o podremos almacenar) un número entero.

Vamos a ver qué **símbolo** usamos en C para designar los punteros:

```
int num;            /* "num" es un número entero */
int *pos;           /* "pos" es un "puntero a entero" (dirección de
                       memoria en la que podremos guardar un entero) */
```

Es decir, pondremos un asterisco entre el tipo de datos y el nombre de la variable. Ese asterisco puede ir junto a cualquiera de ambos, también es correcto escribir

```
int* pos;
```

Esta nomenclatura ya la habíamos utilizado aun sin saber que era eso de los punteros. Por ejemplo, cuando queremos acceder a un fichero, hacemos

```
FILE* fichero;
```

Antes de entrar en más detalles, y para ver la diferencia entre trabajar con "arrays" o con punteros, vamos a hacer dos programas que pidan varios números enteros al usuario y muestren su suma. El primero empleará un "array" (una tabla, de tamaño predefinido) y el segundo empleará memoria que reservaremos durante el funcionamiento del programa.

El primero podría ser así:

```
/*---------------------------*/
/*  Ejemplo en C nº 71:      */
/*  c071.c                   */
/*                           */
/*  Sumar varios datos       */
/*  Version 1: con arrays    */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

main() {
  int datos[100];  /* Preparamos espacio para 100 numeros */
  int cuantos;     /* Preguntaremos cuantos desea introducir */
  int i;           /* Para bucles */
  long suma=0;     /* La suma, claro */

  do {
    printf("Cuantos numeros desea sumar? ");
    scanf("%d", &cuantos);
    if (cuantos>100)  /* Solo puede ser 100 o menos */
      printf("Demasiados. Solo se puede hasta 100.");
  } while (cuantos>100);  /* Si pide demasiado, no le dejamos */
```

```
  /* Pedimos y almacenamos los datos */
  for (i=0; i<cuantos; i++) {
    printf("Introduzca el dato número %d: ", i+1);
    scanf("%d", &datos[i]);
  }

  /* Calculamos la suma */
  for (i=0; i<cuantos; i++)
    suma += datos[i];

  printf("Su suma es: %ld\n", suma);
}
```

Los más avispados se pueden dar cuenta de que si sólo quiero calcular la suma, lo podría hacer a medida que leo cada dato, no necesitaría almacenar todos. Vamos a suponer que sí necesitamos guardarlos (en muchos casos será verdad, si los cálculos son más complicados). Entonces nos damos cuenta de que lo que hemos estado haciendo hasta ahora **no es eficiente**:

- Si quiero sumar 1000 datos, o 500, o 101, no puedo. Nuestro límite previsto era de 100, así que no podemos trabajar con más datos.
- Si sólo quiero sumar 3 números, desperdicio el espacio de 97 datos que no uso.
- Y el problema sigue: si en vez de 100 números, reservamos espacio para 5000, es más difícil que nos quedemos cortos pero desperdiciamos muchísima más memoria.

La solución es reservar espacio estrictamente para lo que necesitemos, y eso es algo que podríamos hacer así:

```
/*---------------------------*/
/*  Ejemplo en C nº 72:      */
/*  c072.c                   */
/*                           */
/*  Sumar varios datos       */
/*  Version 2: con punteros  */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

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

main() {
  int* datos;      /* Necesitaremos espacio para varios numeros */
  int cuantos;     /* Preguntaremos cuantos desea introducir */
  int i;           /* Para bucles */
  long suma=0;     /* La suma, claro */
  do {
    printf("Cuantos numeros desea sumar? ");
    scanf("%d", &cuantos);
    datos = (int *) malloc (cuantos * sizeof(int));
    if (datos == NULL)  /* Si no hay espacio, avisamos */
      printf("No caben tantos datos en memoria.");
  } while (datos == NULL);  /* Si pide demasiado, no le dejamos */

  /* Pedimos y almacenamos los datos */
  for (i=0; i<cuantos; i++) {
    printf("Introduzca el dato número %d: ", i+1);
```

```
    scanf("%d", datos+i);
  }

  /* Calculamos la suma */
  for (i=0; i<cuantos; i++)
    suma += *(datos+i);

  printf("Su suma es: %ld\n", suma);
  free(datos);
}
```

Este fuente es más difícil de leer, pero a cambio es mucho más eficiente: funciona perfectamente si sólo queremos sumar 5 números, pero también si necesitamos sumar 120.000 (y si caben tantos números en la memoria disponible de nuestro equipo, claro).

Vamos a ver las diferencias:

En primer lugar, lo que antes era `int datos[100]` que quiere decir "a partir de la posición de memoria que llamaré *datos*, querré espacio para a guardar 100 números enteros", se ha convertido en `int* datos` que quiere decir "a partir de la posición de memoria que llamaré *datos* voy a guardar varios números enteros (pero aún no sé cuantos)".

Luego reservamos el espacio exacto que necesitamos, haciendo `datos = (int *) malloc (cuantos * sizeof(int));` Esta orden suena complicada, así que vamos a verla por partes:

- "malloc" es la orden que usaremos para reservar memoria cuando la necesitemos (es la abreviatura de las palabra "memory" y "allocate").
- Como parámetro, le indicamos cuanto espacio queremos reservar. Para 100 números enteros, sería "100*sizeof(int)", es decir, 100 veces el tamaño de un entero. En nuestro caso, no son 100 números, sino el valor de la variable "cuantos". Por eso hacemos "malloc (cuantos*sizeof(int))".
- Para terminar, ese es el espacio que queremos reservar para nuestra variable "datos". Y esa variable es de tipo "int *" (un puntero a datos que serán números enteros). Para que todo vaya bien, debemos "convertir" el resultado de "malloc" al tipo de datos correcto, y lo hacemos forzando una conversión como vimos en el apartado 2.4 (operador "molde"), con lo que nuestra orden está completa:
  *datos = (int *) malloc (cuantos * sizeof(int));*
- Si "malloc" nos devuelve NULL como resultado (un "puntero nulo"), quiere decir que no ha encontrado ninguna posición de memoria en la que nos pudiera reservar todo el espacio que le habíamos solicitado.
- Para usar "malloc" deberemos incluir "stdlib.h" al principio de nuestro fuente.

La forma de guardar los datos que teclea el usuario también es distinta. Cuando trabajábamos con un "array", hacíamos `scanf("%d", &datos[i])` ("el dato número i"), pero con punteros usaremos `scanf("%d", datos+i)` (en la posición datos + i). Ahora ya no necesitamos el símbolo **"ampersand" (&)**. Este símbolo se usa para indicarle a C en qué posición de memoria debe almacenar un dato. Por ejemplo, `float x;` es una variable que podremos usar para guardar un número real. Si lo hacemos con la orden "scanf", esta orden no espera que le digamos en qué variable deber guardar el dato, sino en qué posición de memoria. Por eso hacemos `scanf("%f",`

`&x);` En el caso que nos encontramos ahora, `int* datos` ya se refiere a una posición de memoria (un puntero), por lo que no necesitamos & para usar "scanf".

Finalmente, la forma de acceder a los datos también cambia. Antes leíamos el primer dato como datos[0], el segundo como datos[1], el tercero como datos[2] y así sucesivamente. Ahora usaremos el asterisco (*) para indicar que queremos saber el valor que hay almacenado en una cierta posición: el primer dato será *datos, el segundo *(datos+1), el tercero será *(datos+2) y así en adelante. Por eso, donde antes hacíamos `suma += datos[i];` ahora usamos `suma += *(datos+i);`

También aparece otra orden nueva: **free**. Hasta ahora, teníamos la memoria reservada estáticamente, lo que supone que la usábamos (o la desperdiciábamos) durante todo el tiempo que nuestro programa estuviera funcionando. Pero ahora, igual que reservamos memoria justo en el momento en que la necesitamos, y justo en la cantidad que necesitamos, también podemos volver a dejar disponible esa memoria cuando hayamos terminado de usarla. De eso se encarga la orden "free", a la que le debemos indicar qué puntero es el que queremos liberar.

9.3. Repasemos con un ejemplo sencillo

El ejemplo anterior era "un caso real". Generalmente, los casos reales son más aplicables que los ejemplos puramente académicos, pero también más difíciles de seguir. Por eso, antes de seguir vamos a ver un ejemplo más sencillo que nos ayude a asentar los conceptos: Reservaremos espacio para un número real de forma estática, y para dos números reales de forma dinámica, daremos valor a dos de ellos, guardaremos su suma en el tercer número y mostraremos en pantalla los resultados.

```
/*---------------------------*/
/*  Ejemplo en C nº 73:      */
/*  c073.c                   */
/*                           */
/*  Manejo básico de         */
/*  punteros                 */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

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

main() {
  float n1;           /* Primer número, estático */
  float *n2, *suma;   /* Los otros dos números */

  n1 = 5.0;           /* Damos un valor prefijado a n1 (real) */
  n2 = (float *) malloc (sizeof(float)); /* Reservamos espacio para n2 */
  *n2 = 6.7;          /* Valor prefijado para n2 (puntero a real) */

  suma = (float *) malloc (sizeof(float)); /* Reservamos espacio para suma */
  *suma = n1 + *n2;   /* Calculamos la suma */

  printf("El valor prefijado para la suma era %4.2f\n",
    *suma);
```

```
  printf("Ahora es tu turno: Introduce el primer número ");
  scanf("%f",&n1);   /* Leemos valor para n1 (real) */

  printf("Introduce el segundo número ");
  scanf("%f",n2);    /* Valor para n2 (puntero a real) */

  *suma = n1 + *n2;  /* Calculamos nuevamente la suma */

  printf("Ahora la suma es %4.2f\n", *suma);

  free(n2);          /* Liberamos la memoria reservada */
  free(suma);
}
```

Las diferencias son:

- n1 es un "float", así que le damos valor normalmente: n1 = 0; Y pedimos su valor con scanf usando & para indicar en qué dirección de memoria se encuentra: scanf("%f", &n1);
- n2 (y también "suma") es un "puntero a float", así que debemos reservarle espacio con "malloc" antes de empezar a usarlo, y liberar con "free" el epacio que ocupaba cuando terminemos de utilizarlo. Para guardar un valor en la dirección de memoria "a la que apunta", usamos un asterisco: *n2 = 0; Y pedimos su valor con scanf, pero sin necesidad de usar &, porque el puntero ES una dirección de memoria: scanf("%f", n2);

(En este ejemplo, no hemos comprobado si el resultado de "malloc" era NULL, porque sólo pedíamos espacio para dos variables, y hemos dado por sentado que sí habría memoria disponible suficiente para almacenarlas; en un caso general, deberemos asegurarnos siempre de que se nos ha concedido ese espacio que hemos pedido).

9.4. Aritmética de punteros

Si declaramos una variable como `int n=5` y posteriormente hacemos `n++`, debería resultar claro que lo que ocurre es que aumenta en una unidad el valor de la variable n, pasando a ser 6. Pero ¿qué sucede si hacemos esa misma operación sobre un puntero?

```
int *n;
n = (int *) malloc (sizeof(int));
*n = 3;
n++;
```

Después de estas líneas de programa, lo que ha ocurrido no es que el contenido de la posición n sea 4. Eso lo conseguiríamos modificando "*n", de la misma forma que le hemos dado su valor inicial. Es decir, deberíamos usar

```
(*n) ++;
```

En cambio, nosotros hemos aumentado el valor de "n". Como "n" es un puntero, estamos modificando una dirección de memoria. Por ejemplo, si "n" se refería a la posición de memoria

número 10.000 de nuestro ordenador, ahora ya no es así, ahora es otra posición de memoria distinta, por ejemplo la 10.001.

¿Y por qué "por ejemplo"? Porque, como ya sabemos, el espacio que ocupa una variable en C depende del sistema operativo. Así, en un sistema operativo de 32 bits, un "int" ocuparía 4 bytes, de modo que la operación

```
n++;
```

haría que pasáramos de mirar la posición 10.000 a la 10.004. Generalmente no es esto lo que querremos, sino modificar el valor que había almacenado en esa posición de memoria. Olvidar ese * que indica que queremos cambiar el dato y no la posición de memoria puede dar lugar a fallos muy difíciles de descubrir (o incluso a que el programa se interrumpa con un aviso de "Violación de segmento" porque estemos accediendo a zonas de memoria que no hemos reservado).

9.5. Punteros y funciones: parámetros por referencia

Hasta ahora no sabíamos cómo modificar los parámetros que pasábamos a una función. Recordemos el ejemplo 64:

```
/*---------------------------*/
/*  Ejemplo en C nº 64:      */
/*  c064.c                   */
/*                           */
/*  Dos variables locales    */
/*  con el mismo nombre      */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

void duplica(int n) {
   n = n * 2;
}

main() {
   int n = 5;
   printf("n vale %d\n", n);
   duplica(n);
   printf("Ahora n vale %d\n", n);
}
```

Cuando poníamos este programa en marcha, el valor de n que se mostraba era un 5, porque los cambios que hiciéramos dentro de la función se perdían al salir de ella. Esta forma de trabajar (la única que conocíamos hasta ahora) es lo que se llama "pasar **parámetros por valor**".

Pero existe una alternativa. Es lo que llamaremos "pasar **parámetros por referencia**". Consiste en que el parámetro que nosotros pasamos a la función no es realmente la variable, sino la dirección de memoria en que se encuentra dicha variable (usando &). Dentro de la

función, modificaremos la información que se encuentra dentro de esa dirección de memoria (usando *), así:

```
/*---------------------------*/
/*  Ejemplo en C nº 74:      */
/*  c074.c                   */
/*                           */
/*  Modificar el valor de    */
/*  un parámetro             */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

void duplica(int *x) {
   *x = *x * 2;
}

main() {
   int n = 5;
   printf("n vale %d\n", n);
   duplica(&n);
   printf("Ahora n vale %d\n", n);
}
```

Esto permite que podamos obtener más de un valor a partir de una función. Por ejemplo, podemos crear una función que intercambie los valores de dos variables enteras así:

```
/*---------------------------*/
/*  Ejemplo en C nº 75:      */
/*  c075.c                   */
/*                           */
/*  Intercambiar el valor de */
/*  dos parámetros           */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

void intercambia(int *x, int *y) {
   int auxiliar;
   auxiliar = *x;
   *x = *y;
   *y = auxiliar ;
}

main() {
   int a = 5;
   int b = 12;
   intercambia(&a, &b);
   printf("Ahora a es %d y b es %d\n", a, b);
}
```

Este programa escribirá en pantalla que a vale 12 y que b vale 5. Dentro de la función "intercambia", nos ayudamos de una variable auxiliar para memorizar el valor de x antes de cambiarlo por el valor de y.

Ejercicio propuesto: Crear una función que calcule las dos soluciones de una ecuación de segundo grado ($Ax^2 + Bx + C = 0$) y devuelva las dos soluciones como parámetros.

9.6. Punteros y arrays

En C hay muy poca diferencia "interna" entre un puntero y un array. En muchas ocasiones, podremos declarar un dato como array (una tabla con varios elementos iguales, de tamaño predefinido) y recorrerlo usando punteros. Vamos a ver un ejemplo:

```
/*----------------------------*/
/*  Ejemplo en C nº 76:       */
/*  c076.c                    */
/*                            */
/*  Arrays y punteros (1)     */
/*                            */
/*  Curso de C,               */
/*    Nacho Cabanes           */
/*----------------------------*/

#include <stdio.h>

main() {
   int datos[10];
   int i;

   /* Damos valores normalmente */
   for (i=0; i<10; i++)
     datos[i] = i*2;
   /* Pero los recorremos usando punteros */
   for (i=0; i<10; i++)
     printf ("%d ", *(datos+i));
}
```

Pero también podremos hacer lo contrario: declarar de forma dinámica una variable usando "malloc" y recorrerla como si fuera un array:

```
/*----------------------------*/
/*  Ejemplo en C nº 77:       */
/*  c077.c                    */
/*                            */
/*  Arrays y punteros (2)     */
/*                            */
/*  Curso de C,               */
/*    Nacho Cabanes           */
/*----------------------------*/

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

main() {
   int *datos;
   int i;

   /* Reservamos espacio */
   datos = (int *) malloc (20*sizeof(int));
   /* Damos valores como puntero */
```

```
   printf("Uso como puntero... ");
   for (i=0; i<20; i++)
     *(datos+i) = i*2;
   /* Y los mostramos */
   for (i=0; i<10; i++)
     printf ("%d ", *(datos+i));
   /* Ahora damos valores como array */
   printf("\nUso como array... ");
   for (i=0; i<20; i++)
     datos[i] = i*3;
   /* Y los mostramos */
   for (i=0; i<10; i++)
     printf ("%d ", datos[i]);
   /* Liberamos el espacio */
   free(datos);
}
```

9.7. Arrays de punteros

Igual que creamos "arrays" para guardar varios datos que sean números enteros o reales, podemos hacerlo con punteros: podemos reservar espacio para "20 punteros a enteros" haciendo

```
int *datos[20];
```

Tampoco es algo especialmente frecuente en un caso general, porque si fijamos la cantidad de datos, estamos perdiendo parte de la versatilidad que podríamos tener al usar memoria dinámica. Pero sí es habitual cuando se declaran varias cadenas:

```
char *mensajesError[3]={"Fichero no encontrado", "No se puede escribir",
  "Fichero sin datos"};
```

Un ejemplo de su uso sería este:

```
/*---------------------------*/
/*  Ejemplo en C nº 78:      */
/*  c078.c                   */
/*                           */
/*  Arrays de punteros       */
/*                           */
/*  Curso de C,              */
/*    Nacho Cabanes          */
/*---------------------------*/

#include <stdio.h>

main() {
   char *mensajesError[3]={"Fichero no encontrado",
     "No se puede escribir",
     "Fichero sin datos"};

   printf("El primer mensaje de error es: %s\n",
     mensajesError[0]);
   printf("El segundo mensaje de error es: %s\n",
     mensajesError[1]);
   printf("El tercer mensaje de error es: %s\n",
     mensajesError[2]);
}
```

9.8. Punteros y estructuras

Igual que creamos punteros a cualquier tipo de datos básico, le reservamos memoria con "malloc" cuando necesitamos usarlo y lo liberamos con "free" cuando terminamos de utilizarlo, lo mismo podemos hacer si se trata de un tipo de datos no tan sencillo, como un "struct".

Eso sí, la forma de acceder a los datos en un struct cambiará ligeramente. Para un dato que sea un número entero, ya sabemos que lo declararíamos con `int *n` y cambiaríamos su valor haciendo algo como `*n=2`, de modo que para un struct podríamos esperar que se hiciera algo como `*persona.edad = 20`. Pero esa no es la sintaxis correcta: deberemos utilizar el nombre de la variable y el del campo, con una flecha (->) entre medias, así: `persona->edad = 20`. Vamos a verlo con un ejemplo:

```
/*--------------------------*/
/*  Ejemplo en C nº 79:     */
/*  c079.c                  */
/*                          */
/*  Punteros y structs      */
/*                          */
/*  Curso de C,             */
/*    Nacho Cabanes         */
/*--------------------------*/

#include <stdio.h>

main() {
   /* Primero definimos nuestro tipo de datos */
   struct datosPersona {
     char nombre[30];
     char email[25];
     int edad;
   };

   /* La primera persona será estática */
   struct datosPersona persona1;
   /* La segunda será dinámica */
   struct datosPersona *persona2;

   /* Damos valores a la persona estática */
   strcpy(persona1.nombre, "Juan");
   strcpy(persona1.email, "j@j.j");
   persona1.edad = 20;

   /* Ahora a la dinámica */
   persona2 = (struct datosPersona*)
     malloc (sizeof(struct datosPersona));
   strcpy(persona2->nombre, "Pedro");
   strcpy(persona2->email, "p@p.p");
   persona2->edad = 21;

   /* Mostramos los datos y liberamos la memoria */
   printf("Primera persona: %s, %s, con edad %d\n",
     persona1.nombre, persona1.email, persona1.edad);
   printf("Segunda persona: %s, %s, con edad %d\n",
     persona2->nombre, persona2->email, persona2->edad);
   free(persona2);
}
```