TEMA 6. Punteros

1. Introducción

Como ya alguna vez se ha dicho en este manual, el lenguaje C es un lenguaje de programación de alto nivel pero que por razones históricas[1] guarda importantes reminiscencias de lenguajes de bajo nivel. Esto quiere decir por ejemplo, que en C se puede acceder a las zonas de memoria donde se guardan las variables o trabajar al nivel de bytes sobre la memoria que el programa reserva sobre la RAM. El tema de los punteros en C es, con gran diferencia, el más difícil de entender del lenguaje C. Esta dificultad viene dada porque intrínsecamente lo es, sus creadores no estuvieron muy afortunados en el uso de los operadores, pero fundamentalmente es debida al mal uso (abuso en numerosos casos) que los libros de texto y algunos profesores hacen de los punteros. Así es bastante común usar los punteros como herramienta de acceso y definición de arrays, sin otra motivación más que hacer su comprensión más difícil.

Por tanto, debemos advertir que en numerosas ocasiones podremos ver en libros de texto y apuntes de C un uso de punteros absolutamente innecesario. Asimismo en numerosos cursos de enseñanza universitaria de programación se alienta su uso, sin motivo aparente, consiguiendo únicamente hacer más abstrusa su impartición. Por ejemplo, es habitual encontrar que las cadenas de caracteres se declaran como punteros a char, lo cual es propio de malos programadores y peores docentes.

Esto no significa que no sea necesario dar los punteros en C, lo que quiere decir es que su uso debe limitarse a los casos donde son imprescindibles, sobre todo si estamos dando un asignatura de programación y no una asignatura de arquitectura de ordenadores o sistemas operativos donde es posible que su uso esté más justificado. Como estas asignaturas no son materia de este manual nos limitaremos a explicar el uso de los punteros en el paso de parámetros a funciones, que es dónde realmente son imprescindibles en un curso de programación de alto nivel.

2. Definición de puntero.

Un puntero es la forma coloquial en la que se denomina en C a la dirección de memoria donde se almacena una variable. Cuando definimos una variable en C, por ejemplo:

```
int i=4;
```

el compilador reserva una zona de la memoria RAM para guardar el valor de i (un cuatro en este caso), de forma que dependiendo del tipo de variable esa zona tendrá más o menos bytes. El programador de ordenadores no necesita habitualmente saber cuál es la dirección de memoria de i, le basta saber que cada vez que use la variable i, el

[1] https://es.wikipedia.org/wiki/C_(lenguaje_de_programación)

compilador “acudirá” a la dirección de memoria y obtendrá (si interviene en alguna expresión) o cambiará (si está a la derecha de una asignación o en una sentencia scanf) el valor de i.

El lenguaje C proporciona un mecanismo para acceder al valor de la dirección de memoria donde se ha guardado una variable, y lo hace mediante el operador & delante del nombre de la variable en cuestión. Así &i es el valor de la dirección donde se guarda i. De esta manera C es capaz de trabajar con zonas de memoria e incluso definir variables que contengan direcciones de memoria. Estas son las variables de tipo puntero que se definen mediante el operador *. De esta forma:

```
int *direc;
```

define una variable que es capaz de contener como valor direcciones de memoria. En este caso direcciones de memoria que pueden contener a un valor entero. Por lo tanto tenemos dos símbolo & y * que normalmente suelen ser confundidos. Aclaremos su uso:

El operador & se aplica sobre variables “normales” (las que hemos definido hasta ahora) y de esta manera obtenemos la dirección de memoria donde se almacena la variable. Supongamos que las direcciones de memoria se identifican por dos números de 4 cifras en hexadecimal separados por dos puntos. Sea la variable i de tipo int con un valor de 4, y sea 003E:F7E0 la dirección de memoria de i donde está almacenado el 4 ¿Cuánto vale i? Su valor es 4. ¿Cuánto vale &i? Su valor es 003E:F7E0. Y si queremos almacenar la dirección de memoria de i en una variable, ¿qué se puede hacer? Pues definir una variable puntero como direc y entonces podemos hacer

```
direc = &i;
```

Por tanto, a la pregunta de ¿qué valor guarda direc? La respuesta es 003E:F7E0. Por tanto, el símbolo & devuelve una dirección de memoria de una variable que puede ser almacenada sólo en variables de tipo puntero definidas mediante el símbolo *.

La pregunta clave, sin embargo es la siguiente: ¿Qué necesidad tiene un programador que usa el C como lenguaje de alto nivel de hacer uso de las direcciones de memoria de una variable? La siguiente sección explica por qué y cuándo.

3. Paso de parámetros por referencia.

Hasta ahora todas las funciones que hemos hecho sólo pueden devolver un valor en el nombre de la función. Algunas hemos hecho que modifican también un array pero eso ha sido porque ya demostramos en la sección 2.4.1 del Tema 4 que cuando un array era pasado como argumento a una función, si se modificaba el array que hacía de parámetro

formal, quedaba modificado el array de la invocación o parámetro real. ¿Cómo hacer que una función pueda devolver dos datos distintos?

Por ejemplo, planteemos el siguiente ejercicio. Construya una función que reciba dos variables de tipo int y que al finalizar las dos variables tengan sus valores intercambiados. Esto es si escribimos:

```
void main(void){
      int m=8, n=7;

      printf (" Antes: m = %d n=%d\n",m,n);
      intercambia(m,n);
      printf (" Después: m = %d n=%d\n",m,n);
}
void intercambia (int a, int b){
      int temp;

      temp = a;
      a = b;
      b = temp;
}
```

El programa al ejecutarse debería dar como salida:

```
Antes: m = 7 n = 8
Después: m = 8 n = 7
```

Sin embargo no lo hace. ¿Por qué? Pues no intercambia los valores de m y n porque los parámetros formales a y b reciben una COPIA de los parámetros reales m y n, y entonces aunque a y b sí intercambian sus valores de forma que al final de la función a valdría 8 y b 7, los valores de m y n no sufren ningún cambio. ¿Cómo arreglarlo? Pues haciendo que los parámetros formales de la función intercambia no reciban una copia de los valores de m y n, sino las direcciones de memoria donde están los valores de m y n. De esta manera, la función intercambia tendría acceso a poder cambiar los valores de m y n. ¿Cómo se hace para que intercambia reciba direcciones de memoria? Pues en principio, hay que cambiar la invocación ya que ahora no sería intercambia(m,n); sino intercambia (&m,&n); y por supuesto la cabecera de la función ya que ahora a y b no serían variables de tipo int sino variable punteros de tipo int:

```
void intercambia (int *a, int *b){…}
```

¿Y eso es todo? Pues por desgracia no. Tenga en cuenta que a y b son variables dirección de memoria y que en el cuerpo de la función intercambia las direcciones de memoria sirven de poco ya que necesitamos los valores que están en esas direcciones. Esto es, necesitamos acceder al 8 y al 7. ¿Cómo se hace? Pues accediendo al contenido

de una dirección de memoria. Para ello, C proporciona un operador que actuando sobre una variable dirección de memoria devuelve su contenido. Este operador es *[2]. De esta forma *a devuelve el valor almacenado en la dirección a. El código queda entonces:

```
#include <stdio.h>

void intercambia (int *, int *);
void main(void){
      int m=8, n=7;

      printf (" Antes: m = %d n=%d\n",m,n);
      intercambia(&m,&n);
      printf (" Después: m = %d n=%d\n",m,n);
}
void intercambia (int *a, int *b){
      int temp;

      temp = *a;
      *a = *b;
      *b = temp;
}
```

Nótense varias cuestiones:

- Las variables m y n son "normales", solamente se pone & delante en la invocación a la función intercambia.
- Las variables a y b son variables puntero, esto es contienen direcciones de memoria. Por eso cuando se usan en el cuerpo de la función, llevan el operador * por delante, ya que no nos interesa los valores de a y b, sino el contenido de esas direcciones de memoria.
- La variable temp es también "normal", sirve para almacenar el contenido de a y después pasárselo a b.
- En el programa principal no hay *.
- En la función no hay &.
- Si ejecuta este programa m y n tendrán sus valores cambiados.

¿Cómo funciona este programa?

1. Define dos variables m y n, guarda en la dirección de m un 8 y en la de n un 7.
2. Invoca a la función intercambia de forma que se pasan como argumentos las direcciones de memoria de m y n. Supongamos que la dirección de memoria de m es 0041:FB24 (que tiene un 8 almacenado) y la de n 0041:FB30 (que tiene un 7).

[2] Nota del autor: Desde mi punto de vista un grave error de los creadores del C ya que un mismo símbolo * tiene dos funciones relacionadas pero muy distintas, lo cual crea una gran confusión sobre su uso. Como se verá más adelante C++ soluciona este error de diseño, precisamente eliminando el uso del * en el paso de parámetros.

3. Cuando el control de ejecución pasa a la función intercambia el valor de a es 0041:FB24 y el de b 0041:FB30.
4. La sentencia temp = *a; hace que la variable temp cambie su valor por el valor almacenado en 0041:FB24, esto es temp pasa a valer 8.
5. La sentencia *a = *b; hace que el valor almacenado en 0041:FB24 (un 8) cambie por el valor almacenado en 0041:FB30 (un 7). Por tanto, ahora 0041:FB24 almacena un 7.
6. La sentencia *b = temp; hace que la dirección de memoria 0041:FB30 guarde el valor de temp, esto es un 8.
7. Por tanto, la dirección 0041:FB24 almacena un 7 y la dirección 0041:FB30 un 8.
8. Termina intercambia y el control vuelve al programa principal, ¿Qué valor tiene m? El que esté almacenado en su dirección de memoria 0041:FB24 esto es un 7 y ¿Qué valor tiene n? El que está guardado en 0041:FB30, es decir, un 8.

4. Reglas para un buen uso

Las siguientes reglas resumen un buen uso de los operadores * y &. En este manual el uso de las variables de tipo puntero queda restringido a los argumentos de entrada/salida en las funciones que tengan como salida más de un valor. La principal dificultad está en distinguir los dos usos diferentes del símbolo * y el saber diferenciar cuándo una variable es de tipo "normal" y cuando un puntero.

- El símbolo * se usa para DEFINIR una variable de tipo puntero.
- El símbolo & se usa para indicar la dirección de una variable "normal".
- El símbolo * se usa para ACCEDER al valor que guarda un puntero.
- El símbolo & se usa sólo en la invocación de funciones, de forma que se pone delante de los parámetro reales de E/S[3].
- Cuando un parámetro formal es de E/S en el prototipo y cabecera se indica con un tipo puntero, esto es, poniendo un * entre el tipo y el nombre del parámetro.
- En el cuerpo de la función los argumentos formales que son parámetros de E/S se usan con un * delante.
- Los & sólo se usan en el programa principal.
- Los * sólo se usan en las funciones.

4.1 Ejercicio. Construya una función que reciba tres argumentos reales con los coeficientes de una ecuación de 2º grado y devuelva las dos raíces solución de la misma. Para simplificar suponga que el discriminante de la ecuación ($b^2 \times 4ac$) es positivo.

- Entrada: 3 valores reales con los coeficientes.
- Salida: 2 valores reales con la solución de la ecuación.

[3] Ahora se puede entender por qué la función scanf necesita ponerle un & a las variables que se van a leer. Si scanf va a leer una variable necesita que sea argumento de E/S y por eso en la invocación la variable va antecedida de un &.

Como la función tiene dos argumentos de salida debemos de recurrir al uso de direcciones de memoria para implementar los argumentos formales. Por tanto la función quedaría[4]:

```
void solu_ec2grado(double p,double q,double r,double *x,double *y){
      *x=(-q+sqrt(pow(q,2)-4*p*r))/(2*p);
      *y=(-q-sqrt(pow(q,2)-4*p*r))/(2*p);
}
```

Y una posible invocación para resolver $x^2 - x - 6 = 0$ sería:

```
void main(void){
      double a,b,c,x1,x2;

      a=1.0;
      b=-1.0;
      c=-6.0;

      solu_ec2grado(a,b,c,&x1,&x2);

      printf("las soluciones son x1= %lf x2=%lf\n",x1,x2);
}
```

5. Paso de Parámetros en C++

El C++ es una extensión de C para el paradigma de la programación orientada a objeto. Sin embargo, el C++ no es un lenguaje OO puro, sino que permite que pueda ser usado como lenguaje imperativo o estructurado. En numerosos cursos de iniciación a la programación se estudia como lenguaje C++, pero sin usar el paradigma OO. Este lenguaje es conocido coloquialmente como C± (léase C más menos). Algunas de las ventajas del C++ con respecto al C son el uso de los operadores cin y cout para leer o escribir datos, en vez de scanf y printf, mucho más engorrosos por el formato.

Sin embargo, la principal ventaja del C++ es la simplificación del uso de parámetros de entrada/salida en las funciones. Como ejemplo veamos como quedarían las dos funciones intercambia y solu_ec2grado en C++[5]:

```
void intercambia2 (int &x, int &y){
      int z;

      z=x;
      x=y;
      y=z;
}
```

[4] No olvide incluir math.h para poder hacer uso de las funciones pow y sqrt

[5] El compilador de Visual C compila indistintamente programas C ó C++, dependiendo de la extensión de los archivos que contenga el código sea .c ó .cpp. Por defecto, usa el compilador de C++.

```
void solu_ec2grado2(double p,double q,double r,double &x,double &y){

      x=(-q+sqrt(pow(q,2)-4*p*r))/(2*p);
      y=(-q-sqrt(pow(q,2)-4*p*r))/(2*p);
}
```

Nótese que la simplificación es enorme, el símbolo * ha desaparecido y sólo el símbolo & delante de un argumento formal indica que es un parámetro de E/S. Sin embargo, en el cuerpo de la función las variables x e y se usan como unas variables más.

La invocación también se simplifica:

```
void main(void){
      double a,b,c,x1,x2;
      int m=8, n=7;
      a=1.0;
      b=-1.0;
      c=-6.0;

      intercambia2(m,n);
      printf("m=%d n=%d\n",m,n);

      solucion_ec2grado2(a,b,c,x1,x2);
      printf("x1= %lf x2=%lf\n",x1,x2);
}
```

En la invocación a las funciones, los parámetros reales se escriben sin ningún tipo de operador, y el compilador entiende que al ser argumentos de entrada/salida (por tener un & en el prototipo) los valores de m y n ó x1 y x2 que deben ser transferidos a las funciones son sus direcciones de memoria.

6. Relación entre punteros y arrays.

6.1 Antecedentes. Hay dos cuestiones que ya se han tenido en cuenta en este manual y que no han sido explicadas. Una es el hecho de que al leer una variable de tipo cadena de caracteres con scanf hemos señalado que, a diferencia con el resto de los tipos, no se ponía & delante del nombre de la variable de tipo Cadena. La segunda cuestión es que demostramos en su momento que un array cuando era pasado como argumento a una función, era argumento de entrada/salida sin necesidad de poner & ó *. ¿Cuál es la explicación para esto? La explicación es que en C el nombre de un array (recuerde que las cadenas de caracteres lo son) es realmente una dirección de memoria (un puntero) y por tanto no debe usarse con los arrays el operador &. De esta manera, si tenemos definidos el tipo Cadena o el tipo TablaReales como en temas anteriores, podemos escribir:

```
      Cadena s;
```

```
      TablaReales v;

      scanf("%s",s);
      leeVector(v,n);
```

Donde las variables s y v no tienen ningún & delante, aunque son argumentos de entrada/salida tanto en la función scanf como en leeVector.

6.2 Abuso de punteros. El principal abuso de los punteros en C viene dado porque el nombre de una variable de tipo array es realmente la dirección de memoria del primer elemento de un array. Para explicarlo pongamos un ejemplo:

Definamos la variable tabla como un array para cinco enteros inicializándolo:

```
int tabla[5]={7,4,2,8,5};
```

Si usamos el formato %u ó %p (formato para visualizar direcciones de memoria) con la sentencia printf y ejecutamos:

```
printf("%u\n",tabla);
printf("%p\n",tabla);
```

El resultado sería similar a éste:

```
4126688
003EF7E0
```

que son dos formas distintas de referenciar una dirección de memoria. ¿Qué quiere decir esto? Pues que en la dirección de memoria 003E:F7E0 se almacena el valor 7 o primer elemento de la tabla. Es decir, en C se cumple estas igualdades:

```
tabla = &tabla[0];
*tabla = tabla[0]
```

Como ya se señaló en el Tema 4, las posiciones de un array ocupan posiciones consecutivas, lo que facilita lo que se conoce como "aritmética de punteros", esto es, es posible "avanzar" en las posiciones de un array incrementando el valor de su nombre. De esta manera:

```
tabla+1 = &tabla[1];
*(tabla+1) = tabla[1]
```

Con esto, los típicos bucles for para tratar un array podrían quedar así[6]:

```
	for(i=0;i<n;i++){
		*(v+i) = i*i;
	}

	for (i=0;i<n;i++){
		suma = suma + *(v+i);
	}
```

6.3 Definiciones de arrays como punteros. La propiedad anterior da lugar a que en numerosos libros de texto, se definan por ejemplo cadenas de caracteres como punteros a char, que según hemos visto son equivalentes. Esto es, las siguientes declaraciones de la variable s son "aparentemente" equivalentes:

```
	char s[10];
	char *s;
	char s[];
```

¿Por qué son equivalentes? En la primera declaración s es un array pero sabemos que también es una dirección de memoria para guardar un char (el primero del array). Y qué es la segunda declaración, pues s es la dirección de memoria para guardar un char. Es decir, exactamente lo mismo. Finalmente la tercera declaración, es similar pero no se está diciendo cuál es el tamaño del array. Sin embargo, no son totalmente equivalentes, porque la primera sí está reservando una cantidad de memoria concreta, mientras que las otras dos no. Esto en numerosas ocasiones puede dar problemas en tiempo de ejecución, al intentar acceder a zonas de memoria que no están reservadas y que pueden ser de otras variables del programa.

Pero no solo con cadenas de caracteres se da esta posibilidad. También con arrays de cualquier tipo es posible definirlos como int *. Por ejemplo, una función para sumar los elementos de un array de enteros podría ser escrita así:

```
int suma (int *t, int n){
	int i;
	int suma =0;

	for(i=0;i<n;i++){
		suma = suma + t[i];
		// también suma = suma + *(t+i);
	}
	return suma;
}
```

[6] Juzgue el lector la "necesidad" de hacer un código tan poco inteligible

Definir un parámetro formal de esta manera no da problemas de memoria porque t solamente referencia a la dirección del parámetro real cuyo tamaño debe ser pasado también como argumento. Otra cuestión muy distinta y que también se puede ver en libros de texto es definir en el programa principal arrays mediante el uso de *. Como vemos es posible tratar los arrays con la declaración y la aritmética de punteros. No hay ninguna razón para ello, más allá de crear un halo de dificultad sin necesidad. A pesar de ello, en numerosos libros de texto y cursos universitarios de programación se siguen usando. Por nuestra parte, desaconsejamos totalmente esta forma de trabajar con arrays en C y recomendamos categóricamente definir los arrays mediante su tipo en una clausula typedef. En un curso de programación, el alumno se debe abstraer de los problemas concretos que el lenguaje C tiene por ser un lenguaje con más de 40 años y cuyo propósito de diseño no fue aprender a programar.

7. Problemas.

1. Construya una función que recibe dos números enteros correspondientes a minutos y segundos y devuelve tres enteros representando horas, minutos y segundos que corresponden. Por ejemplo, si se dan 123 minutos y 80 segundos, la salida de la función sería 2 horas, 4 minutos y 20 segundos.

2. Construya una función que reciba un array de números reales y devuelva el máximo y la posición que ocupa.

3. Construya una función que reciba una matriz de reales y devuelva el máximo y el mínimo de sus elementos.

4. Construya una función que lea dos enteros con el número de filas y columnas de una matriz de reales y que posteriormente lea desde teclado los elementos de dicha matriz que debe ser un valor de salida de la función.

5. Construya una función que dada una matriz de reales y un valor devuelva la posición (fila y columna) que ocupa y si no está (-1,-1).

6. Usando la función intercambia implemente una función que reciba una MatrizReal m y dos números de fila f1 y f2, y devuelva m modificada (no otra matriz, sino la misma m) de forma que los elementos de las filas f1 y f2 estén intercambiadas. Por ejemplo, si intercambiamos las filas 1 y 3:

$$m = \begin{pmatrix} 2 & 2 & 0 & 5 & 6 \\ 3 & 1 & 6 & 3 & 4 \\ 9 & 2 & 3 & 5 & 8 \\ 3 & 5 & 6 & 8 & 0 \end{pmatrix} \rightarrow m = \begin{pmatrix} 2 & 2 & 0 & 5 & 6 \\ 3 & 5 & 6 & 8 & 0 \\ 9 & 2 & 3 & 5 & 8 \\ 3 & 1 & 6 & 3 & 4 \end{pmatrix}$$

7. Implemente una función tal que dada una MatrizReal devuelva el valor mínimo y la primera posición (fila y columna) donde se encuentra. Por ejemplo en la matriz m anterior devolvería como mínimo 0 y como posición fila 0 columna 2.

8. Escriba una función de nombre filtraParesYSuma, tal que dado un array de enteros construya un array con los números de las posiciones pares, es decir, un array filtrado. Además, la función deberá devolver la suma de las posiciones pares. Por tanto, la función deberá devolver dos resultados: la suma y el número de elementos del nuevo array. Tiene tres combinaciones para resolver este problema que corresponden con las siguientes cabeceras. Haga las dos primeras.

```
void filtraParesYSuma(TablaEnteros,int,TablaEnteros, int *, int *);
int filtraParesYSuma(TablaEnteros,int,TablaEnteros,int *);
   // la función devolverá en su nombre la suma
int filtraParesYSuma(TablaEnteros,int,TablaEnteros,int *);
   // la función devolverá en su nombre el número de elementos del
array filtrado
```

9. En el programa principal declare e inicialice un array de enteros e invoque la función del ejercicio anterior. Si no obtiene el resultado esperado utilice el depurador.

10. Realice una función de nombre divideSecuenciaNucleotidicaEnDos, que dado un array con la secuencia nucleotídica del genoma de un organismo (array de caracteres) obtenga dos arrays con la división en dos mitades iguales de la secuencia. Es decir, el primer array contiene la primera mitad de la secuencia y el segundo array contiene la segunda mitad de la secuencia. Estudie si influye en el código que el número de elementos de la secuencia original sea par o impar.

11. En el programa principal invoque la función anterior y muestre por pantalla las dos mitades resultantes a partir de una secuencia nucleotídica creada previamente.

12. Tenemos un array de valores enteros con las 21 medidas de las pulsaciones de un paciente tomadas 3 veces al día (mañana, tarde y noche) durante una semana.

    a) Implemente una función tal que dado un índice del array (un valor entero entre 0 y 20) devuelva el día y el momento a que corresponde la medición. Por ejemplo: el índice 0 es la mañana del primer día (devuelve día=1 y tiempo=1), el índice 2 es la última medición tomada del primer día (día=1, tiempo=3), el índice 6 es la medición del tercer día por la mañana (día=3, tiempo=1), etc. Utilice los operadores / y % para este ejercicio.

    b) Escriba una función tal que dado un array con los pulsos muestre en pantalla una secuencia como la siguiente, invocando la función anterior:

Las pulsaciones del día 1 por la mañana fueron ¿?
Las pulsaciones del día 1 por la tarde fueron ¿?
Las pulsaciones del día 1 por la noche fueron ¿?
Las pulsaciones del día 2 por la mañana fueron ¿?
…..
…..