Departamento de Ingeniería Telemática y Electrónica

Programación C
Arrays y estructuras

Telecomunicación Campus Sur UPM

(Versión de fecha 16 de octubre de 2024)

ÍNDICE

Los tipos de datos básicos, también llamados intrínsecos o elementales, son los números enteros, los números con decimales y los caracteres. Son tipos de datos que dependen del procesador y todos los lenguajes de programación los implementan y permiten operar con ellos.

Estos tipos de datos básicos se pueden agrupar de diferentes maneras para obtener estructuras de datos más complejas. Cada lenguaje de programación ofrece sus propios conjuntos de datos complejos, si bien, hay determinados tipos que suelen estar implementados en todos los lenguajes.

Uno de estos tipos es el denominado array, que consiste en una colección ordenada de elementos del mismo tipo a los que se accede a través de uno o más índices.

Otro tipo muy utilizado son las denominadas en C estructuras, que son un conjunto de variables agrupadas bajo un mismo nombre, que pueden ser de diferente tipo. A los elementos que componen la definición de una estructura se les denomina campos. A cada variable que se crea de un tipo de estructura concreta se le suele denominar un objeto o una instancia de la estructura.

En este documento se va a explicar cómo utilizar en C los arrays, unidimensionales o multidimensionales y las estructuras.

1 Arrays unidimensionales: vectores

Un *array* unidimensional es una colección ordenada de valores del mismo tipo agrupados bajo el mismo nombre de variable y que permite acceder a los valores individuales a través de un índice. La sintaxis para declarar un *array* es la siguiente:

```
tipo_datos nombre[numero_de_elementos];
```

Es habitual llamar *vectores* a los *arrays* unidimensionales. A cada uno de los valores del array se le denomina *elemento* o *componente* del *array*. Al número de elementos del array se le denomina *tamaño* o *dimensión* del *array*. Una vez declarado el tamaño de un array, no es posible modificarlo. Tampoco es posible cambiar el tipo de datos de los elementos del array.

El siguiente código declara un *array* llamado *posiciones* que permite guardar 3 números enteros y un *array* llamado *valores* que permite almacenar 10 números del tipo *double*:

```
int posiciones[3];
double valores[10];
```

El número de elementos del *array* se puede declarar mediante un valor entero literal, como se ha hecho en los dos ejemplos anteriores. También se puede utilizar un literal que haya sido definido en una directiva `#define`, como se hace en el siguiente ejemplo:

```
#define DIM 10

double vector_1[DIM];
```

El código anterior define una constante llamada *DIM* de valor 10 y luego crea un *array* de 10 elementos. A todos los efectos, la utilización del valor `DIM` declarado en la directiva `#define` equivale a haber utilizado directamente el valor literal 10 en la declaración del *array*: durante el preprocesamiento previo a la compilación, en todos los lugares donde aparece `DIM`, se sustituye por su valor 10.

Cuando la dimensión de un *array* se declara mediante un literal, como se ha hecho en los ejemplos anteriores, es posible inicializar sus valores en la propia declaración poniendo, tras el nombre del array, el signo $=$ y las componentes entre llaves {} y separadas por comas. Así se ha hecho en el siguiente ejemplo:

```
#define DIM 3

double v_1[DIM] = {1.0, 2.0, 3.0};
int N[4] = {-1, 3, 5, 7};
```

Cuando se asignan los valores de las componentes directamente en la declaración del *array*, no es necesario especificar el tamaño, se pueden dejar vacíos los corchetes, pues el compilador lo deduce de la declaración:

```
double v_1[] = {1.0, 2.0, 3.0};
int N[] = {-1, 3, 5, 7};
```

También es posible forzar que todos los elementos del *array* se inicialicen con el valor 0, asignando unas llaves vacías o poniendo el valor 0 entre las llaves, como se hace en el siguiente código:

```
double V[3] = {}; // Inicializa V con ceros
int A[5] = {0};   // Inicializa A con ceros
```

Un *array* se puede declarar constante utilizando la cláusula *const*. En los *arrays* declarados constantes hay que asignar valor a las componentes en el momento de la declaración y dicho valor no se podrá modificar con posterioridad. El código siguiente declara un *array* constante llamado A con 4 componentes de tipo *int*:

```
const int A[] = {10, 12, 21, 32};
```

Es posible definir el tamaño de un *array* de manera dinámica, esto es, durante la ejecución del programa. Un ejemplo podría ser cuando el tamaño del *array* es un valor proporcionado por el usuario. En ese caso, el tamaño del *array* no es conocido en el momento de la compilación y no se pueden inicializar sus valores en la declaración del mismo, como se ha hecho en los ejemplos anteriores. El código del Ejemplo 1 pide al usuario un número entero y declara un *array* de números *double* de esa dimensión:

Ejemplo 1 Declaración dinámica de la dimensión de un *array*

```
#include <stdio.h>

int main( void ){
   int dimension;
   printf("Dimensión: ");
   scanf("%d", &dimension);

   double A[dimension];
   return 0;
}
```

¡Importante!

Una vez declarada, la dimensión de un array no se puede modificar.

2 Acceso a los elementos de un vector

Para acceder al valor guardado en una de las componentes de un *array* se utiliza el nombre del *array* y, entre corchetes, el índice correspondiente al elemento al que se quiere acceder.

```
nombre_array[indice]
```

El índice del primer elemento es el cero y el del último una unidad menos que el tamaño del *array*. A todos los efectos, cada componente de un *array* se comporta como una variable del tipo de datos con el que se haya declarado el *array*. En ese sentido, se puede modificar su valor o hacerle formar parte de operaciones.

El Ejemplo 2 declara un *array* de nombre A con 5 valores enteros y le asigna unos valores iniciales. A continuación, modifica el valor de la segunda componente; por último, declara una variable x cuyo valor se asigna en función del valor de la última componente del *array* A:

Ejemplo 2 Acceso a los elementos de un *array*

```
// Se declara e inicializa el array A
int A[] = { 1, -3, 5, 0, 6};

// Se modifica el valor de la 2ª componente de A
A[1] = 2500;

// Se crea una variable x y se le asigna
// el triple del valor de la 5ª componente de A
int x = 3*A[4]; // x vale 18
```

Para recorrer uno a uno todos los elementos de un *array* unidimensional se puede utilizar un bucle *for*. El Ejemplo 3 utiliza un bucle *for* para imprimir el vector V en pantalla.

Ejemplo 3 Utilización de un bucle *for* para recorrer un *array*

```
#include <stdio.h>
#define DIM 5

int main( void ) {
   double V[DIM] = {1.0, -3.14, 6.78, -2.4, 3.6};

   printf("V = {");
   for(int i=0; i<DIM; i++) {
      printf("%.2f, ", V[i]);
   }
   printf( "\b\b}\n");

   return 0;
}
```

En el ejemplo anterior, en cada iteración del bucle se imprime el valor de la componente correspondiente seguido de una coma. Para eliminar la última coma, la sentencia *printf()* posterior al bucle utiliza dos veces el carácter especial $\backslash b$ (*back space*), antes de imprimir la llave de cierre.

3 *Arrays* multidimensionales: matrices

Es posible utilizar *arrays* con más de una dimensión. En el caso de *arrays* de dos dimensiones, se podrían interpretar como *matrices*, y su sintaxis es la siguiente:

```
tipo_datos nombre[num_filas][num_columnas];
```

Al igual que sucede con los *arrays* unidimensionales, todos los elementos del *array* tienen que ser del mismo tipo de datos. Todas las filas tienen que tener el mismo número de columnas. Es habitual también denominar *tablas* a los *arrays* de dos dimensiones.

Es posible asignar valores a las componentes del *array* en el momento de la declaración. En este caso, hay que usar un juego de llaves de apertura y cierre para la matriz, y otro juego de llaves para cada fila de la matriz; al final de las filas, excepto la última, se pone una coma y, al final de la declaración, punto y coma.

El siguiente código declara una matriz de 2 filas y 5 columnas y asigna valores a los elementos directamente en la declaración:

```
int A[][5] = {
   {11, 12, 13, 14, 15},
   {21, 22, 23, 24, 25}
};
```

Observe que, cuando se asigna valor directamente a los elementos en la declaración de la matriz, no es necesario especificar el número de filas, pudiendo dejar vacíos los corchetes correspondientes.

4 Acceso a los elementos de una matriz

En el caso de las matrices (*arrays* bidimensionales), al igual que se mencionó en el caso de los vectores (*arrays* unidimensionales), cada componente se comporta, a todos los efectos, como una variable del tipo de datos que tenga asignado el *array* y se puede utilizar su valor en expresiones en combinación con otras variables y operadores. También es posible modificar su valor, siempre que el *array* no se haya declarado como constante.

Para acceder a una componente concreta de una matriz se utilizan dos índices entre corchetes: el índice correspondiente a la fila y el de la columna. Los índices varían entre cero y una unidad menos que el número de elementos de la línea (fila o columna) correspondiente.

El código siguiente declara una matriz llamada A de números enteros con 2 filas y 3 columnas, asignando un valor inicial a sus componentes; a continuación, calcula la suma de los elementos de la primera fila y, por último, modifica el valor del elemento A_{23}:

```
int A[][3] = {
   {11, 12, 13},
   {21, 22, 23}
};
int sum = A[1][1] + A[1][2] + A[1][3];
A[2][3] = 100;
```

Para recorrer todos los elementos de una matriz ordenadamente, se pueden utilizar dos bucles anidados (un bucle dentro de otro bucle). Con el bucle exterior se van recorriendo las filas. Para cada fila, el bucle interior recorre todas las columnas, esto es, todos los elementos de la fila. En el Ejemplo 8 se mostrará cómo imprimir en pantalla una matriz.

5 La función *sizeof()*

El tipo de datos `size_t` es un tipo entero sin signo que se utiliza para representar el tamaño de los objetos en memoria, incluyendo los *arrays*. El número de bytes que utilizan los valores del tipo `size_t` depende del procesador que se utilice. Está definido en la librería `<stddef.h>`, y está garantizado que al menos utilice 2 bytes. No es necesario incluir en el programa ninguna cláusula `#include` para poder utilizar el tipo `size_t`.

La función *sizeof()* recibe como parámetro el nombre de un tipo de datos y devuelve cuántos bytes ocupan en memoria los valores de dicho tipo de datos. También se pueden pasar como parámetros a la función *sizeof()* el nombre de una variable o un valor literal de algún tipo de datos. El valor devuelto por *sizeof()* es un entero del tipo *size_t* (long unsigned integer) que, dependiendo del procesador concreto, tendrá un tamaño u otro.

La función *sizeof()* está incorporada al lenguaje, no es necesario utilizar ninguna sentencia *#include* en la cabecera del programa.

Se puede utilizar la propia función *sizeof()* para saber el tamaño del tipo *size_t* en el ordenador concreto que

se esté utilizado, como hace el Ejemplo 4:

Ejemplo 4 Tamaño del tipo *size_t* en un ordenador concreto

```
#include <stdio.h>

int main() {

   // El tamaño del tipo size_t depende del procesador.
   // En el ordenador del autor, con procesador de 64 bits,
   // los valores del tipo size_t ocupan 8 bytes
   printf("size_t -> %lu \n", sizeof(size_t)); // 8

   return 0;
}
```

En el código del Ejemplo 4 puede observar que, para imprimir la salida de la función *sizeof()*, se ha utilizado un especificador de formato `%lu` (*long unsigned integer*).

El código del Ejemplo 5 muestra varios casos de uso de la función *sizeof()*.

Ejemplo 5 Ejemplos de resultados ofrecidos por la función *sizeof()*

```
#include <stdio.h>

int main() {

   printf("int -> %lu \n", sizeof(int));       // Tipo int: 4 bytes
   printf("double -> %lu \n", sizeof(double)); // Tipo double: 8 bytes
   printf("char -> %lu \n", sizeof(char));     // Tipo char: 1 byte

   printf("17 -> %lu \n", sizeof(17));         // Valor int: 4 bytes
   double x;
   printf("x -> %lu \n", sizeof(x));           // Variable double: 8 bytes
}
```

Utilizando la función *sizeof()* se puede calcular el número de elementos de los *arrays*. Por ejemplo, si A es un *array* de una sola dimensión, el número de elementos se puede calcular con la siguiente expresión:

```
int num_elementos = sizeof(A) / sizeof(A[0]);
```

En el caso de las matrices (*arrays* de dos dimensiones) también es posible calcular el número de filas o el de columnas. Observe cómo se ha hecho en el Ejemplo 6:

Ejemplo 6 Cálculo de los elementos, filas y columnas de una matriz

```
#include <stdio.h>

int main() {
   int A[][4] = { {11, 12, 13, 14}, {21, 22, 23, 24} };

   int num_elementos = sizeof(A) / sizeof(A[0][0]);
   int num_filas = sizeof(A) / sizeof(A[0]);
   int num_columnas = num_elementos / num_filas;
   printf("Elementos= %d Filas= %d Columnas= %d \n",
      num_elementos, num_filas, num_columnas);
}
```

6 Ejemplos de algoritmos básicos con *arrays*

6.1. Imprimir un *array* en pantalla

Imprimir un *array* en pantalla es un buen ejercicio para aprender a recorrer un *array*, elemento a elemento. Como se ha visto anteriormente, para recorrer un *array* unidimensional (vector) se suele utilizar un bucle *for*, mientras que para recorrer un *array* bidimensional (matriz) se suelen usar dos bucles *for* anidados.

El Ejemplo 7 imprime en pantalla un vector de números *double* que ha declarado previamente. Los elementos del *array* se imprimen en una línea, con dos decimales y separados por un espacio. Al finalizar el bucle, se imprime un cambio de línea. La salida se muestra a la derecha del código.

Ejemplo 7 Impresión en pantalla de un vector

```c
#include <stdio.h>

int main() {
   double V[] = {1.0, -3.7, 2.3, 4.1, 6.5};

   for(int i=0; i<5; i++) {
      printf("%.2f ", V[i]);
   }
   printf("\n");
}
```

```
1.00 -3.70 2.30 4.10 6.50
```

El Ejemplo 8 es similar al anterior pero, en este caso, se imprime una matriz de 2×3. La salida por pantalla se muestra a la derecha del código. Observe detenidamente el planteamiento de los dos bucles anidados, es importante que comprenda perfectamente su funcionamiento. El bucle exterior utiliza la variable de índice i para recorrer las filas. Para cada valor de i, se realiza un ciclo completo del bucle interior, con variable de índice j. La sentencia *printf()* que hay dentro del bucle interior se ejecuta $2 \times 3 = 6$ veces, tomando en cada caso los siguientes valores:

```
(i=0, j=0)     (i=0, j=1)     (i=0, j=2)
(i=1, j=0)     (i=1, j=1)     (i=1, j=2)
```

Observe que, dentro del primer bucle y tras finalizar cada ciclo completo del bucle interior, se ejecuta una instrucción *fprintf("\n")*, para realizar el cambio de línea entre filas.

Ejemplo 8 Impresión en pantalla de una matriz

```c
#include <stdio.h>

#define FILAS 2
#define COLS  3

int main() {
   double A[][COLS] = {
      {1.0, -3.7, 2.3},
      {4.1, 6.5, -7.0}
   };

   for(int i=0; i<FILAS; i++) {
      for(int j=0; j<COLS; j++) {
         printf("%.2f ", A[i][j]);
      }
      printf("\n");
   }
}
```

```
1.00 -3.70 2.30
4.10 6.50 -7.00
```

6.2. Copia de un *array* en otro

En este caso se pretende crear una copia de un *array* existente. Habrá que crear una variable *array* para la copia, del mismo tamaño y con el mismo tipo de datos que el *array* original. El código del Ejemplo 9 declara y

define una variable del tipo vector de 3 números *double* llamado V y lo copia en otra variable del mismo tipo y tamaño llamada W. Antes de hacer el bucle de la copia, es necesario declarar el vector W con el mismo tamaño y tipo de datos que V.

Ejemplo 9 Copia de un vector en otro

```
int main() {
   double V[] = {1.0, -1.0, 1.0};
   double W[3];

   for(int i=0; i<3; i++) {
      W[i] = V[i];
   }
}
```

En este caso, por simplificar el código, no se ha realizado ninguna salida del programa por pantalla. El lector puede probar a ejecutar el programa a través del depurador (*debugger*), de forma que pueda comprobar el contenido de las variables antes y después de la copia. Para ello, deberá marcar dos *breakpoints* en las posiciones que se muestran en la Figura 1.

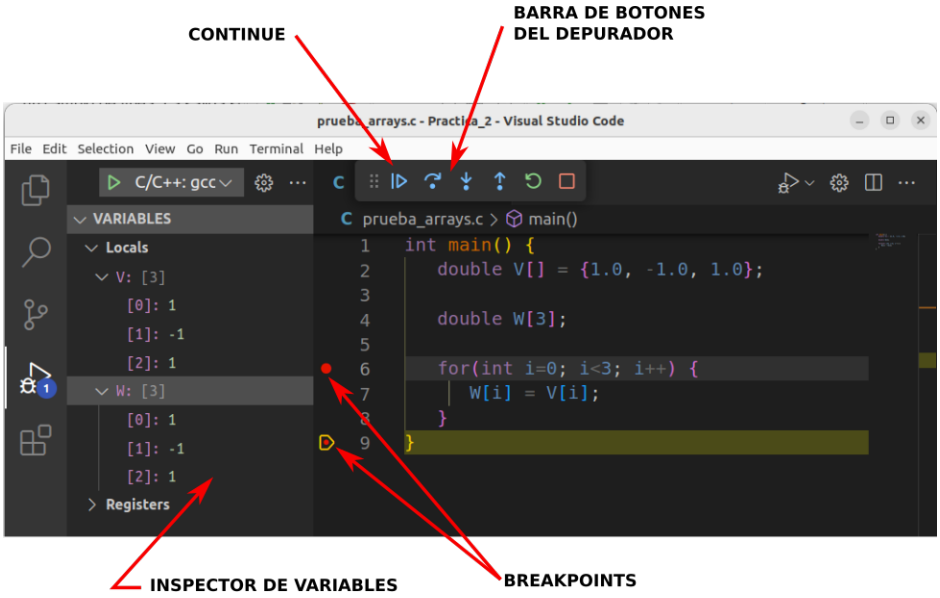

Figura 1: Ejecución del Ejemplo 9 en el depurador

Recuerde que, para marcar un *breakpoint* en una instrucción del programa, debe *pinchar* con el ratón a la izquierda del número de línea. Ejecute el programa a través de la opción de menú *Run->Start Debugging* o pulsando la tecla *F5*. Comenzará la ejecución del programa y se detendrá en el primer *breakpoint*, que se ha fijado tras la instrucción que declara el vector W. En ese momento, en el panel izquierdo de la ventana de VSCode, puede inspeccionar el contenido de las variables y comprobar que la variable V tiene sus tres componentes, mientras que la variable W tiene valores basura, pues aún no se han asignado valores. A continuación pulse *continue*: la flecha azul que aparece en la parte superior del editor, o la tecla *F5*. El programa continuará hasta el siguiente *breakpoint* que se ha fijado justo antes de acabar el programa. Si vuelve a inspeccionar el contenido de las variables, verá que ahora las componentes de W si contienen la copia de las de V.

Para copiar una matriz en otra, el procedimiento es similar, pero en este caso se utilizan dos bucles anidados. El código del Ejemplo 10 muestra cómo hacerlo. Al igual que se ha sugerido en el ejemplo anterior, puede utilizar el depurador para observar el comportamiento del programa. Si sitúa los puntos de ruptura en los sitios

adecuados, podrá ver en qué orden se van rellenando las componentes de la matriz copia.

Ejemplo 10 Copia de una matriz en otra

```
int main() {
   int A[][3] = {
      {1, 2, 3},
      {4, 5, 6}
   }

   int B[2][3];

   for(int i=0; i<2; i++) {
      for(int j=0; j<3; j++) {
         B[i][j] = A[i][j];
      }
   }
}
```

6.3. Invertir un vector

El problema que se pretende resolver es el de hacer una copia de un vector, pero invirtiendo el orden de las componentes, esto es, la última componente del vector original pasará a ocupar la primera posición en la copia, la penúltima componente del vector original pasará a ocupar la segunda posición de la copia y así sucesivamente.

El código del Ejemplo 11 muestra cómo se podría hacer. Es importante fijarse en cómo se plantean dentro del bucle los índices de las componentes del vector original y del vector copia. El vector original tiene 6 componentes, luego la última componente del vector original, $V_5 = V_{5-0} = V_{DIM-1-i}$, se copia en la componente W_0 del vector copia. La componente $V_4 = V_{5-1} = V_{DIM-1-1}$ del vector original se copia en la W_1, y así sucesivamente. Una vez más, puede ejecutar el programa a través del depurador para ir viendo los valores sucesivos que toman las variables.

Ejemplo 11 Copia invertida de un vector

```
#define DIM 6

int main() {
   double V[] = {1.0, -1.0, 3.5, -4.2, 6.3, 0.5};

   double W[DIM];

   for(int i=0; i<DIM; i++) {
      W[i] = V[DIM-1-i];
   }
}
```

6.4. Traspuesta de una matriz

La matriz traspuesta es una operación elemental del Álgebra que consiste en obtener una matriz a partir de otra, en la que se cambian filas por columnas, esto es, las filas de la matriz original pasan a ser las columnas de la matriz traspuesta. Llamando A a la matriz original, la operación trasponer se denota con A^T. Si se llama B a la matriz traspuesta de A, la expresión que permite el cálculo de sus componentes es:

$$B = A^T$$
$$B_{ij} = A_{ji}$$

Si la matriz original es de dimensión $m \times n$, la matriz traspuesta será de dimensión $n \times m$. Habrá que tener esto en cuenta al crear la matriz destinada a albergar la traspuesta. El programa del Ejemplo 12 calcula la matriz traspuesta de una matriz A dada.

Ejemplo 12 Cálculo de la matriz traspuesta

```
#include <stdio.h>

#define FILAS 2
#define COLS 3

int main() {
   int A[FILAS][COLS] = { {1, 2, 3}, {4, 5, 6} };

   int B[COLS][FILAS];

   for(int i=0; i<COLS; i++) {
      for(int j=0; j<FILAS; j++) {
         B[i][j] = A[j][i];
      }
   }

   for(int i=0; i<COLS; i++) {
      for(int j=0; j<FILAS; j++) {
         printf("%d ", B[i][j]);
      }
      printf("\n");
   }
}
```

```
1 4
2 5
3 6
```

Observe el código anterior. La matrix A es de 2 filas y 3 colunas, por lo que la matriz B se ha creado con 3 filas y 2 columnas. Observe ahora los índices de los bucles anidados que resuelven la operación de trasponer: el bucle exteriorda valores a i entre 0 y $COLS$, mientras que el bucle interior da valores a j entre 0 y $FILAS$. Lo que hacen los bucles es ir rellenando la matriz B. La salida del programa se puede ver a la derecha del código.

6.5. Contar el número de elementos que cumplen determinada condición

En este caso se trata de recorrer los elementos de un *array* y contar cuántos de ellos cumplen determinada condición. Para ello se utiliza un *contador*, esto es, una variable que inicialmente vale 0 y se incrementa en una unidad cada vez que un elemento cumple la condición. La condición sera un bloque *if* dentro del bucle o de los bucles que recorren los elementos del *array*.

El Ejemplo 13 cuenta el número de elementos de la matriz A que están comprendidos en el intervalo abierto $(-3, 3)$, o sea, que son mayores que -3 y (simultáneamente) menores que 3. Conviene recordar que la expresión

algebraica $(a < x < b)$ en programación hay que escribirla $(a < x \quad \&\& \quad x < b)$.

Ejemplo 13 Algoritmo del contador

```
#include <stdio.h>

int main() {
   int A[][5] = {
      {-1, 20, -3, 5, 0},
      {4, -5, 6, -1, -7},
      {-2, -3, 4, 2, 6}
   };

   int contador = 0;

   for(int i=0; i<3; i++) {
      for(int j=0; j<5; j++) {
         if(-3<A[i][j] && A[i][j]<3) {
            contador = contador + 1;
         }
      }
   }

   printf("Números en (-3, 3): %d \n", contador); // Escribe 5
}
```

6.6. Modificación de los elementos que cumplen determinada condición

En este caso, el objetivo es modificar algunas componentes de un *array*, las que cumplan determinada condición. Para ello, habrá que recorrer uno a uno los elementos del *array* y, en cada caso, comprobar si el elemento cumple o no la condición. Si la cumple, habrá que hacer la modificación que corresponda. El programa constará de uno o dos bucles, según si se trata de un vector o una matriz, y una bifurcación *if* que permita comprobar la condición.

El código del Ejemplo 14 analiza un vector de números enteros y multiplica por 2 los elementos impares.

Ejemplo 14 Cambiar elementos que cumplen una condición

```
int main() {
   int V[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

   for(int i=0; i<10; i++) {
      if(V[i]%2 == 1) {
         V[i] = V[i] * 2;
      }
   }
}
```

Hay que recordar que la condición para que un número entero sea impar es que su resto al dividirlo por 2 sea 1. El operador $\%$ permite calcular el resto de una división de dos números enteros. El lector puede utilizar el depurador para comprobar el funcionamiento del programa. La Figura 2 muestra la salida en el depurador del ordenador del autor.

6.7. Extracción de los elementos que cumplen determinada condición

En este caso se trata de analizar los elementos de un *array* y extraer a un nuevo *array* aquellos que cumplan determinada condición. El primer problema con el que se encuentra este algoritmo es que, a priori, no sabe

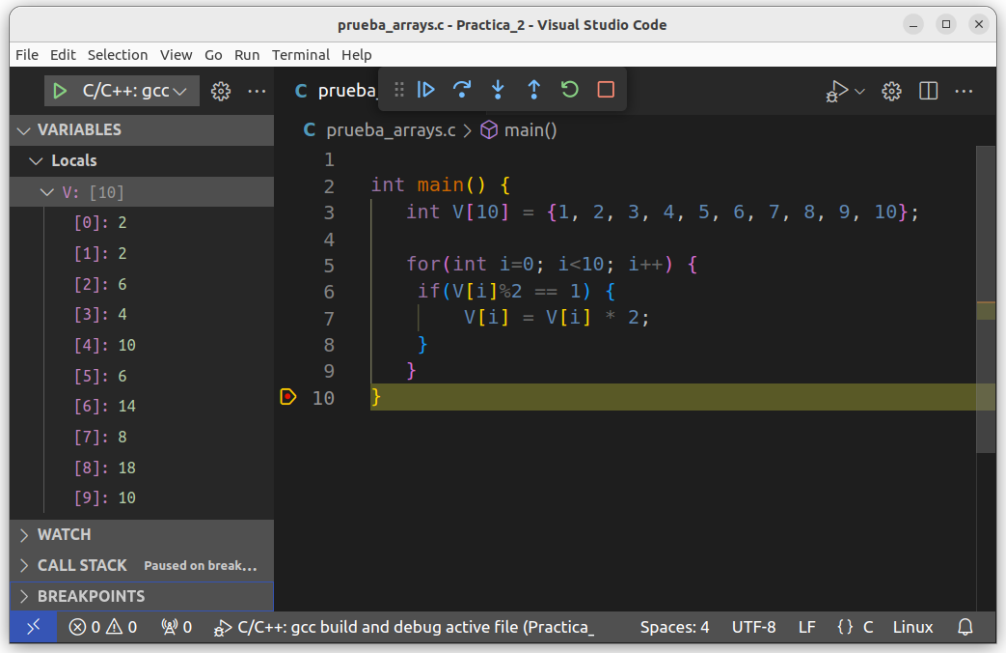

Figura 2: Salida en el depurador del Ejemplo 14

cuántos elementos cumplirán la condición y, por tanto, no sabe el tamaño del *array* que tiene que crear. El segundo problema radica en que sea un vector o una matriz el *array*, la salida habrá que darla en forma de vector, pues no tiene por qué verificar la dimensiones de una matriz.

El Ejemplo 15 analiza una matriz A y extrae a un vector todos los elementos negativos que haya. El programa realiza dos pasadas a la matriz. En la primera pasada cuenta cuántos elementos cumplen la condición; entonces, crea un vector del tamaño adecuado y realiza una segunda pasada a la matriz original extrayendo al vector los elementos de valor negativo. Observe que, con el fin de saber el índice en el que toca añadir cada elemento, necesita llevar un contador con los elementos que lleva ya añadidos.

Ejemplo 15 Extracción de elementos

```
int main() {
   int A[][4]= {
      {-1, 1, 3, 4},
      {1, -2, 5, -7},
      {6, 5, 3, -2},
      {0, 2, -3, 1}
   };

   // Primera pasada: contar
   int num_negativos=0;
   for(int i=0; i<4; i++) {
      for(int j=0; j<4; j++) {
         if(A[i][j]<0) {
            num_negativos++;
         }
      }
   }

   // Declarar vector para albergar negativos
   int W[num_negativos];
```

```
   // Segunda pasada: extraer                                                    .
   int contador=0;
   for(int i=0; i<4; i++) {
      for(int j=0; j<4; j++) {
         if(A[i][j]<0) {
            W[contador] = A[i][j];
            contador++;
         }
      }
   }
}
```

La Figura 3 muestra la salida en el depurador del ordenador del autor.

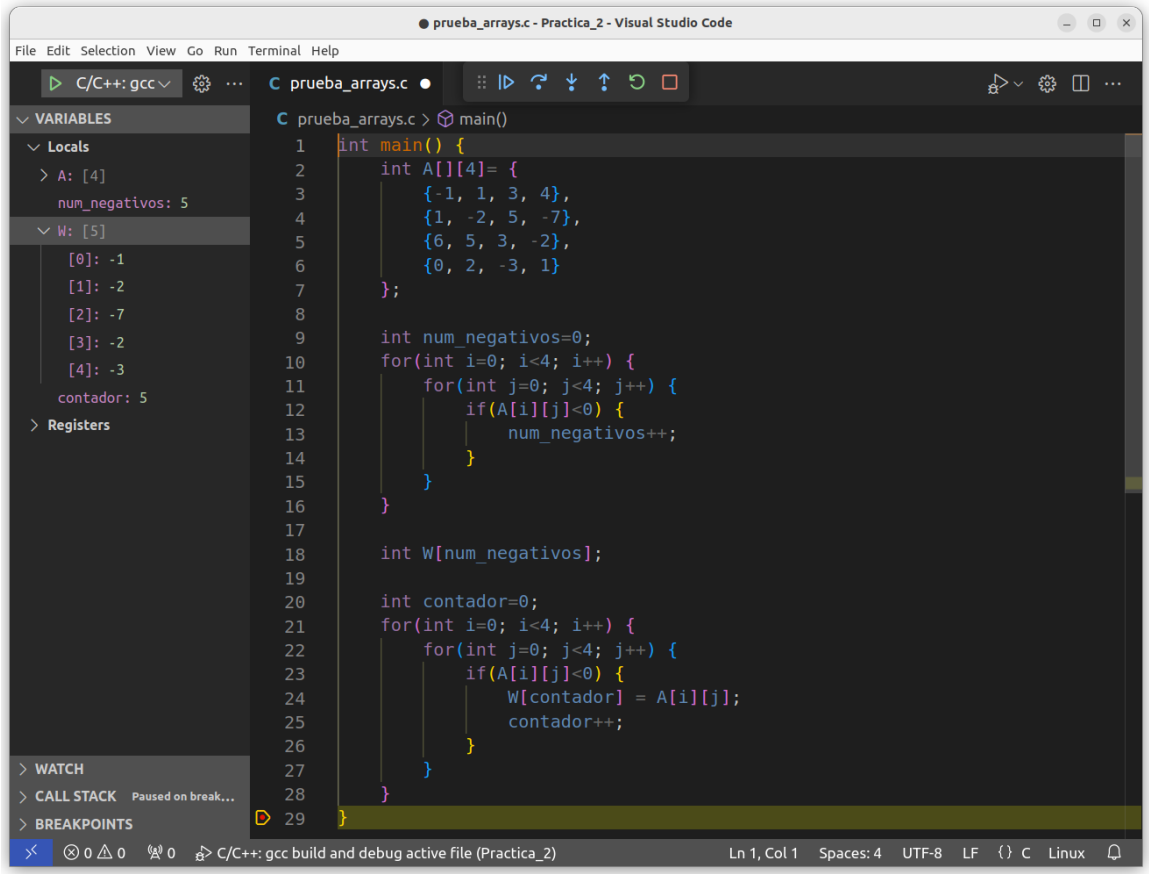

Figura 3: Salida en el depurador del Ejemplo 15

6.8. Algoritmos de la suma y del producto

El problema que se quiere resolver es la suma o el producto de todas las componentes de un *array*. En el caso de la suma, se necesita crear una variable que sirva para ir acumulando las sumas, que inicialmente vale 0. A continuación, se recorren una a una las componentes del *array* y se van sumando a la variable que acumula

las sumas. El Ejemplo 16 suma todas las componentes de un vector.

Ejemplo 16 Algoritmo de la suma aplicado a un vector

```
#include <stdio.h>

int main() {
   double V[] = {17.3, 4.5, -2.1, 3.0, 5.0};

   double suma = 0.0;
   for(int i=0; i<5; i++) {
      suma = suma + V[i];
   }
   printf("Suma= %.2f \n", suma);
}
```

En el algoritmo del producto se pretende calcular el producto de todos los elementos de un *array*. Hay que crear una variable para acumular los productos, que inicialmente vale 1. Esa es la principal diferencia con el algoritmo de la suma, en el que la variable que sirve para acumular las sumas se inicializa en 0. A continuación, se recorren todos los elementos del *array* y se va acumulando el producto. El Ejemplo 17 calcula el producto de todos los elementos de una matriz.

Ejemplo 17 Algoritmo del producto aplicado a una matriz

```
#include <stdio.h>

int main() {
   int A[][4] = { {2, 5, 4, 6}, {1, 3, 4, 2} };

   int prod = 1;
   for(int i=0; i<2; i++) {
      for(int j=0; j<4; j++) {
         prod = prod * A[i][j];
      }
   }
   printf("Producto= %d \n", prod);
}
```

Es frecuente que estos dos algoritmo haya que usarlos en combinación con alguna condición, lo que hace necesario introducir un condicional para solo sumar o multiplicar los elementos que cumplan determinada condición. El Ejemplo 18 suma todas las componentes positivas de un vector.

Ejemplo 18 Suma de las componentes positivas de un vector

```
#include <stdio.h>

int main() {
   int V[] = {2, -5, 4, -6, -1, 3, 4, 2};

   int suma = 0;
   for(int i=0; i<8; i++) {
      if(V[i]>0) {
         suma = suma + V[i];
      }
   }
   printf("Suma= %d \n", suma);
}
```

6.9. Algoritmos del máximo y del mínimo

En este caso se quiere encontrar el valor máximo o el mínimo de las componentes de un *array*. El procedimiento, en el caso del máximo, comienza creando una variable que permita guardar el valor máximo a la que se asigna inicialmente el valor de la primera componente del *array*; a continuación, se recorren una a una todas las componentes del *array* y, en cada caso, se mira si la componente es mayor que el valor máximo guardado, en cuyo caso se sustituirá. En el caso del mínimo se procede de manera similar.

El Ejemplo 19 determina el valor máximo entre las componentes de un vector.

Ejemplo 19 Cálculo del máximo de un vector

```
#include <stdio.h>

int main() {
   int V[] = {2, -5, 4, -6, -1, 3, 4, 2};

   int maximo = V[0];
   for(int i=0; i<8; i++) {
      if(V[i]>maximo) {
         maximo = V[i];
      }
   }

   printf("Máximo= %d \n", maximo); // Imprime 4

   return 0;
}
```

Hay que hacer una observación al código del ejemplo anterior. Podría haberse comenzado el bucle en la segunda componente del vector, pues empezar en la primera es redundante. Es cierto que se haría una iteración menos y, en un sentido purista, el algoritmo sería más eficiente. Pero el ahorro de tiempo de computación en un caso real es insignificante, y la generalización del procedimiento diciendo que se recorren todos los elementos del *array* empezando por el primero justifica mantener así el algoritmo. Piense por ejemplo en el caso de una matriz: ¿cuáles serían los índices de los bucles para empezar en la segunda componente y cuál sería el orden adecuado para recorrer los elementos de la matriz? Por ello, se ha preferido en este texto mantener que, en el bucle, se recorren todos los elementos del *array*.

En el algoritmo del máximo o del mínimo puede ser necesario conocer, no solo el valor máximo de las componentes, sino también qué posición ocupa en el *array*. En ese caso, se necesita alguna variable más que permita ir guardando la posición de la componente correspondiente. El Ejemplo 20 determina cuál es la componente mínima de una matriz, señalando además la fila y columna que ocupa.

Ejemplo 20 Cálculo del valor y posición del mínimo de una matriz

```
#include <stdio.h>

int main() {
   int A[][4] = { {2, -5, 4, -6}, {-1, 3, 4, 2} };

   int minimo = A[0][0];
   int fila = 0;
   int columna = 0;
```

```
   for(int i=0; i<2; i++) {
      for(int j=0; j<4; j++) {
         if(A[i][j]<minimo) {
            minimo = A[i][j];
            fila = i;
            columna = j;
          }
       }
   }

   printf("Mínimo= %d Fila= %d Columna= %d\n",
      minimo, fila, columna);

   return 0;
}                                                                    .
```

Otro detalle que conviene observar es que, si existen en el *array* más de una componente con el valor máximo, como sucede en este caso, solo se encontrará la primera. Se deja como ejercicio al lector la codificación de un algoritmo que permita determinar todas las posiciones en las que se produce el valor máximo o mínimo.

6.10. Solución en los casos dinámicos

En todos los ejemplos que se han visto en este apartado, el *array* que había que procesar era conocido en el momento de la compilación. Lo cierto es que, en la mayoría de los casos reales, el *array* y su tamaño no son conocidos en el momento de la compilación, sino que se conocen de manera dinámica, durante la ejecución del programa.

En estos casos, la solución a los algoritmos que se han visto puede variar un poco o al menos necesitar de una fase previa encargada de la adquisición y declaración del *array*.

El problema de conocer el *array* en tiempo de ejecución es similar en cierto modo al Ejemplo 15 en el que se pretendía extraer de un *array* los elementos que cumplían determinada condición. Allí se resolvió haciendo dos pasadas al *array* original: una primera para contar cuántos elementos cumplían la condición y, con ello, poder crear el *array* destinado a albergar los elementos y una segunda pasada en la que se extraían los elementos.

El *array* que se necesita procesar puede provenir de distintas fuentes. Por ejemplo, podría ser un *array* de números que se le solicitan al usuario; también podría suceder que las componentes del *array* estuvieran guardadas en el disco en un fichero de datos; otro caso habitual es cuando las componentes del *array* entran de manera continua a través de una conexión, por ejemplo porque provienen de las lecturas que proporciona un sensor.

Básicamente hay dos maneras de resolver el problema:

- Procesar los datos en varias pasadas, las primeras para adquirir y crear el *array* y las siguientes para procesar el *array* ya creado.
- Ir procesando los datos a medida que se adquieren, sin necesidad de crear el *array*.

En cada caso concreto, será más útil o más eficiente utilizar una u otra forma de resolver el problema. Se va a resolver un ejemplo concreto utilizando las dos técnicas mencionadas, de forma que el lector pueda comprender su funcionamiento y aplicar, en cada caso que le surja, la que le parezca más adecuada.

El problema que se va a resolver es el del cálculo de la media de una serie de datos que se solicitan al usuario. El usuario irá tecleando números enteros positivos; cuando el usuario teclee un negativo, se dará por concluida la serie de datos y se calcularán el número de datos introducidos y la media de los valores. El número negativo que marca el final de la serie de datos no se contará ni como número de la serie ni para el cálculo de la media.

En primer lugar se va a resolver procesando los datos a medida que entran, sin crear ningún *array*. Se necesita una variable que permita ir acumulando la suma de los números tecleados por el usuario (algoritmo de la suma) y un contador que permita saber cuántos números se han sumado. La media será el cociente entre la suma de todos los números y el número de ellos.

El proceso se hará dentro de un bucle que se ejecutará mientras el usuario no teclee un número negativo. Se ha preferido resolver con un bucle infinito, dentro del cuál se comprueba si el número tecleado es negativo, en cuyo caso una instrucción *break* forzará la salida del bucle. El código se puede ver en el Ejemplo 21.

Ejemplo 21 Cálculo directo de la media de un *array* dinámico

```
#include <stdio.h>

int main() {
   int num;
   int contador = 0;
   double suma = 0.0;
   double media = 0.0;

   while(1) {
      printf("Teclee entero positivo (negativo para finalizar): ");
      scanf("%d", &num);
      if(num<0) {
         break;
      }
      contador++;
      suma = suma + num;
   }

   if(contador>0) {
      media = suma / contador;
   }

   printf("Contador: %d Suma: %.0f Media: %.2f \n",
      contador, suma, media);

   return 0;
}
```

Observe que, a la salida del bucle, solo se calcula la media si el contador es mayor que 0. Se trata de evitar la división por 0 que se produciría si el primer número que teclea el usuario es negativo.

En la segunda solución se va a crear un *array*, inicialmente relleno con ceros y con la suficiente capacidad para albergar todos los números que pudiera teclear el usuario. Para ello se ha declarado una constante DIM de valor 1000 mediante una cláusula `#define`. Si se considerara necesario, se podría ampliar la capacidad máxima del *array* de entrada modificando el valor de la constante. El código es el del Ejemplo 22.

Cuando se han leído todos los números tecleados por el usuario, se crea un *array* para albergar tantos números como indique el contador y, utilizando un bucle, se copian al nuevo *array* las componentes válidas del *array* que almacena las entradas. Una vez obtenido este *array*, se procesa con un algoritmo de la suma, tal como se explicó en el Ejemplo 16.

Ejemplo 22 Cálculo de la media, creando previamente un *array*

```
#include <stdio.h>

#define DIM 1000

int main() {
   int num;
   int datos[DIM] = {0};
   int contador = 0;

   while(1) {
      printf("Teclee entero positivo (negativo para finalizar): ");
      scanf("%d", &num);
      if(num<0) {
         break;
      }
      datos[contador] = num;
      contador++;
   }
   if(contador==0) {
      printf("No se teclearon números válidos\n");
      return 0;
   }

   // Crear y rellenar el array con los datos válidos                    .
   int datos_validos[contador];
   for(int i=0; i<contador; i++) {
      datos_validos[i] = datos[i];
   }

   // Algoritmo de la suma para calcular la media
   double suma = 0.0;
   for(int i=0; i<contador; i++) {
      suma = suma + datos_validos[i];
   }
   double media = suma / contador;

   printf("Contador: %d Suma: %.0f Media: %.2f \n",
      contador, suma, media);

   return 0;
}
```

En este caso concreto, esta solución puede parecer más compleja, pero hay situaciones en las que la creación del *array* con los datos de entrada puede ser conveniente o incluso necesario. Suponga, por ejemplo, que se quieren guardar las entradas o, incluso, que las entradas se producen en diferentes momentos de tiempo y cada vez que hay nuevas entradas hay que actualizar el *array* y los cálculos. Por ello, es conveniente conocer las dos técnicas y elegir que se adapte mejor al problema que se quiere resolver.

7 Estructuras

Una estructura es un tipo de datos compuesto que agrupa, bajo un mismo nombre, varias variables que pueden ser del mismo tipo o de tipos diferentes. En las estructuras, a cada componente se le suele denominar un *campo* de la estructura. Cada campo de la estructura tiene su propio tipo de datos, que puede ser un tipo básico (entero, double, char,...), o un tipo compuesto, por ejemplo un *array* u otra estructura. La sintaxis para definir una estructura es la siguiente:

```
struct nombre {
   tipo_1 nombre_1;
   tipo_2 nombre_1;
   ...;
   tipo_n nombre_n;
};
```

El tipo consta de dos palabras: *struct* y nombre. La lista de las componentes se encierran entre llaves { } y se finaliza en punto y coma. Cada componente individual (cada campo), declara su tipo de datos, su nombre y acaba con punto y coma.

El siguiente ejemplo declara una estructura que se utiliza para representar puntos en el plano, definidos por sus coordenadas x e y:

```
struct Punto2D {
   double x;
   double y;
};
```

La declaración de una estructura lo que hace es definir un nuevo tipo de datos cuyo nombre es el que se haya dado a la estructura. Para poder utilizar el nuevo tipo de datos hay que declarar variables de dicho tipo. Para declarar una variable del tipo de una estructura que haya sido declarada previamente, se usa la palabra clave *struct*, seguida del nombre del tipo de datos y el nombre de la variable que se quiere crear. El siguiente ejemplo declara una variable llamada $p1$ del tipo de datos estructura $Punto2D$ del ejemplo anterior:

```
struct Punto2D p1;
```

Para asignar valores a una variable del tipo *struct* se pueden utilizar distintos procedimientos. Se puede hacer directamente en la declaración de la variable, de manera similar a lo visto para los *arrays*, poniendo entre llaves la lista ordenada de los valores que se quieren asignar a los campos de la estructura. Los valores entre llaves deben respetar el orden que se haya utilizado para los campos en la definición de la estructura.

El código siguiente crea una variable del tipo $Punto2D$ y asigna unos valores iniciales a los campos x e y:

```
struct Punto2D p2 = {10.0, -5.0};
```

También es posible inicializar a 0 todos los campos en la declaración de la variable, poniendo un 0 entre llaves:

```
struct Punto2D p3 = {0};
```

Para acceder a las variables individuales de los campos de una variable-estructura se utiliza la notación *punto*, poniendo el nombre de la variable, un punto y el nombre del campo correspondiente. Por ejemplo, en la variable $p3$ utilizada en el ejemplo anterior, se puede acceder al campo x poniendo $p3.x$ y al campo y, poniendo $p3.y$.

Consejo:

Es un buen criterio llamar a los tipos de datos creados por el usuario, como por ejemplo las estructuras, utilizando un nombre cuya primera letra sea una mayúscula. En el ejemplo anterior, al tipo de datos se le ha llamado *Punto2D*.

Observación:

Cuando se trabaja con estructuras es habitual referirse con la palabra *estructura* tanto al tipo de datos que se ha creado como a las variables de dicho tipo de datos. Esto puede dar lugar a confusiones en algunas situaciones. Realmente, la *estructura* sería el tipo de datos y, a cada variable concreta que se crea de dicho tipo de datos, se le denomina variable o *instancia* de dicho tipo de datos , *instancia* de dicha estructura o también *objeto* de dicho tipo de datos.
Así, en los ejemplos en los que se está trabajando, *Punto2D* sería una estructura y $p1, p2$ y $p3$ serían variables del tipo *Punto2D* o también, instancias de *Punto2D* u *objetos* del tipo *Punto2D*. Para ser más precisos, el tipo de datos es *struct Punto2D*, incluyendo la palabra clave *struct*.

Los campos de las estructuras son, a todos los efectos, variables del tipo que tengan asignado, pudiendo utilizarse en expresiones combinadas con operadores. También es posible asignarles valores, lo que se puede utilizar para su inicialización o modificación.

El código del Ejemplo 23 crea una estructura llamada *Fecha* con tres campos enteros llamados d, m, a de tipo *int*. A continuación crea una instancia del tipo *Fecha* e inicializa los valores de los campos uno a uno; finalmente muestra en pantalla el objeto *Fecha* creado.

Ejemplo 23 Asignación de valores a los campos de una estructura

```
#include <stdio.h>

int main() {
   struct Fecha {
      int d;
      int m;
      int a;
   };
   struct Fecha diaD;
   diaD.d = 6;
   diaD.m = 6;
   diaD.a = 1944;
   printf("%d/%d/%d\n", diaD.d, diaD.m, diaD.a);
   return 0;
}
```

Otra forma de inicializar una instancia de una estructura es como copia de una instancia ya existente. En el código del Ejemplo 24 se define un tipo de estructura denominado *Complejo* que permite modelar un número complejo, y crea una instancia $z1$ de *Complejo* a la que se asignan unos valores iniciales. A continuación, se crea

una nueva instancia $z2$ como copia de $z1$ y se muestran los campos de ambas variables en pantalla:

Ejemplo 24 Utilización del operador de asignación en estructuras

```
#include <stdio.h>

int main() {
   struct Complejo {
      double real;
      double imag;
   };

   struct Complejo z1 = {1.0, 2.5};

   struct Complejo z2 = z1;

   printf("%lf %lf \n", z1.real, z1.imag);
   printf("%lf %lf \n", z2.real, z2.imag);

}
```

Observación:

Si bien el operador de asignación está definido para copiar una instancia de una estructura en otra, otros operadores como los operadores aritméticos, de comparación y lógicos no están definidos para operar en estructuras. Así, por ejemplo, no se podrá comprobar si dos instancias de un mismo tipo de estructuras son iguales utilizando el operador $==$.

Los campos de una estructura no tienen por qué ser tipos básicos, pueden ser a su vez estructuras, *arrays* u otros tipos compuestos. El código del Ejemplo 25 muestra el caso de una estructura llamada *Segmento* cuyos campos son estructuras *Punto2D* para representar los extremos del segmento; a su vez, la estructura *Punto2D* tiene un único campo que es un *array* con dos números enteros correspondientes a las coordenadas del punto. Observe en la instrucción *printf()* la forma de obtener las coordenadas de cada punto.

Ejemplo 25 Estructura con camposTítulo del cuadro

```
#include <stdio.h>

int main() {
   struct Punto2D {
      int coords[2];
   };
   struct Segmento {
      struct Punto2D p1;
      struct Punto2D p2;
   };
   struct Punto2D P1 = {0, 0};
   struct Punto2D P2 = {5, 0};
   struct Segmento s = {P1, P2};

   printf("P1=(%d, %d) \n", s.p1.coords[0], s.p1.coords[1]);
   printf("P2=(%d, %d) \n", s.p2.coords[0], s.p2.coords[1]);

   return 0;
}
```

Truco

Una forma de aprovechar el operador de asignación para crear instancias de determinada estructura es definir una instancia constante, con unos valores por defecto, que es la que se utilizará para inicializar otras instancias, como se hace en el siguiente ejemplo:

Ejemplo 26 Estructura constante para valores por defecto

```
#include <stdio.h>

int main() {
   struct Notas {
      int alumno;
      double algebra;
      double programacion;
   };

   const struct Notas NotaInicial = {0, 5.0, 5.0};

   struct Notas n1 = NotaInicial;
   n1.alumno = 3541;
   n1.algebra = 5.5;
   n1.programacion = 8.0;

   printf("Alumno=%d Álgebra=%.1lf Programación=%.1lf\n",
   n1.alumno, n1.algebra, n1.programacion);

   return 0;
}
```

8 Definición de tipos de datos personalizados con *typedef*

Es posible definir un *alias* (nombre alternativo) para un tipo de datos existente, de forma que se pueda utilizar dicho alias para crear variables de dicho tipo de datos. La sintaxis es la siguiente:

```
typedef tipo_datos alias;
```

Esta definición de un *alias* para cierto tipo de datos se suele denominar *creación de un tipo de datos personalizado*.

Por ejemplo, se podría crear el alias *ENTERO* para referirse a las variables del tipo *int*, como se hace en el siguiente ejemplo:

```
#include <stdio.h>

int main() {
   typedef int ENTERO;
   ENTERO n = 0;
   printf("%d \n", n);
}
```

Observe una vez más, que una vez definido el alias, se puede utilizar para declarar variables de ese tipo de datos.

Es posible definir como tipo personalizado un *array* unidimensional de cierto número de componentes, siguiendo la siguiente sintaxis:

```
typedef tipo_datos alias[num_componentes];
```

Por ejemplo, si se quisiera crear un tipo de datos personalizado para guardar las coordenadas de puntos del espacio de 3 dimensiones, se podría definir a partir de los *arrays* de 3 componentes de tipo *double*, como se hace en el siguiente ejemplo:

```
typedef double Punto3D[3];
Punto3D p1 = {1.5, 0.0, 3.4};
```

También se podría aplicar a *arrays* de más dimensiones, por ejemplo, para crear un tipo personalizado para las matrices 2×2:

```
typedef double Matriz2X2[2][2];
Matriz2X2 A = {{1.0, 0.0},{0.0, 1.0}};
```

La utilización de esta técnica puede simplificar el trabajo con estructuras. Si se asigna un alias al tipo de datos de una estructura existente, la creación de variables se podrá hacer utilizando el alias, sin necesidad de incluir la palabra clave *struct* en la declaración de nuevas variables. Así se ha hecho en el Ejemplo 27, declarando el alias *Punto* para la estructura *P2D* y posteriormente declarando una variable del tipo *Punto*.

Ejemplo 27 Utilización de *typedef* con estructuras

```
#include <stdio.h>

int main() {
   struct P2D {
      int x;
      int y;
   };

   typedef struct P2D Punto;

   Punto p1 = {3, 2}; // Sin typedef habría que usar: struct P2D p1={3, 2};

   printf("%d %d \n", p1.x, p1.y);
}
```

La instrucción *typedef* se puede usar directamente a la vez que se declara la estructura. En el Ejemplo 28, que es equivalente al Ejemplo 27, se puede ver que el código se simplifica si se utiliza esta técnica.

Ejemplo 28 Utilización de *typedef* en la declaración de la estructura

```
#include <stdio.h>

int main() {
   typedef struct P2D {
      int x;
      int y;
   } Punto;

   Punto p1 = {3, 2};

   printf("%d %d \n", p1.x, p1.y);
}
```

El nombre que se utilice para el alias no tiene por qué ser diferente que el nombre de la propia estructura, se puede usar el mismo nombre, como se hace en el siguiente ejemplo:

```
typedef struct Punto {
   double x;
   double y;
} Punto;

Punto P = {3.0, 2.0};
```

De hecho, cuando se utiliza la instrucción *typedef* para crear una estructura, no es necesario utilizar un nombre tras la palabra clave *struc*, se puede utilizar únicamente el nombre final de la instrucción, con lo que el ejemplo anterior se podría escribir de la siguiente forma:

```
typedef struct {
   double x;
   double y;
} Punto;

Punto P = {3.0, 2.0};
```