Tecnología de la Programación
Introducción a C

2020

Juan Antonio Sánchez Laguna
Grado en Ingeniería Informática
Facultad de Informática
Universidad de Murcia

TABLA DE CONTENIDOS

El lenguaje de programación C

El lenguaje de programación C, creado en 1972 por Dennis M. Ritchie, es uno de los más extendidos y utilizados a nivel mundial. Es un lenguaje de programación de propósito general, pero muy básico y portable, pues fue originalmente diseñado para desarrollar el Sistema Operativo.

Su sintaxis y semántica actual es la especificada en el estándar ISO C11, aunque los cambios con respecto a la especificación anterior, la ISO C99, son irrelevantes al nivel al que se pretende aprender a manejarlo en este curso.

Para crear una aplicación en C, primero se escribe el código fuente en uno o varios ficheros con extensión `.c`. Normalmente, alguno de ellos se llama `main.c`. Después, se compilan los ficheros obteniendo un fichero objeto por cada fichero fuente. Y finalmente, los ficheros objeto se enlaza entre sí, y con las bibliotecas del sistema, para crear la aplicación. Por último, ésta se ejecuta desde el Sistema Operativo.

Estructura de un programa en C

Un programa en C es un conjunto de declaraciones y definiciones en el que, como mínimo, debe aparecer la definición de una función sin parámetros llamada `main` que devuelva un valor entero (`int`). Esta función se denomina función principal y será la primera que se ejecute cuando el Sistema Operativo inicie la aplicación. La función principal suele estar definida en el fichero `main.c`.

Así pues, el código fuente del programa más pequeño que se puede escribir en C es el consistente en un fichero `main.c` con la siguiente definición de la función `main`:

```
int main() {
}
```

Fig. 1. El programa más pequeño que se puede escribir en C

El código fuente anterior se puede compilar y enlazar generando un fichero ejecutable directamente por el Sistema operativo. Existen muchos compiladores y Sistemas Operativos diferentes. El compilador gcc de GNU es uno de los más extendidos, y el comando utilizado para generar una aplicación a partir de un fichero `main.c` con el código anterior es el siguiente.

```
gcc –std=c99 –o test main.c
```

Fig. 2. Compilación de un programa escrito en C usando el compilador gcc de GNU

El primer argumento indica que el código fuente se interprete usando las reglas definidas en el estándar ISO C99. El segundo, que se genere un fichero ejecutable llamado `test`. Y finalmente, que el código fuente se encuentra en el fichero llamado `main.c`. Cuando este comando termina, es posible ejecutar el programa `test`. Cuando se ejecuta, evidentemente, no hace nada.

```
>./test
>
```

Fig. 3. Resultado de la ejecución del programa más pequeño que se puede escribir en C

Tipos de Datos y Variables

En C existen múltiples Tipos de Datos para representar números enteros y reales. Cada uno es capaz de representar un rango de valores distinto, con signo o sin signo, o con más o menos precisión en el caso de reales.

Sin embargo, los más comúnmente utilizados son sólo tres: enteros, reales y caracteres, cuyos nombres son, respectivamente, `int`, `double` y `char`.

Para declarar una variable en C se escribe el nombre del Tipo de Datos, después el de la variable y la declaración se acaba con un punto y coma. Es posible declarar múltiples variables del mismo Tipo de Datos separando sus nombres con comas, pero suele quedar más claro si cada una se declara en una línea distinta.

Los nombres de las variables, como los de las funciones y Tipos de Datos, deben ser únicos, y C distingue entre mayúsculas y minúsculas. Por tanto, la variable `n` no es la misma que la variable `N`, y la función principal se llama `main` no `Main`. Además, no se pueden usar tildes ni la ñ en los nombres de variables, tipos ni funciones.

```
int main() {
  int n;
  char letra;
  double altura;

  n = 5;
  letra = 'z';
  altura = 1.9;
}
```

Fig. 4. Ejemplo de declaración de variables en C

Como puede verse en el ejemplo anterior, una vez declaradas, es posible modificar el valor de las variables usando la sentencia de asignación. En Pascal se usan los símbolos `:=` para asignar un valor, pero en C se usa sólo el símbolo `=`. Es recomendable leer la sentencia de asignación como "toma el valor" para distinguirlo de las ocasiones en las que se comparan dos valores y se dice que uno es "igual" que el otro.

Una variable declarada y no inicializada puede tener cualquier valor porque C no inicializa automáticamente ninguna variable. Pero si se desea, es posible dar un valor inicial al mismo tiempo que se hace la declaración.

```
int main() {
  int n = 0;
  char letra = 'a';
  double altura = 0;

  n = 5;
  letra = 'z';
  altura = 1.9;
}
```

Fig. 5. Ejemplo de declaración e inicialización simultánea de variables en C

Como se puede ver, los valores literales de tipo carácter se expresan encerrados entre comillas simples. Los valores reales se pueden escribir usando el punto decimal, pero también es posible usar la notación científica. Por ejemplo, el valor 1.9 se puede expresar como 0.19E1.

Ámbito o visibilidad de las variables

El ámbito o visibilidad de un identificador de variable es la zona del código en la que se puede usar dicho identificador para referirse a la variable en cuestión.

Las variables declaradas dentro de un bloque de código sólo son visibles desde el punto en el que son declaradas hasta el final del bloque. Y aunque es posible declarar variables en cualquier parte de un bloque, es recomendable agruparlas al principio del mismo.

Los bloques de código en C son las secuencias de código delimitadas por los símbolos { y }. Cada función tiene asociado un bloque de código, pero dentro de un bloque se pueden definir otros, y cada uno delimita un ámbito distinto.

Las variables declaradas dentro del bloque de código asociado a una función se denominan locales a esa función y sólo son visibles en ella. Las variables declaradas fuera de cualquier función se denominan globales y son visibles en todas las funciones definidas a partir del punto del fichero donde estén declaradas.

Las variables locales residen en la Pila del Sistema, mientras que las variables globales están en la parte de memoria del programa denominada zona de datos. Como regla general, se debe evitar el uso de variables globales, pues suelen ser el resultado de un mal diseño de la aplicación.

Expresiones aritméticas

Como en cualquier otro lenguaje, es posible escribir expresiones aritméticas. En ellas se pueden usar valores literales, variables, paréntesis para hacer agrupaciones y los operadores aritméticos: `+`, `-`, `/`, `*` y `%`. El símbolo del tanto por ciento es el operador que calcula el módulo, es decir, el resto de la división entera entre dos números.

```
int main() {
  int medio;
  double tercio;
  int cuarto;
  double segundos;

  medio = 1 / 2;
  tercio = 1 / 3;
  cuarto = 1.0 / 4.0;
  segundos = 2000*365*24*60*60;
}
```

Fig. 6. Ejemplo de comportamientos automáticos de C al resolver expresiones aritméticas

Sin embargo, en C las operaciones se realizan según los Tipos de Datos de los operandos. Esto significa, por ejemplo, que en una división entre operandos enteros el resultado será entero, y si hay alguno real el resultado será real. Por tanto, las variables `medio` y `tercio` terminan valiendo cero, aunque la segunda sea de tipo real. La variable `cuarto` también termina valiendo cero. Aunque en este caso se trata de una división real, cuyo resultado es `0.25`, C hace un redondeo automático para almacenar dicho valor en una variable que es de tipo entero.

Por último, la variable `segundos`, que debería terminar guardando el número de segundos que hay en 2000 años, termina con un número negativo. La razón es que C no avisa de desbordamientos. Si se produce un resultado fuera del rango, no se interrumpe el programa ni se señaliza el desbordamiento, sino que simplemente el resultado es erróneo. Todos los operandos de la multiplicación son enteros, por tanto, los cálculos intermedios se hacen con enteros, y terminan desbordándose.

Forzado de tipo (casting)

Para obtener un valor real en una operación con valores literales basta con que alguno de los operandos se exprese como un valor literal de tipo real. Es decir, escribir `1.0/2.0` en lugar de `1/2`. Pero cuando los operandos son variables de tipo entero no es posible añadirles decimales para que se consideren reales.

Sin embargo, hay una forma de hacerlo denominada "casting". Consiste en escribir entre paréntesis el nombre del Tipo de Datos al que se quiere forzar la conversión del operando junto al que se sitúa. En el ejemplo siguiente, al forzar a que la variable `unidad` se interprete como un valor real, la división se hace con reales.

```
int main() {
  int unidad = 1;
  int dos = 2;
  double mitad;
  mitad = (double)unidad / dos;
}
```

Fig. 7. Ejemplo de casting en C

El preprocesador

El preprocesador es un programa usado por el compilador en la primera fase de la compilación. Transforma el código fuente a medida que interpreta las directivas que encuentra en el programa. Las directivas son comandos que comienzan por el símbolo `#`. Las dos directivas más utilizadas son: `#include` y `#define`.

Hay dos formas de escribir la directiva `#include`:

```
#include <nombre de fichero>
#include "nombre de fichero"
```

Ambas se interpretan sustituyendo la línea donde aparecen por el contenido del fichero indicado mediante su nombre, que puede estar expresado mediante una ruta absoluta o una relativa.

Si el nombre del fichero se escribe encerrado entre los símbolos < >, el preprocesador lo busca en una de las carpetas del Sistema Operativo donde haya sido instalado el compilador. Si el nombre del fichero se escribe encerrado entre comillas dobles, se busca en la misma carpeta donde está el fichero que contiene la directiva.

La directiva `#define` sirve para hacer sustituciones en el código fuente y se puede utilizar para definir constantes. Su sintaxis es la siguiente:

```
#define nombre sustituto
```

Mientras que las variables suelen escribirse en minúscula, para las constantes se usan mayúsculas. En el siguiente ejemplo se define y usa la constante PI.

```
#define PI 3.141592

int main() {
  double radio = 3;
  double area = PI * radio * radio;
}
```

Fig. 8. Ejemplo de uso de #define en C

La biblioteca estándar

Habitualmente es necesario mostrar por la pantalla el resultado del programa. Pero C es un leguaje muy simple, y a diferencia de Pascal, no incluye ninguna instrucción como `writeln`. Pero con el compilador siempre se incluye una colección de módulos con funciones básicas, entre las que se encuentran las de entrada/salida. Esta colección se denomina biblioteca estándar de C.

Para poder usar una función en C ésta tiene que estar previamente declarada en el código fuente. Las declaraciones de las funciones de la biblioteca estándar se encuentran en los ficheros de cabecera de sus módulos. Estos ficheros tienen el mismo nombre que el del módulo al que pertenecen pero terminado en `.h`.

Por tanto, para usar cualquier función de la biblioteca estándar hay que incluir el fichero de cabecera apropiado con la directiva `#include` del preprocesador.

Entre las funciones del módulo `stdio` se encuentra `printf` que sirve para mostrar un texto, también llamado cadena de caracteres literal. Las cadenas de caracteres literales se indican en C encerrando los caracteres que las forman entre comillas dobles. Así pues, para mostrar el mensaje "Hola mundo" se hace lo siguiente.

```
#include <stdio.h>

int main() {
  printf( "\tHola mundo\n" );
}
```

Fig. 9. Programa "Hola mundo" en C

Justo antes del texto a mostrar, pero dentro de la propia cadena, aparece el carácter de escape `'\t'` . Los caracteres de escape son símbolos especiales que no pueden ser escritos directamente. Entre los más usados están `'\t'` que representa un tabulador, `'\n'` que representa un salto de línea, y los usados para escribir una barra invertida, una comilla simple y una comilla doble: `'\\'`, `'\''` y `'\"'`.

Pero la función `printf` también permite mostrar valores de variables o resultados de expresiones que se le pasen como argumentos adicionales a la cadena. Para ello se usan los comodines, que son secuencias de caracteres encabezadas por el símbolo `%`. Cada comodín incluido en la cadena es sustituido por el valor del correspondiente argumento adicional, según el orden de aparición.

```
#include <stdio.h>

int main() {
  int s = 365*24*60*60;
  char letra = 'P';
  double pi = 3.141592;
  printf( "Un año tiene %d segundos\n", s );
  printf( "El nombre del número %f empieza por la letra %c\n", pi, letra );
}
```

Fig. 10. Ejemplo de uso de printf

Los comodines permiten indicar el Tipo de Datos del valor a mostrar, e incluso, el formato con el que se quiere que aparezca. En el ejemplo de la Fig. 10 aparecen tres de ellos: `%d` para mostrar valores enteros, `%c` para mostrar un carácter y `%f` para mostrar un número real. El primer comodín se reemplaza por el primer argumento adicional en la llamada, es decir, el siguiente a la cadena. El segundo por el siguiente, y así sucesivamente.

Obviamente, para mostrar el símbolo `'%'` no basta con incluirlo en la cadena, pues se confundiría con el comienzo de un comodín. En este caso hay que escribirlo dos veces seguidas para que aparezca una sola vez por la pantalla.

Además de la función `printf`, el módulo `stdio` contiene muchas otras funciones de entrada/salida útiles como `scanf`, o todas las relativas a manejo de ficheros. La biblioteca estándar incluye otros módulos, entre los que destacan los siguientes:

- `ctype`. Incluye funciones para clasificar caracteres según su tipo, transformar entre mayúscula y minúscula, etc.
- `math`. Incluye funciones para calcular raíces cuadradas, logaritmos, potencias, diferentes tipos de redondeos y funciones trigonométricas.
- `stdlib`. Incluye funciones para gestión de memoria dinámica, generación de números aleatorios, cálculo del valor absoluto, etc.
- `string`. Incluye funciones para realizar operaciones con cadenas de caracteres.
- `time`. Incluye funciones para conocer y manejar la fecha y hora del sistema.

La experiencia permite tener cierta idea de dónde está cada función y cómo se usa. Afortunadamente, existe una amplia documentación de todos estos módulos y funciones. Además, es muy fácil de localizar a través de internet.

Se puede acudir a una web de referencia como la de GNU:

https://www.gnu.org/software/libc/documentation.html

Pero también es posible encontrar ayuda escribiendo directamente en un buscador, como el de Google, el nombre de la función precedido de la palabra **man**, en referencia al manual de la función. Normalmente, suele aparecer entre los primeros tres resultados uno de la web https://linux.die.net que contiene el manual de uso de la función.

Comentarios

A medida que los programas se van haciendo más grandes resulta necesario incluir comentarios que clarifiquen determinadas partes del código. En C hay dos formas de incluir comentarios: comentarios de una línea y comentarios de varias líneas.

Los comentarios de una línea comienzan por dos barras consecutivas (`//`) y terminan al final de la línea donde han empezado. Se usan principalmente para clarificar el uso de variables o separar las diferentes fases de un proceso compuesto por múltiples sentencias.

```
/*
  Este programa sirve para demostrar los dos tipos de comentarios que hay.
  Los hay de varias líneas como éste, pero también de una como el de abajo
*/
int main() {
  double r = 3; // La variable r representa el radio de la circunferencia
  double area = PI * r * r;
}
```

Fig. 11. Ejemplo de uso de comentarios en C

Los comentarios de varias líneas comienzan con la secuencia `/*` y terminan al aparecer la secuencia contraria `*/`. Se pueden usar para documentar las funciones indicando qué hacen, cómo se usan, etc.

Funciones y procedimientos

Además de usar funciones incluidas en la biblioteca estándar, también es posible escribir funciones propias.

Para definir una función en C se indica su nombre, la lista de parámetros separados por comas y encerrados entre paréntesis, y a continuación, el bloque de código de esta entre llaves. Además, antes del nombre se debe indicar el Tipo de Datos de los valores que devuelve la función. Y en cualquier parte del código de la función se puede usar la sentencia `return` para conseguir que la función termine devolviendo un valor concreto. El valor que se quiera devolver se indica a continuación de `return` y antes de terminar la sentencia con punto y coma. Y como la función termina al ejecutar `return`, las sentencias que haya a continuación no se ejecutarán.

```
#include <stdio.h>

int suma( int a, int b ) {
  return a + b;
}

int main() {
  printf("1 + 2 = %d. Y 1 + 2 + 3 = %d\n", suma( 1, 2 ), suma( 1, suma( 2, 3 ) ) );
}
```

Fig. 12. Ejemplo de definición de una función propia

En el ejemplo de la Fig. 12 se muestra la definición de la función `suma`. Esta tiene dos parámetros, ambos de tipo entero, y la propia función devuelve un valor de tipo entero. En su bloque de código sólo aparece la sentencia `return` terminando la función y devolviendo como resultado la suma de `a` y `b`.

Para usar una función se escribe su nombre, y entre paréntesis, los valores que se quiera usar como argumentos de la llamada. Una llamada, o invocación, a función se puede usar como operando en cualquier expresión que acepte operandos del Tipo de Datos que devuelva la función. En el ejemplo, el segundo argumento de la segunda llamada a `suma` es el resultado de una tercera llamada a `suma`.

Además de funciones, en C también se pueden crear procedimientos, es decir, subprogramas que no devuelven un resultado, sino que tienen algún otro efecto. Los procedimientos se definen en C igual que cualquier otra función, pero como Tipo de Datos devuelto se usa la palabra reservada `void`. En un procedimiento se puede usar `return` para forzar la terminación de este, pero sin indicar ningún valor de retorno, sólo el punto y coma. Además, los procedimientos no pueden ser operandos en expresiones de ningún tipo.

```
#include <stdio.h>

void saluda( ) {
  printf( "Hola!" );
}

int main() {
  saluda();
}
```

Fig. 13. Ejemplo de definición de un procedimiento propio

La lista de parámetros de una función o procedimiento puede ser vacía. Pero, tanto en su declaración, como al llamarla, se deben escribir los dos paréntesis.

Paso de parámetros en C

El paso de parámetros a funciones en C se realiza por valor. Es decir, los valores de los argumentos se copian en los parámetros de la función, que, a todos los efectos, se consideran variables locales a la misma. Así pues, desde el código de la función no se puede modificar el valor de una variable usada como argumento.

```
#include <stdio.h>

void incrementa( int n ) {
  n = n + 1;
}

int main() {
  int n = 0;
  incrementa( n );
  printf( "Tras llamar a incrementa, la variable n vale %d\n", n );
}
```

Fig. 14. Demostración de que el paso de parámetros en C es por valor

La función `incrementa` mostrada en la Fig. 14 tiene un parámetro de tipo entero, y en su código incrementa el valor de dicho parámetro. En la función principal se declara una variable llamada `n`, y se ejecuta `incrementa` usando dicha variable como argumento. Sin embargo, a pesar de que coincidan los nombres, la variable `n` declarada en la función `main` y el parámetro `n` no tienen nada que ver. Ambas son variables locales, pero cada una de una función distinta. Al producirse la llamada, el cero que hay en la variable `n` de la función `main` se copia en el parámetro `n` de la función incrementa. A continuación, dicho parámetro, que se puede considerar como una variable local a `incrementa`, es modificado, pero al terminar la llamada, el parámetro `n` desaparece, y la variable `n` de la función `main` sigue valiendo cero.

Expresiones relacionales y lógicas

Las expresiones relacionales y lógicas se forman usando los operadores siguientes:

- Los operadores relacionales: `<`, `>`, `<=`, `>=`, se usan para comparar valores.
- El operador `==` devuelve verdadero cuando los dos operandos son iguales.
- El operador `!=` devuelve verdadero cuando los dos operandos son distintos.
- El operador `&&` es el equivalente al Y lógico (AND de Pascal).
- El operador `||` (dos barras verticales) es el equivalente al O lógico (OR de Pascal).
- El operador `!` representa la negación (NOT de Pascal).

Sin embargo, C no incluye ningún Tipo de Datos para representar valores booleanos[1]. Las comparaciones y expresiones lógicas en C devuelven un valor entero. Un valor cero indica que la expresión es falsa, y un valor distinto de cero indica que es cierta. Por tanto, el resultado de una expresión lógica puede asignarse a una variable entera, y una variable o expresión entera puede utilizarse como expresión lógica en cualquier sentencia condicional.

[1] En realidad, la biblioteca estándar de C sí incluye el módulo `stdbool` en el que se define el tipo `bool` y las constantes `true` y `false`. Pero `bool` es un alias de `int`, y `true` y `false` representan los valores enteros 1 y 0.

Sentencia condicional

C incluye la sentencia condicional `if` que permite ejecutar una parte de código si se cumple cierta condición. Y también se puede usar la construcción `if-else` para poder expresar qué hacer en caso de que no se cumpla la condición. En cualquier caso, la condición debe ser una expresión lógica de tipo entero y se debe escribir entre paréntesis.

```
#include <stdio.h>

int bisiesto( int year ) {
  if ( ( year  % 4 == 0 && year % 100 != 0 ) || year % 400 == 0 ) {
    return 1;
  } else {
    return 0;
  }
}

int main() {
  int y = 2017;
  if ( bisiesto( y ) ) {
    printf( "%d si es bisiesto\n", y );
  } else {
    printf( "%d no es bisiesto\n", y );
  }
}
```

Fig. 15. Ejemplo de uso de operadores lógicos y la sentencia if-else

En el ejemplo de la Fig. 15 aparece la función `bisiesto` que, dado un año, determina si es bisiesto comprobando si es divisible por cuatro pero no por cien, o si lo es por cuatrocientos. Los paréntesis evitan que la expresión sea ambigua. En la función principal se usa otra sentencia `if` para mostrar un mensaje u otro en función del resultado de la llamada a `bisiesto`. Nótese que no es necesario comparar el resultado de la llamada con 1 ya que, de serlo, C considerará cierta la expresión.

Del mismo modo, es habitual devolver directamente el resultado de una expresión booleana en lugar de usarlo para devolver uno o cero con una sentencia `if`.

```
int iguales( int a, int b ) {
  return a == b;
}
```

Fig. 16. Ejemplo de construcción típica en la que se devuelve el resultado de una expresión

Finalmente, es importante recordar que `==` es el símbolo usado para comprobar la igualdad. Y el símbolo `=` es el usado para la sentencia de asignación. Confundirlos en una expresión booleana es uno de los errores más habituales y peligrosos, pues en C la sentencia de asignación también se considera una expresión. Por tanto, el código de la Fig. 17 compila, pero no funciona como se esperaría. Lo que devuelve la función es el valor de `b`, después de asignárselo al parámetro `a`.

```
int iguales( int a, int b ) {
  return  a = b;
}
```

Fig. 17. Ejemplo de error típico al hacer una comparación usando el símbolo de la asignación

Sentencia switch

Además de la sentencia `if`, C cuenta con la sentencia `switch` que permite expresar, de una forma compacta, una cadena de comprobaciones sobre un mismo valor realizadas enlazando sentencias `if-else`.

La sentencia `switch` evalúa una expresión de tipo entero o carácter y compara el resultado con las etiquetas definidas dentro de su cuerpo. Si se encuentra una coincidencia el flujo de programa salta a la instrucción que tenga asociada la etiqueta.

```
switch ( expresión ) {
  case ETIQUETA1:
    sentencias;
  case ETIQUETA2:
    sentencias;
  ...
  case ETIQUETAN:
    sentencias;
  default:
    sentencias;
}
```

Fig. 18. Esquema de uso de una sentencia switch

La expresión que debe evaluar la sentencia `switch` se escribe entre paréntesis, y a continuación, se abre un bloque en donde aparecen las etiquetas y el código asociado. Las etiquetas se declaran con la palabra reservada `case` seguida de un valor literal de tipo entero o carácter y del símbolo de los dos puntos.

Habitualmente, se usa la palabra reservada `break` al final del conjunto de sentencias asociado a cada etiqueta. Al llegar al `break`, el flujo del programa salta al final del `switch`. Esto permite que sólo se ejecute el código asociado a la etiqueta cuyo valor coincidió con el de la expresión. Pero también es posible agrupar varias etiquetas cuando se quiere compartir el código entre ellas.

Si aparece la etiqueta `default`, y el valor de la expresión no coincide con ninguna otra etiqueta, el flujo de programa saltará al código asociado a `default`.

```
void ejecuta( int comando ) {
  switch ( comando ) {
    case 1:
      printf( "El robot avanza\n" );
      break;
    case 2:
      printf( "El robot gira a la derecha\n" );
      break;
    case 3:
      printf( "El robot retrocede\n" );
      break;
    case 4:
      printf( "El robot gira a la izquierda\n" );
      break;
    default:
      printf( "El robot se detiene\n" );
  }
}
```

Fig. 19. Ejemplo de uso de una sentencia switch

Sentencias de iteración

C incluye tres sentencias para realizar iteraciones: `while`, `do-while` y `for`. Las tres producen iteraciones que se repiten mientras la condición especificada sea cierta.

La sentencia `while` primero comprueba la condición, que debe especificarse entre paréntesis, y, si es cierta, ejecuta las instrucciones del cuerpo del bucle antes de volver a comprobar la condición.

```
while ( condición de continuación ) {
  cuerpo;
}
```

Fig. 20. Estructura de un bucle while

La sentencia `do-while` primero ejecuta las instrucciones del cuerpo del bucle, y después, comprueba la condición, que también debe especificarse entre paréntesis. Por lo tanto, siempre se realiza, al menos, una iteración.

```
do {
  cuerpo;
} while ( condición de continuación );
```

Fig. 21. Estructura de un bucle do-while

Finalmente, la sentencia `for` es una versión compacta de un `while`. Habitualmente, se usa para realizar un número conocido de repeticiones.

```
for ( inicialización ; condición de continuación ; avance ) {
  cuerpo;
}
```

Fig. 22. Estructura de un bucle for

La inicialización se hace una vez antes de empezar y puede consistir en una declaración e inicialización de una variable. El ámbito de dicha variable queda reducido al propio bucle, y normalmente, se declara la variable que se usa en el bucle para contar iteraciones.

La condición se comprueba antes de hacer cada iteración y sólo se ejecuta el cuerpo si el resultado es verdadero. Y la sentencia de avance se ejecuta una vez al final de cada iteración, tras ejecutar el cuerpo.

Finalmente, cuando el cuerpo del bucle consiste en una sola sentencia, no es necesario crear un bloque encerrándola. Pero, en general, el código queda más legible si se usa un bloque.

```
int i = 0;
while ( i < 10 ) {
  printf( "%d\n", i );
  i = i + 1;
}
```

```
int i = 0;
do {
  printf( "%d\n", i );
  i = i + 1;
} while ( i < 10 );
```

```
for ( int i = 0 ; i < 10 ; i = i + 1 ) {
  printf( "%d\n", i );
}
```

Fig. 23. Tres bucles idénticos escritos con distintas sentencias de iteración

Tipos de Datos definidos por el programador

En C hay tres formas de crear Tipos de Datos a partir de otros existentes: las estructuras, las uniones y las enumeraciones.

Estructuras

Una estructura es un Tipo de Datos definido mediante la agregación de variables de otros Tipos de Datos, incluidos otras estructuras, uniones y enumeraciones. A las distintas partes que componen una estructura se les llama miembros o campos.

Para definir una estructura se usa la palabra reservada `struct` seguida del nombre de la estructura, y encerrado entre llaves, la lista de los campos que la componen. Cada campo se define indicando su Tipo de Datos y su nombre. Además, tras la llave que cierra la definición de la estructura hay que escribir un punto y coma.

Las estructuras sirven para representar objetos complejos a partir del conjunto de propiedades o atributos que los definen. Por ejemplo, si se desea representar un Punto en el plano mediante un par de coordenadas de tipo entero, se puede crear una estructura con esos dos campos.

```
struct PuntoRep {
  int x;
  int y;
};
```

Fig. 24. Definición de un Tipo de Datos Estructura en C

Una vez definida, se pueden declarar variables del nuevo Tipo de Datos. Para hacerlo hay que escribir la palabra reservada `struct` y el nombre de la estructura antes del nombre de la variable. Es decir, el nombre del tipo de datos incluye la palabra `struct`.

```
struct PuntoRep centro;
```

Fig. 25. Declaración de una variable de tipo estructura en C

Para acceder a los campos de una variable de tipo estructura se usa el operador punto. Si se escribe el nombre de la variable y el del campo deseado unidos por un punto se puede consultar o modificar el valor de dicho campo en dicha variable.

```
struct PuntoRep centro = { .x = 0, .y = 0 };

centro.x = 100;
centro.y = 50;
```

Fig. 26. Declaración y uso de variables de tipo estructura en C

En la Fig. 26 se declara la variable `centro` y luego se modifican sus dos campos. Pero también se muestra una de las formas en la que se pueden inicializar los distintos campos de una estructura en el momento de su declaración. Consiste en escribir una lista de inicializaciones, separadas por comas y encerradas entre llaves, en la que el nombre de cada campo a inicializar va precedido de un punto.

Uniones

Una unión es un Tipo de Datos definido mediante la superposición de dos o más variables de otros Tipos de Datos. Todas las variables unidas comparten el mismo espacio de memoria. Las uniones permiten declarar variables que puedan contener valores de diferentes Tipo de Datos, pero siempre en diferentes momentos.

Para definir una unión se usa la palabra reservada `union` seguida del nombre que se le quiera dar, y encerrado entre llaves, la lista de los campos que la componen. Cada campo se define indicando su Tipo de Datos y su nombre. Además, tras la llave que cierra la definición de la unión hay que escribir un punto y coma.

Las uniones sirven para representar objetos cuyo tipo de datos dependa de la situación. Por ejemplo, para guardar el valor de un operando en una expresión aritmética, unas veces se necesita una variable de tipo entero y otras de tipo real. En lugar de usar dos variables distintas, se puede usar una de tipo unión.

```
union ValorRep {
  int entero;
  double real;
};
```

Fig. 27. Definición de un Tipo de Datos Unión en C

En la Fig. 27 se define un nuevo Tipo de Datos denominado `union ValorRep`, compuesto por la superposición de dos campos distintos. Una vez definido, es posible declarar variables usando su nombre completo, es decir, incluyendo la palabra `union`. Y para darle un valor inicial durante la declaración y para acceder a sus campos se hace igual que con las estructuras.

```
union ValorRep op = { .entero = 0 };

op.entero = 3;
op.real = 5.6;
```

Fig. 28. Declaración y uso de variables de tipo Unión en C

En la Fig. 28 se declara la variable `op` y luego se modifican sus dos campos. Pero, después de la segunda asignación, el campo `entero` se habrá sobrescrito y no valdrá 3, porque en cada momento, sólo uno de los campos puede tener sentido.

Las uniones suelen usarse como campos de otras estructuras en las que algún otro campo indica cómo debe usarse el de tipo unión. Por ejemplo, la Fig. 29 muestra la definición de la estructura `OperandoRep` que agrupa un valor (entero o real) y su tipo codificado con otro entero. También aparece la estructura `OperandoRep` que agrupa dos operandos y un entero que indica el tipo de operación con un código numérico.

```
struct OperandoRep {
  int tipo;                  // 1 -> entero | 2-> real
  uniont ValorRep valor;
}
struct OperacionRep {
  int operador;          // 1 -> suma | 2 -> resta | 3 -> mult | 4 -> div
  struct OperandoRep op1;
  struct OperandoRep op2;
};
struct OperacionRep op;  // op va a guardar la operación 3 + 5.6
```

Fig. 29. Ejemplo de uso combinado de uniones y estructuras en C

Enumeraciones

Una enumeración, o Tipo de Datos Enumerado, es un Tipo de Datos definido mediante un conjunto de nombres.

Para definir una enumeración se usa la palabra reservada `enum` seguida del nombre que se desee, y encerrado entre llaves, la lista de nombres separados por comas. Además, tras la llave que cierra la definición de la enumeración hay que escribir un punto y coma.

Las enumeraciones sirven para representar los diferentes elementos de un conjunto. Por ejemplo, si una variable debe guardar el día de la semana, se puede crear una enumeración que represente los posibles días de una semana y declarar la variable con ese nuevo Tipo de Datos.

```
enum DiaRep { lunes, martes, miercoles, jueves, viernes, sabado, domingo };
```

Fig. 30. Definición de un tipo de datos enumerado en C

Una vez definido el Tipo de Datos enumerado, se pueden declarar variables y a éstas se les puede asignar cualquiera de los nombres de la enumeración.

```
enum DiaRep hoy = lunes;
```

Fig. 31. Declaración y uso de variables de tipo estructura en C

Por desgracia, C permite que las variables de un Tipo de Datos enumerado concreto tomen valores de cualquier otro, o incluso, cualquier valor de tipo entero. Si no lo hiciera serían una gran ayuda para evitar errores, pues permitiría controlar que los valores que toma una variable están dentro de los definidos.

Siguiendo con el ejemplo de las expresiones aritméticas, las estructura que representan un operando y una operación se pueden mejorar usando un Tipo de Datos enumerado para representar el operador.

```
enum TipoRep { entero, real };
enum OperadorRep { suma, resta, multiplicacion, division };
struct OperandoRep {
  enum TipoRep tipo;
  uniont ValorRep valor;
}
struct OperacionRep {
  enum OperadorRep operador;
  struct OperandoRep op1;
  struct OperandoRep op2;
};
int main() {
  struct OperacionRep op;  // op va a guardar la operación 3 + 5.6
  op.operador = suma;
  op.op1.tipo = entero;
  op.op1.entero = 3;
  op.op2.tipo = real;
  op.op2.real = 5.6;
}
```

Fig. 32. Ejemplo de uso combinado de enumeraciones, uniones y estructuras en C

La sentencia typedef

Permite asociar nuevos nombres, o alias, para otros Tipos de Datos. En la Fig. 33 se muestra cómo crear un alias para varios Tipos de Datos. Estos alias se pueden usar para simplificar la escritura de las declaraciones.

```
typedef int Entero;
typedef struct PuntoRep Punto;
typedef enum DiaRep Dia;
typedef union ValorRep Valor;
typedef struct OperacionRep Operacion;

Entero edad = 30;
Punto p = { .x = 0, .y = 0};
Dia hoy = lunes;
Valor v = { .real = 1.5 };
Operacion op = { .operador = resta, .op1.tipo = entero, .op1.entero = 1,
                                    .op2.tipo = real, .op2.real = 2.5 };
```

Fig. 33. Creación de un alias para un Tipo de Datos en C

A cualquier variable de los Tipos de Datos definidos mediante estructuras, uniones y enumeraciones se le puede asignar un valor del mismo Tipo de Datos. La asignación implica una copia campo a campo.

```
Punto centro = { .x = 100, .y = 50 };
Punto origen;
Valor op1 = { .entero = 50};
Valor op2;
Dia salida = lunes;
Dia llegada;

origen = centro;
op2 = op1;
llegada = salida;
```

Fig. 34. Asignación entre variables de tipo estructura, unión y enumeración en C

Pero además de para declarar variables, los nuevos Tipos de Datos se pueden usar para declarar los parámetros y el Tipo de Datos que devuelven las funciones.

```
Dia dia_de_la_semana( int y, int m, int d ) { ... }
Punto medio( Punto a, Punto b ) { ... }
Valor resultado( Valor v1, Operador op, Valor v2 );
```

Fig. 35. Declaración de funciones que devuelven y usan tipos complejos como parámetros

Como pasa con cualquier otro Tipo de Datos, los parámetros cuyo Tipo de Datos es una estructura, una unión o un Tipo de Datos enumerado, reciben una copia del valor usado como argumento al invocar a la función en cuestión. Por lo tanto, tampoco en este caso es posible modificar la variable que se haya usado como argumento.

Del mismo modo, cuando una función devuelve uno de estos Tipos de Datos, lo que se devuelve es una copia del valor usado en la sentencia `return`. Cuando la estructura a devolver tiene muchos campos esta operación puede resultar costosa.

Por último, la terminación `Rep` usada en el nombre de las estructura, uniones y enumeraciones no es casual. Permite distinguir entre el nombre de la estructura que se usa para representar el Tipo de Datos y el nombre del Tipo de Datos que, habitualmente, se creará como un alias con la sentencia `typedef`.

Arrays

Arrays de una dimensión

Un array es una estructura de datos homogénea que permite almacenar uno o más elementos del mismo tipo. La característica principal de los arrays es que todos sus elementos se encuentran almacenados de forma consecutiva y para referirse a ellos se usa la posición que ocupan.

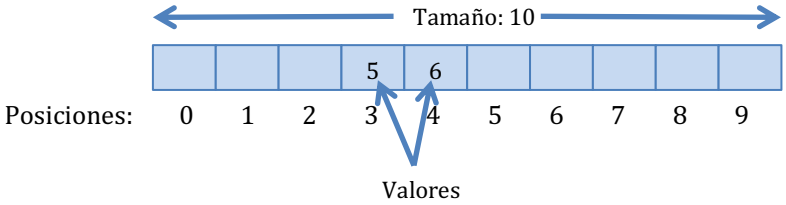

Fig. 36. Esquema gráfico de un array en C

Para declarar un array se escribe el Tipo de Datos de los elementos que contendrá delante del nombre de este, y detrás, una pareja de corchetes con el tamaño del array, es decir, con el número de elementos que debe ser capaz de almacenar. La declaración se acaba con un punto y coma. La Fig. 37 muestra la declaración de un array con capacidad para 10 enteros.

```
int numeros[10];                // Declaración del array
numeros[0] = 1;                 // Modificación del primer elemento
printf( "%d\n", numeros[2] );   // Acceso al tercer elemento
```

Fig. 37. Ejemplo declaración de un array y de acceso a sus elementos

Cada elemento de un array se puede considerar una variable del mismo Tipo de Datos que esté declarado el array. Pero estas variables no tienen nombre, para referirse a ellas se usa el nombre del array que las contiene, y entre corchetes, se especifica la posición que ocupa la variable dentro del array. En C, estas posiciones empiezan en cero y terminan en n-1, siendo n el tamaño del array.

Un array declarado y no inicializado en C puede contener valores indeterminados porque las variables en C no se inicializan automáticamente. Para evitarlo se puede dar un valor inicial a los distintos elementos del array, especificándolos mediante una lista separada por comas y encerrada entre llaves.

```
int numeros[10] = {1,3,5,7};
int datos[100] = {0};
```

Fig. 38. Declaración e inicialización simultánea de un array

Si el número de valores literales es menor que el tamaño del array, se inicializan los primeros elementos del array para los que sí haya valores literales, y el resto se inicializa a cero. Por tanto, es posible inicializar todo un array a cero escribiendo un único cero en la lista de valores.

Cuando se declara un array especificando todos sus elementos no es necesario indicar el tamaño del mismo, pues C lo calcula contando los valores literales que hay en la lista usada para su inicialización.

```
int numeros[] = {1,3,5,7};
```

Fig. 39. Cálculo automático del tamaño de un array a partir de los valores usados para su inicialización

Arrays y funciones

Las funciones en C pueden tener arrays como parámetros. Pero en C el paso de parámetros siempre se hace por valor, es decir, en el parámetro se copia el valor del argumento usado en la invocación. Como en C un array representa la dirección de memoria que ocupa su primer elemento, lo que se copia al parámetro es esa dirección de memoria. Por lo tanto, la función que recibe como parámetro un array puede acceder al contenido del mismo y modificarlo. Además, el paso de parámetros es muy eficiente, pues no depende del tamaño del array.

Para indicar que un parámetro de una función va a ser un array unidimensional se añade una pareja de corchetes al declarar el parámetro. Pero, en este caso, no es necesario indicar ningún tamaño, y si se hace, C lo ignora.

Si dentro en la función se necesita saber el número de elementos del array para acceder a ellos sin salirse de sus límites, se puede añadir un segundo parámetro a la función que permita indicar el tamaño del array al hacer la llamada.

```
#include <stdio.h>

void muestra( int array[], int n ) {
  printf( "[ " );
  for ( int i = 0 ; i < n ; i = i + 1 ) {
    printf( "%d ", array[i] ) ;
  }
  printf( "]\n" );
}

void rellena( int array[], int n, int dato ) {
  for ( int i = 0 ; i < n ; i = i + 1 ) {
    array[i] = dato;
  }
}

int main() {
  int pares[] = {2,4,6};
  int numeros[6];

  muestra( pares, 3 );
  rellena( numeros, 6, 0 );
  muestra( numeros, 6 );
}
```

Fig. 40. Ejemplos de función que reciben como parámetro un array de enteros

En el ejemplo de la Fig. 40 aparecen dos funciones que reciben arrays de enteros como parámetro. La primera, llamada `muestra`, tiene dos parámetros: el array y su tamaño. En su código se realiza un recorrido mostrando todos los valores incluidos en el array. La función `rellena`, demuestra que desde el código de una función se puede modificar el contenido de un array recibido como parámetro. Esta función rellena el array que se le pasa como primer argumento, con el valor recibido en el tercer parámetro.

Sin embargo, las funciones en C no pueden devolver arrays. Al menos, no como los que se ha estudiado hasta el momento, es decir, arrays declarados en la zona de memoria destinada a los datos o en la Pila del sistema. Cuando se estudien los punteros se verá una forma de escribir funciones que devuelvan arrays alojados memoria dinámica.

Arrays multidimensionales

Para declarar un array de dos o más dimensiones se hace igual que para los unidimensionales, pero añadiendo los tamaños de las siguientes dimensiones encerrados en sus correspondientes parejas de corchetes.

```
int matriz[3][4];  // Matriz de 3 filas y 4 columnas
```

Fig. 41. Declaración de un array de dos dimensiones

Las dimensiones del array se leen de izquierda a derecha. En el ejemplo anterior, el array `matriz` tiene dos dimensiones. La primera de tamaño 3 y la segunda de tamaño 4. Es un array con capacidad para tres arrays, cada uno de cuatro enteros. Por tanto, los datos quedarán organizados en memoria como muestra la Fig. 42: primero los cuatro enteros del primer subarray, a continuación, los cuatro enteros del segundo subarray, y finalmente, los cuatro enteros del tercer subarray.

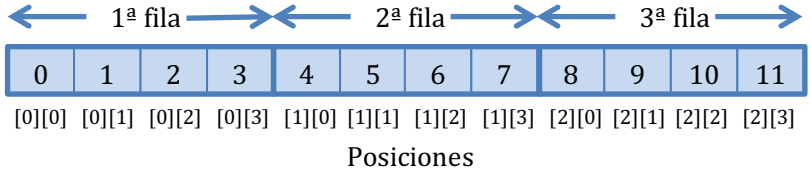

Fig. 42. Esquema gráfico de un array bidimensional de tres filas y cuatro columnas

Para inicializar un array multidimensional se escriben listas de listas de valores, usando llaves de forma anidada como sigue:

```
int matrizA[2][3] = { {1,2,3}, {4,5,6} };
int matrizB[3][2] = { {1,2}, {3,4}, {5,6} };
```

Fig. 43. Declaración e inicialización de arrays bidimensionales

Llamar fila o columna a los subarrays es una decisión arbitraria, sin importancia desde el punto de vista de C, aunque suele llamarse filas a la primera dimensión y columnas a la segunda.

También es posible declarar e inicializar un array de dos o más dimensiones sin indicar el tamaño de la primera de ellas:

```
int matriz[][3] = { {1,2,3}, {4,5,6} };
```

Fig. 44. Cálculo automático del tamaño de la primera dimensión de un array multidimensional

En este caso, C determina automáticamente que el tamaño de la primera dimensión debe ser dos.

El acceso a los distintos elementos se hace indexando cada dimensión entre corchetes, y siempre usando 0 para referirse a la posición del primer elemento.

```
int matriz[3][4];  // Matriz de tres filas y cuatro columnas
matriz[0][0] = 1;  // Primera fila, primera columna
matriz[0][3] = 2;  // Primera fila, última columna
matriz[2][0] = 3;  // Última fila, primera columna
matriz[2][3] = 4;  // Última fila, última columna
```

Fig. 45. Ejemplo de acceso a un array bidimensional

Arrays de tamaño variable automático

Para definir una función con un parámetro que sea un array multidimensional hay que especificar el tamaño de todas las dimensiones, excepto el de la primera. Esa primera dimensión indica el tamaño del array, mientras que las demás indican el tamaño de los elementos de la primera dimensión. Como en el parámetro se copia la dirección de memoria del primer elemento, no importa cuántos haya en el array. Si se sabe lo que ocupa cada uno de ellos, y la dirección de memoria del primero, es posible calcular la posición del resto en función de la posición que se indique.

```
void resetea( int array[][3], int filas, int columnas ) {
  for ( int i = 0 ; i < filas ; i = i + 1 ) {
    for ( int j = 0 ; j < columnas ; j = j + 1 ) {
      array[i][j] = 0;
    }
  }
}
```

Fig. 46. Ejemplo de función que recibe como parámetro un array bidimensional

En el ejemplo de la Fig. 46, el parámetro `columnas` se podría ignorar, puesto que esta función sólo admite arrays de dos dimensiones cuya segunda dimensión tenga tamaño 3. Por lo tanto, esta forma de declarar parámetros que sean arrays de varias dimensiones es muy poco flexible.

Afortunadamente, desde la publicación del estándar C99, es posible usar arrays de tamaño variable automático. Los arrays de tamaño variable automático son los que se declaran especificando su tamaño con una variable local en lugar de con un valor entero literal.

Por lo tanto, es posible escribir una función con un parámetro que sea un array multidimensional de cualquier tamaño, siempre que el tamaño de cada dimensión se especifique usando parámetros de la propia función, que estén previamente declarados.

En la Fig. 47 aparece la función `muestra` con tres parámetros. Los dos primeros representan el tamaño de las dos dimensiones del array que se espera como tercer parámetro.

```
void muestra( int filas, int columnas, int array[filas][columnas] ) {
  for ( int i = 0 ; i < filas ; i = i + 1 ) {
    for ( int j = 0 ; j < columnas ; j = j + 1 ) {
      printf("%d ", array[i][j]);
    }
    printf("\n");
  }
  printf("\n");
}

int main() {
  int matrizA[2][3] = { {1,2,3}, {4,5,6} };
  int matrizB[3][2] = { {1,2}, {3,4}, {5,6} };
  muestra( 2, 3, matrizA );
  muestra( 3, 2, matrizB );
}
```

Fig. 47. Ejemplo de función con parámetro de tipo array de tamaño variable automático

Arrays de estructuras

Los arrays pueden contener valores de cualquier tipo, incluidos los creados por el usuario agregando otros. Para acceder a los campos de las estructuras almacenadas se usa el operador punto como habitualmente.

```
struct PuntoRep {
  int x;
  int y;
};

typedef struct PuntoRep Punto;

int main() {
  Punto segmento[2];
  segmento[0].x = 0;
  segmento[0].y = 0;
  segmento[1].x = 100;
  segmento[1].y = 50;
}
```

Fig. 48. Ejemplo de array de estructuras

El mecanismo de indexación

El mecanismo de indexación es el sistema usado para calcular la posición en memoria de los elementos de un array a partir del valor usado con el operador [].

En C, el nombre de un array representa la posición que ocupa en memoria el primero de sus elementos. Dicha posición no cambia durante la ejecución del programa. Por tanto, los arrays no son variables sino constantes, y como tales, no es posible modificarlas. Esto implica que a un array no se le puede asignar otro array porque ambos son valores constantes.

Para acceder a los elementos de un array se utiliza el operador []. Y el valor encerrado entre corchetes se interpreta como el número de elementos que hay entre el primer elemento del array y el elemento al que se quiere acceder. De ahí que para acceder al primer elemento se use el valor cero.

Si a ese número, multiplicado por la cantidad de de bytes que ocupa cada elemento en memoria, se le suma la dirección del primero de sus elementos, se obtiene la dirección en memoria del elemento deseado.

Por ejemplo, en la Fig. 49 se muestra el cálculo para acceder al tercer elemento del array `datos`. A la dirección del primer elemento (24) se le suma el resultado de multiplicar 2, que es el valor usado con el operador [], por el tamaño que ocupa en memoria cada elemento. En este caso es un entero, y normalmente ocupan 4 bytes.

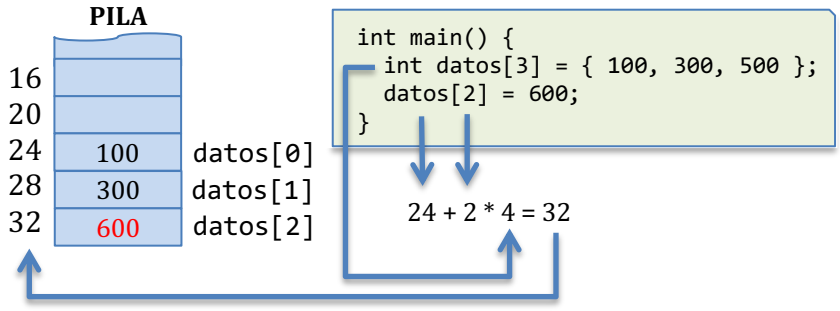

Fig. 49. Cálculo de la posición en memoria de los elementos de un array

Cadenas de caracteres

Un array puede guardar elementos de cualquier tipo. Por ejemplo, se puede declarar y usar un array de caracteres así:

```
int main() {
  char letras[] = {'P', 'a', 'c', 'o'};

  letras[0] = 'S';
  for ( int i = 0 ; i < 4 ; i = i + 1 ) {
    printf( "%c", letras[i] ) ;
  }
  printf( "\n" );
}
```

Fig. 50. Ejemplo de uso de un array de caracteres

En el ejemplo anterior se ve como se puede manejar un array de caracteres, pero esto no es lo mismo que el Tipo de Datos **string** de Pascal.

Una cadena de caracteres en C es una secuencia de caracteres acabada en una marca de fin. Por convenio, la marca de fin se asume que es el carácter ASCII `0`, que se puede escribir con el valor literal `'\0'`.

Las cadenas de caracteres se guardan en arrays de caracteres, pero el tamaño de ambos no tiene por qué coincidir ya que el final de la cadena estará indicado por la marca de fin, independientemente de la capacidad del array. Evidentemente, el array que almacena una cadena debería tener el tamaño suficiente como para contenerla entera, incluyendo la marca de fin.

La función `printf` puede mostrar cadenas de caracteres usando el comodín `%s`.

```
char nombre[] = {'P', 'a', 'c', 'o', '\0'};
printf( "%s\n", nombre );
```

Fig. 51. Ejemplo de cadena de caracteres declarada letra a letra

Para simplificar la escritura de código, C permite inicializar arrays de caracteres asignándoles una cadena de caracteres literal en la declaración. En estos casos, C se encarga de contar el número de caracteres encerrado entre las comillas dobles, de sumarle uno para tener en cuenta el espacio necesario para la marca de fin, de copiar dentro del array el texto, y de colocar al final la marca de fin. En el ejemplo izquierdo de la Fig. 52, C calcula automáticamente que el array `nombre` debe tener capacidad para 5 elementos: las cuatro letras de “Paco” y la marca de fin.

```
char nombre[] = "Paco";
```

```
char nombre[] = "Pa\0co";
```

Fig. 52. Ejemplos de cadena de caracteres declaradas con una cadena literal

Por otro lado, si se incluye la marca de fin de forma explícita a mitad de una cadena de caracteres se estará acortando la longitud efectiva de la misma. Por ejemplo, si la variable `nombre` se declarara como en el ejemplo derecho de la Fig. 52, C reservaría espacio para 6 caracteres, copiará todos los incluidos en la cadena literal, y añadirá la marca de fin como último carácter. Pero al imprimir el nombre con `printf` sólo se mostrará el texto `"Pa"`.

Además de `printf`, la biblioteca estándar incluye muchas otras funciones que realizan operaciones con cadenas de caracteres usando el mismo convenio de la marca de fin: `strcpy`, `strcmp`, `length`, etc. Y, evidentemente, es posible crear otras funciones que trabajen con cadenas de caracteres del mismo modo.

Lo interesante del uso de una marca de fin es que permite escribir funciones que no necesitan recibir el tamaño del array a través de un parámetro, pues siempre es posible detectar dónde acaba la secuencia de caracteres. Por ejemplo, para escribir una función que calcule la longitud de una cadena de caracteres, lo único que hay que hacer es contar cuántos caracteres hay antes de encontrar la marca de fin.

```
int longitud( char cadena[] ) {
  int contador = 0;
  int i = 0;
  while ( cadena[i] != '\0' ) {
    contador = contador + 1;
    i = i + 1;
  }
  return contador;
}
```

Fig. 53. Ejemplo de función que recibe un parámetro de tipo cadena de caracteres

En este otro ejemplo se modifica la cadena recibida como parámetro poniendo todos sus caracteres en mayúsculas. Para ello se usa la función `toupper`, declarada en el fichero de cabecera `<ctype.h>`

```
void mayusculas( char cadena[] ) {
  int i = 0;
  while ( cadena[i] != '\0' ) {
    cadena[i] = toupper( cadena[i] ) ;
    i = i + 1;
  }
}
```

Fig. 54. Función que pasa toda una cadena a mayúsculas

Finalmente, vamos a ver un ejemplo de uso de la función `sprintf` de la biblioteca estándar de C. Esta función nos permite generar cadenas de texto con formato usando valores guardados en variables igual que `printf` pero, en lugar de mostrar el resultado en la pantalla lo guarda en un array de caracteres creando una nueva cadena.

```
int main(){
  char resultado[80];
  int edad = 18;
  sprintf( resultado, "Mi edad es %d\n", edad );
  printf( "%s", resultado);
}
```

Fig. 55. Ejemplo de uso de la función sprintf

En el ejemplo de la Fig. 55, tras ejecutarse `sprintf`, el array resultado contendrá el texto “Mi edad es 18” acabado en un salto de línea. Al usar esta función lo más importante es asegurarse de que el array en el que se escribe tiene capacidad suficiente porque la función no lo comprueba. En este caso, 80 caracteres son más que suficientes para el texto que se genera cuya longitud será como máximo de 16 caracteres cuando `edad` almacene un número de tres dígitos.

Punteros

Un puntero es una dirección de memoria. En C se pueden declarar variables de tipo puntero, cuyos valores serán direcciones de memoria. Los punteros se pueden usar para acceder a los valores de otras variables, para lo que, en primer lugar, se debe conocer la dirección de memoria de esas variables. Pero también se usan para acceder a zonas de memoria no ligadas a ninguna variable. Esas zonas de memoria habrá que reservarlas y liberarlas en tiempo de ejecución de forma explícitamente.

Para declarar una variable de tipo puntero hay que indicar el Tipo de Datos al que apuntará. No es lo mismo apuntar a un carácter que a un valor real, pues la cantidad de memoria que ocupa cada Tipo de Datos puede ser distinta, y la forma de interpretar los bits allí guardados también.

Para declarar una variable de tipo puntero se escribe el nombre del Tipo de Datos al que apuntará, un asterisco y el nombre de la variable de tipo puntero que se está declarando.

```
int * iptr;    // Puntero a entero
char * cptr;   // Puntero a carácter
double * dptr; // Puntero a real
```

Fig. 56. Declaración de variables de tipo puntero

Los operadores & y *

Para obtener la dirección de memoria de una variable se usa el operador unario &. Y cuando un puntero guarda la dirección de otra variable se dice que apunta a ella.

```
int i = 3;
char c = 'a';
double d = 1.5;
int * iptr;
char * cptr;
double * dptr;

iptr = &i; // iptr apunta a la variable i
cptr = &c; // cptr apunta a la variable c
dptr = &d; // dptr apunta a la variable d
```

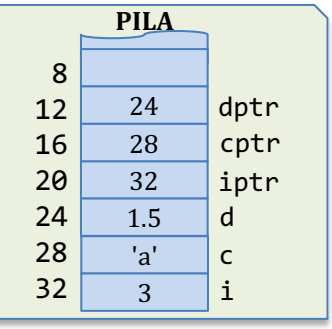

Fig. 57. Uso del operador & para obtener la dirección de una variable

Usando el operador &, los tres punteros declarados en la Fig. 57 toman el valor de las direcciones de memoria de las tres primeras variables declaradas. Como se puede ver en la representación gráfica de la Pila del Sistema, el valor de cada variable de tipo puntero se corresponde con la dirección que tiene cada una de las variables a las que apuntan.

Como pasa con cualquier otra variable, es conveniente dar un valor inicial a las variables de tipo puntero. Si en el momento de la declaración ya se tiene claro a qué otra variable debe apuntar el puntero, se le puede asignar su dirección usando el operador &. Si no, siempre se le puede asignar la constante `NULL`, definida en el fichero de cabecera `<stdlib.h>` y cuyo valor es cero. Esta no es una dirección válida, por lo que sirve para indicar que el puntero no tiene un valor correcto.

```
int i = 3;
int * iptr = &i;
char * cptr = NULL;
double * dptr = NULL;
```

Fig. 58. Inicialización de variables de tipo puntero con la constante NULL

Una vez que se tiene un puntero apuntando a una variable, es posible acceder a su contenido. El símbolo * delante de un puntero representa el dato al que apunta el puntero. Este es el denominado "operador de indirección o desreferencia", pues utiliza el valor guardado en la variable de tipo puntero para acceder al contenido de la zona de memoria referida.

En el ejemplo de la Fig. 59, se usan los punteros `iptr` y `cptr` y el operador de desreferencia para modificar el valor de las variables `i` y `c` de forma indirecta. También se accede indirectamente al valor de la variable `d` para mostrarlo con una llamada a `printf` a través del puntero `dptr`. Evidentemente, si las variables `i`, `c` y `d` se pueden usar directamente, no tiene mucho sentido usar punteros para acceder a su valor. Pero, en el siguiente apartado se verá la utilidad de esta herramienta.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int i = 3;
  char c = 'a';
  double d = 1.5;
  int * iptr = &i;      // iptr apunta a la variable i
  char * cptr = NULL;
  double * dptr = NULL;
  cptr = &c;            // cptr apunta a la variable c
  dptr = &d;            // dptr apunta a la variable d
  *iptr = 5;            // la variable i toma el valor 5
  *cptr = 'b';          // la variable c toma el valor 'b'
  printf( "%d\n", i );
  printf( "%f\n", *dptr );
  printf( "%c\n", c );
}
```

Fig. 59. Ejemplo de uso del operador de los operadores & y *

Simulación de paso de parámetros por referencia

Del mismo modo que se declaran variables de tipo puntero también es posible declarar parámetros de funciones de este tipo. Esto sirve para **simular un paso de parámetro por referencia**, pues desde el interior de la función es posible modificar el valor de las variables a las que apunten los parámetros de tipo puntero. Obviamente, será necesario llamar a la función pasándole la dirección de la variable que se desee modificar, que habrá sido obtenida con el operador `&`.

```
#include <stdio.h>

void incrementa( int * iptr ) {
  *iptr = *iptr + 1;
}

int main() {
  int i = 0;
  incrementa( &i );
  printf( "%d\n", i );  // Se muestra un uno
}
```

Fig. 60. Simulación de paso de parámetros por referencia usando punteros

La función `incrementa` mostrada en la Fig. 60 recibe como parámetro la dirección de una variable de tipo entero. En su código, incrementa en una unidad el valor de la variable apuntada por `iptr`. Desde la función `main` se llama a `incrementa` usando como argumento la dirección de la variable `i`. Al terminar la llamada, `i` valdrá uno.

La función scanf

La función `scanf`, declarada en el fichero de cabecera <stdio.h>, sirve para leer datos introducidos por teclado, y aplica la técnica de la simulación de paso de parámetro por referencia para conseguirlo.

```
#include <stdio.h>

int main() {
  int i = 0;
  printf( "Por favor, escribe un entero:\n" );
  scanf( "%d", &i );
  printf( "Entero leido = %d\n", i );
}
```

Fig. 61. Ejemplo de uso de la función scanf para leer un número entero

El formato de la cadena que se pasa como primer argumento en la llamada a `scanf` es similar al de `printf` e indica lo que se espera encontrar en el texto que introduzca el usuario. En el ejemplo de la Fig. 61 se usa el comodín `%d` para leer un número entero. El segundo argumento de la llamada a `scanf` es la dirección de la variable `i`. Por tanto, si el texto que se lee se puede interpretar como un número entero, su valor quedará almacenado en la variable `i`.

Si se desea leer un texto se usa el comodín `%s`. En ese caso, será necesario contar con un array de caracteres suficientemente grande como para almacenar todos los caracteres que introduzca el usuario. Y teniendo en cuenta que el nombre de un array representa la posición en memoria de su primer elemento, no es necesario usar & para calcularla, de hacerlo se obtendría el mismo resultado.

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

int main() {
  char cadena[80];
  printf( "Por favor, escribe un texto de menos de 80 caracteres:\n" );
  scanf( "%79s", cadena );    // cadena es equivalente a &cadena[0]
  printf( "Cadena leida = %s\n", cadena );
}
```

Fig. 62. Ejemplo de uso de la función scanf para leer una cadena

En el ejemplo de la Fig. 62 se declara el array `cadena` con capacidad para 80 caracteres. Al llamar a `scanf` se usa el comodín `%s`, pero se ha intercalado un número entre el símbolo de tanto porciento y la letra s. Así se consigue limitar la longitud de la cadena leída. En este caso, el límite se fija en 79 caracteres porque hay que dejar, al menos, un carácter libre en el array para la marca de fin.

La función `scanf` devuelve el número de valores que ha podido leer, y conviene comprobar que se han leído todos los datos que se esperaba. Especialmente, si se realizan varias llamadas consecutivas a `scanf` para leer diferentes tipos de datos.

```
  int n;
  int m;
  printf( "Por favor, escribe dos números enteros\n" );
  int res = scanf( "%d %d", &n, &m );
  if ( res != 2 ) printf( "Error en los datos de entrada\n" );
  // Usar n y m
```

Fig. 63. Ejemplo de uso del valor que devuelve scanf para detectar errores en la entrada de datos

Punteros a estructuras

Además de a tipos simples, también es posible declarar y usar punteros a estructuras. En este caso se puede usar la pareja de símbolos -> para acceder a cada uno de los campos de la estructura apuntada.

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

struct PuntoRep {
  int x;
  int y;
};

int main() {
  struct PuntoRep p = { .x = 0, .y = 0 };
  struct PuntoRep * pptr = &p;

  printf( "(%d,%d)\n", p.x, p.y );
  printf( "(%d,%d)\n", pptr->x, pptr->y );
  pptr->x = 10;
  pptr->y = 20;
  printf( "(%d,%d)\n", p.x, p.y );
  printf( "(%d,%d)\n", pptr->x, pptr->y );
}
```

Fig. 64. Ejemplo de uso de punteros a estructuras

Al utilizar punteros a estructuras como parámetros de una función se gana en eficiencia, pues no se necesita copiar toda la estructura al parámetro. Y además es posible modificar los campos de la variable usada como argumento en la llamada a la función. En el siguiente ejemplo, la función `mueve` desplaza el punto cuya dirección se recibe como primer parámetro tantas unidades en cada eje como indiquen los dos siguientes parámetros.

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

struct PuntoRep {
  int x;
  int y;
};

void mueve( struct PuntoRep * pptr, int dx, int dy ) {
  pptr->x = pptr->x + dx;
  pptr->y = pptr->y + dy;
}

int main() {
  struct PuntoRep p = { .x = 0, .y = 0 };
  struct PuntoRep * pptr = &p;

  printf( "(%d,%d)\n", p.x, p.y );
  printf( "(%d,%d)\n", pptr->x, pptr->y );
  mueve( &p, 10, 20 );
  printf( "(%d,%d)\n", p.x, p.y );
  printf( "(%d,%d)\n", pptr->x, pptr->y );
}
```

Fig. 65. Ejemplo de función cuyos parámetros son punteros a estructuras

Punteros a estructuras incompletas

C también permite declarar punteros a estructuras incompletas. Es decir, punteros a estructuras que aún no están definidas, o no lo están completamente.

```
#include <stdio.h>

typedef struct PuntoRep * Punto;

int main( int argc, char * argv[] ) {
  Punto pptr;             // Esta línea es correcta
  struct PuntoRep centro; // Esta línea da error de compilación
  ...
}

struct PuntoRep {
  int x;
  int y;
};
```

Fig. 66. Ejemplo de uso de punteros a estructuras incompletas

En el ejemplo de la Fig. 66 aparece la definición de `Punto` como alias de puntero a `struct PuntoRep`. Pero a esa altura del fichero, la estructura aún no está definida. Tampoco lo está a la altura de la declaración de la variable `pptr`, pero en ningún caso da error pues ambos son punteros. Sin embargo, la declaración de la variable `centro` sí que produce error porque no se trata de un puntero. Las definiciones y declaraciones de punteros a estructuras incompletas son válidas siempre que, finalmente, aparezca la declaración de la estructura en algún lugar del código fuente. Y en este caso, la estructura aparece definida al final del fichero `main.c`. Esta técnica resultará muy útil cuando se estudie cómo implementar Tipos de Datos Abstractos en C.

Por otro lado, los punteros a estructuras incompletas también se utilizan al definir estructuras recursivas. Es decir, estructuras cuyos campos apuntan a estructuras del mismo Tipo de Datos que se está definiendo.

```
struct Nodo {
  int dato;
  struct Nodo * sig;
};
```

Fig. 67. Definición de una estructura recursiva

La estructura definida en la Fig. 67 representa un nodo de una lista. Y aunque se estudiará con detalle en temas posteriores, es evidente que lo que representa es una secuencia de elementos del mismo tipo. Como puede verse, en el momento en que se está definiendo el campo `sig` la propia estructura `struct Nodo` no está completamente definida. Gracias a que C permite declarar punteros a estructuras incompletas es posible crear este tipo de estructuras.

Arrays y punteros

Los punteros y los arrays son muy parecidos en C. Ambos representan direcciones de memoria, y con ambos se puede usar el operador `[]` para acceder a los diferentes elementos situados consecutivamente en memoria a partir de la dirección que representan.

De hecho, como el nombre de un array representa la dirección en memoria de su primer elemento, su valor es el mismo que el que devuelve el operador `&` aplicado a su primer elemento. Por tanto, si los Tipos de Datos coinciden, a una variable de tipo puntero se le puede asignar el valor de una variable que represente un array.

```
int datos[] = {10, 20, 30};
int * iptr1 = datos;
int * iptr2 = &datos[0];
printf( "%p %p %p\n", datos, iptr1, iptr2 );  // El comodín %p sirve para punteros
```

Fig. 68. Demostración de que el nombre de un array representa la dirección de su primer elemento

En la Fig. 68 se declara un array de enteros llamado `datos`, y dos punteros a enteros. El primer puntero toma el valor del propio array, y al segundo puntero se le asigna el resultado de aplicar el operador `&` al primer elemento del array. Al mostrar por pantalla los valores de las tres variables, el resultado es el mismo.

El operador `[]` se usa para referirse a los elementos incluidos en un array, pero también se puede usar con variables de tipo puntero. Obviamente, para que esto tenga sentido, el puntero debe contener la dirección de memoria de una zona en la que haya varios elementos del mismo tipo de forma consecutiva. Y, como se acaba de ver, una forma de conseguirlo es hacer que el puntero tome el valor de un array.

```
#include <stdio.h>

int main() {
  int datos[] = {1, 3, 5};
  int * iptr = NULL;

  iptr = datos;
  iptr[0] = 100; // Equivale a *iptr = 100
  iptr[2] = 500;
  printf( "%d\n", datos[0] );          // Muestra 100
  printf( "%d\n", iptr[1] * iptr[2] ); // Muestra 1500
}
```

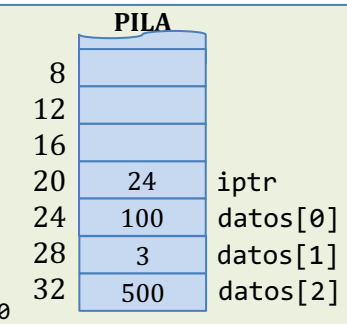

Fig. 69. Ejemplo de uso de un puntero como si fuera un array

En el ejemplo de la Fig. 69 se declara un array de enteros y la variable `iptr` de tipo puntero a entero. Tras hacer que la variable `iptr` apunte al primer elemento del array `datos`, se usa el operador `[]` para modificar tanto ese como el último de los elementos del array original. Después se muestra por pantalla que, efectivamente, el primer elemento del array ha cambiado. Y finalmente, otra vez usando `iptr`, se calcula y muestra el producto de los dos últimos elementos del array. Nótese la equivalencia entre `iptr[0]` y `*iptr`.

Una variable de tipo puntero en C puede utilizarse como si fuera un array. Pero un array no puede usarse como si fuera un puntero, puesto que es un valor constante, no una variable. De ahí que no se pueda asignar un array a otro.

Memoria Dinámica

El espacio de memoria requerido por las variables globales se conoce en el momento de compilar el programa. Estas variables se almacenan en la zona de datos. De las variables locales se conoce el tamaño que ocupará cada una, pero hasta que no se produce una llamada a función, no necesitan tener asignada una zona de memoria. Por eso existe la Pila del Sistema, para ir alojándolas a medida que haga falta. Tanto la reserva como la liberación de este espacio de memoria se hace automáticamente, por eso también se las suele llamar variables automáticas.

Sin embargo, hay situaciones que requieren usar una cantidad de memoria que no se conoce hasta que se ejecuta el programa. Por ejemplo, en un programa que primero pregunte al usuario cuántos datos quiere ordenar, y después pregunte por los datos, ¿dónde se almacenarán esos datos? Si en el programa se usa un array de un tamaño concreto puede que éste sea demasiado grande o demasiado pequeño.

Para solucionar estos problemas existe la denominada memoria dinámica o heap. Durante la ejecución del programa es posible reservar una porción de memoria de esa zona para, por ejemplo, guardar un array del tamaño indicado por el usuario. Y para poder manejar estas porciones de memoria reservadas se utilizan punteros. Pero estas reservas de memoria dinámica no se liberan de forma automática, hay que hacerlo de forma explícita en cuanto dejen de ser necesarias.

Las funciones malloc y free

Para reservar y liberar memoria dinámica se dispone de las funciones `malloc` y `free` declaradas en el fichero de cabecera `<stdlib.h>`. La función `malloc` reserva la cantidad de memoria que se le indique en bytes y devuelve la dirección del primer byte de la zona reservada, es decir, un puntero. La función `free` recibe como parámetro dicho puntero y libera toda la memoria reservada.

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

int main() {
  int * darray = NULL;
  darray = malloc( 10 * sizeof( int ) );
  darray[0] = 1;
  darray[9] = 9;
  for ( int i = 0 ; i < 10 ; i = i + 1 ) {
    printf( "%d\n", darray[i] );
  }
  free( darray );
}
```

Fig. 70. Creación y uso de un array en memoria dinámica

En el ejemplo de la Fig. 70 se usa `malloc` para reservar memoria dinámica para alojar un array de diez enteros. El número de bytes necesarios se calcula multiplicando el número de elementos del array por el tamaño en memoria necesario para cada elemento. Este dato se puede conocer usando el operador `sizeof` que devuelve los bytes que ocupa el Tipo de Datos que se le indique.

La dirección de memoria devuelta por `malloc` es almacenada en el puntero `darray`. Y como se vio anteriormente, es posible usar el operador `[]` para acceder a todos sus elementos. Finalmente, usando la función `free` se libera la memoria que se había reservado, dejándola otra vez disponible para futuras llamadas a `malloc`.

Funciones que devuelven arrays creados en memoria dinámica

Las funciones también pueden devolver valores de tipo puntero. Esto permite escribir funciones que devuelvan un array, pero este deberá estar alojado en memoria dinámica. De esta forma no desaparecerá automáticamente al terminar de ejecutarse la función, como ocurre con los arrays locales alojados en la pila.

Los elementos del array devuelto pueden ser de cualquier tipo, incluidos los definidos por el programador, y la función siempre se declara indicando que el Tipo de Datos devuelto es un puntero al Tipo de Datos de los elementos que contendrá el array. Por ejemplo, para crear una función que devuelva un array relleno con n números enteros generados aleatoriamente con la función `rand`, se define una función indicando que devolverá un valor de tipo puntero a entero:

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

int * rellena( int n ) {
  int * darray = NULL;
  darray = malloc( n * sizeof( int ) );
  for ( int i = 0 ; i < n ; i = i + 1 ) {
    darray[i] = rand();     // rand está declarada en <stdlib.h>
  }
  return darray;
}

int main() {
  int * numeros = NULL;
  numeros = rellena( 10 );
  for ( int i = 0 ; i < 10 ; i = i + 1 ) {
    printf( "%d\n", numeros[i] );
  }
  free( numeros );
}
```

Fig. 71. Ejemplo de una función que crea, rellena y devuelve un array en memoria dinámica

La función `rellena` mostrada en la Fig. 71 usa la variable local `darray`, que es de tipo puntero a entero, para gestionar la memoria reservada con `malloc`. Tras rellenar todo el array creado en la memoria dinámica se devuelve su dirección. La variable `darray` desaparece, pues es local a la función y está alojada en la Pila, pero la memoria a la que apuntaba no desaparece.

La función `rellena` reserva la memoria, pero cede la responsabilidad de su gestión a la función `main`, que es la función desde donde se invoca a `rellena`. Por lo tanto, al final de la función `main` se procede a la liberación de la memoria a la que, en ese momento, apunta la variable local `numeros`.

También se pueden escribir funciones que devuelvan cadenas de caracteres alojadas en memoria dinámica.

```
char * saludo( ) {
  char * cptr = malloc( sizeof( char ) * 4 );  // 3 letras y la marca de fin
  cptr[0] = 'H';
  cptr[1] = 'i';
  cptr[2] = '!';
  cptr[3] = '\0';        // OJO: No hay que olvidar colocar la marca de fin
  return cptr;
}
```

Fig. 72. Función que devuelve una cadena creada en memoria dinámica

La función `saludo` reserva memoria para almacenar cuatro caracteres: ¡los tres del texto "Hi!" y la marca de fin. Tras rellenarlo con el texto adecuado, en la última posición del array reservado se guarda el carácter que representa la marca de fin. Así pues, la zona de memoria se considerará una cadena de caracteres.

La función que llame a `saludo` debe encargarse de almacenar la dirección de memoria devuelta y de liberar la memoria asociada cuando ya no se necesite la cadena. Un error muy común se da al usar la llamada a `saludo` como argumento de `printf` como muestra el listado de la izquierda de la Fig. 73. Haciendo eso, la dirección de memoria de la cadena se pierde tras ser usada por `printf`. Al no haberla guardado antes no es posible usar `free` y la memoria no se libera.

```
int main() {
  // OJO: Uso incorrecto!!!
  printf( "%s\n", saludo() );
}
```

```
int main() {
  char * msg = saludo();
  printf( "%s\n", msg );
  free( msg );
}
```

Fig. 73. Usos de la función saludo: izquierda incorrecto, derecha correcto

La forma correcta de usar una función que devuelve la dirección de memoria de un nuevo array alojado en memoria dinámica es la mostrada en el listado de la derecha de la Fig. 73. Es decir, usando una variable temporal de tipo puntero para acceder a la zona de memoria reservada, y finalmente, liberarla.

Otra operación muy habitual es la de duplicar una cadena de caracteres. La siguiente función se encarga de hacerlo.

```
char * duplica( char * cadena ) {
  int n = longitud( cadena );  // Obtiene el número de caracteres de la cadena
  char * cptr = malloc( sizeof( char ) * (n+1) ); // n+1 para contar la marca de fin
  int i = 0;
  while ( i < n ) {
    cptr[i] = cadena[i];
    i = i + 1;
  }
  cptr[n] = '\0'; // Se coloca la marca de fin como en la última posición del array
  return cptr;
}
```

Fig. 74. Función que duplica una cadena usando memoria dinámica

En la función `duplica` mostrada en la Fig. 74 hay que destacar cuatro cosas:

- El parámetro `cadena` se declara como un puntero a carácter. Esta forma de declarar un parámetro para recibir cadenas de caracteres es muy habitual. Es equivalente a declararlo como `char cadena[]`, pues un puntero a carácter es precisamente un array de caracteres de tamaño no especificado.
- La función `longitud`, definida en la Fig. 53, se usa para calcular el número de caracteres que tiene la cadena original recibida como parámetro. Esto es necesario para poder reservar la memoria adecuada.
- Al reservar memoria con `malloc` se le suma uno a la longitud de la cadena para tener espacio para poder colocar la marca de fin.
- Tras copiar el contenido de la cadena original se coloca la marca de fin en la última posición del array apuntado por `cptr`, convirtiéndolo así formalmente en una cadena de caracteres.

Funciones que devuelven estructuras creadas en memoria dinámica

También es posible, y muy habitual, crear estructuras de datos en memoria dinámica con los Tipos de Datos definidos por el programador. Por ejemplo, para escribir una función que devuelva la dirección en memoria dinámica de una nueva estructura que represente un punto en el plano se haría lo siguiente.

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

struct PuntoRep {
  int x;
  int y;
};

typedef struct PuntoRep * Punto;

Punto crea( int x, int y ) {
  Punto p = malloc( sizeof( struct PuntoRep ) );
  p->x = x;
  p->y = y;
  return p;
}

int main() {
  Punto centro = crea(10,20);
  printf("%d,%d\n", centro->x, centro->y );
  free( centro );
}
```

Fig. 75. Ejemplo de función que crea y devuelve una estructura en memoria dinámica

Como se puede ver en la Fig. 75, tras definir la estructura `struct PuntoRep` se usa la sentencia `typedef` para crear un alias llamado `Punto`. En concreto, un alias de puntero a la estructura `struct PuntoRep`. Esta forma de definir nombres de Tipos de Datos se usará habitualmente en el tema de Tipos de Datos Abstractos, pero de momento, baste con entender que simplifica la escritura del código que sigue.

Dentro de la función `crea`, el operador `sizeof` se aplica a la estructura `struct PuntoRep` para determinar su tamaño, y poder reservar así la cantidad de memoria adecuada. La dirección devuelta por `malloc` se almacena en la variable `p` declarada de tipo `Punto`, que como ya se ha dicho, es un puntero a la estructura `struct PuntoRep`.

Una vez que se tiene un puntero a una estructura se puede acceder a sus campos usando el operador `->`. Es importante asignar un valor inicial a todos los campos de una estructura que acaba de ser alojada en memoria dinámica. De no hacerlo, la probabilidad de que contengan valores aleatorios es muy alta. En este caso, se guarda en los campos de la nueva estructura los valores recibidos a través de los parámetros `x` e `y`.

La función `crea` termina devolviendo la dirección almacenada en el puntero `p`, para que sea la función `main` quien se encargue de gestionar el nuevo punto. Especialmente, de liberar la memoria cuando ya no sea necesario. Y `main` lo hace usando para ello la variable `centro`.

Funciones que devuelven arrays de estructuras en memoria dinámica

Cuando se necesita devolver más de una estructura, se puede hacer devolviendo un array en memoria dinámica cuyos elementos sean dichas estructuras. En el siguiente ejemplo se muestra cómo generar un array en memoria dinámica cuyos elementos serán estructuras que representarán Puntos.

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

struct PuntoRep {
  int x;
  int y;
};

struct PuntoRep * genera( int n ) {
  struct PuntoRep * puntos = malloc( n * sizeof( struct PuntoRep ) );
  for ( int i = 0 ; i < n ; i = i + 1 ) {
    puntos[i].x = rand();
    puntos[i].y = rand();
  }
  return puntos;
}

int main() {
  // Generar cien puntos aleatorios
  struct PuntoRep * dibujo = genera( 100 );
  // Usar los cien puntos contenidos en el array dibujo
  // Liberar el array con los cien puntos
  free( dibujo );
}
```

Fig. 76. Ejemplo de creación de un array en memoria dinámica que contiene estructuras

En el ejemplo mostrado en la Fig. 76 aparece la función `genera`. Lo que hace es reservar memoria para almacenar un array con capacidad para guardar `n` variables de tipo `struct PuntoRep`, es decir, `n` puntos. Una vez hecho, rellena los campos de cada punto incluido en el array con valores aleatorios. Para acceder a cada campo se usa el operador `.` y no `->` puesto que los elementos del array son estructuras `struct PuntoRep`, no punteros a las mismas. Finalmente devuelve la dirección del nuevo array alojado en memoria dinámica.

En la función `main`, se crea uno de estos arrays con cien puntos, se guarda la dirección en la variable `dibujo` y después de usar los puntos del array, se libera la memoria reservada con una llamada a `free`.

Por otro lado, algunas veces interesa[2] mantener una colección de estructuras en memoria dinámica. Esto se puede hacer con un array de punteros a dichas estructuras. Si, además, el tamaño de dicho array se desconoce hasta el momento en que se vaya a usar, será necesario crearlo en memoria dinámica. En estos casos lo que se hace es escribir una función que devuelva un array en memoria dinámica cuyos elementos serán punteros a estructuras que también estarán alojadas en memoria dinámica. El siguiente apartado explica cómo hacerlo.

[2] Por ejemplo, para ordenar un array puede resultar más eficiente realizar intercambios entre valores de tipo puntero que entre valores de tipo `struct`.

Funciones que devuelven arrays de punteros a estructuras en memoria dinámica

En el siguiente ejemplo se muestra cómo generar un array en memoria dinámica cuyos elementos serán punteros que apuntarán a Puntos, que serán estructuras que también estarán en memoria dinámica.

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

struct PuntoRep {
  int x;
  int y;
};

typedef struct PuntoRep * Punto;

Punto crea( int x, int y ) {
  Punto p = malloc( sizeof( struct PuntoRep ) );
  p->x = x;
  p->y = y;
  return p;
}

Punto * genera( int n ) {
  Punto * puntos = malloc( n * sizeof( Punto ) );
  for ( int i = 0 ; i < n ; i = i + 1 ) {
    puntos[i] = crea( rand(), rand() );
  }
  return puntos;
}

int main() {
  // Generar cien puntos aleatorios
  Punto * dibujo = genera( 100 );
  // Usar los cien puntos contenidos en el array dibujo
  // Liberar los cien puntos y el array que los contenía
  for ( int i = 0 ; i < 100 ; i = i + 1 ) {
    free( dibujo[i] );
  }
  free( dibujo );
}
```

Fig. 77. Ejemplo de creación de un array en memoria dinámica que contiene estructuras creadas en memoria dinámica

En el ejemplo mostrado en la Fig. 77 aparece la función `crea`, estudiada anteriormente, y encargada de crear un punto en memoria dinámica. La función `genera` es nueva, y lo que hace es reservar memoria para almacenar un array con capacidad para guardar `n` variables de tipo `Punto`. Una vez hecho, rellena dicho array con puntos obtenidos llamando a `crea` con valores aleatorios. Finalmente devuelve la dirección del nuevo array alojado en memoria dinámica.

En la función `main`, se crea uno de estos arrays con cien puntos, y después de usarlos, se libera siguiendo el orden inverso al de su creación. Primero el contenido del array, y finalmente, el propio array.

La diferencia entre este ejemplo y el de la Fig. 76 está en que en este caso sí es necesario liberar de forma explícita la memoria de cada punto, mientras que en el ejemplo anterior no. En general, el número de llamadas a `malloc` y a `free` debe coincidir, si no, quedará memoria sin liberar.

Uso avanzado de punteros y memoria dinámica

Aritmética de punteros

Como los punteros son direcciones de memoria, y una dirección de memoria no es más que un número entero positivo, es posible realizar algunas operaciones aritméticas y lógicas con variables de este tipo. En concreto, se puede sumar o restar un valor entero a un puntero. El resultado es la dirección equivalente a avanzar/retroceder en memoria tantos elementos del Tipo de Datos apuntado por el puntero, como indique el entero que se suma o resta.

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

int main() {
  char cadena[] = "Uno, dos, tres";
  char * cptr1 = cadena + 3;            // Apunta a la primera coma
  char * cptr2 = cadena + 5;            // Apunta a la letra 'd'
  *cptr1 = '\0';                        // Equivalente a cptr1[0] = '\0';
  printf( "%s %s\n", cadena, cptr2);
}
```

Fig. 78. Ejemplo de aritmética de punteros con cadenas

La Fig. 78 muestra un ejemplo típico de uso de aritmética de punteros en conjunción con cadenas de caracteres. La variable `cptr1` toma el valor de `cadena` incrementado tres unidades, es decir, apunta al cuarto elemento del array original, que inicialmente, contiene una coma. La variable `cptr2` se inicializa apuntando a la posición que ocupa la letra 'd' de la palabra "dos" en la cadena original. A continuación, se modifica el contenido de la zona de memoria apuntada por `cptr1`, lo que implica que la cadena original cambia. Por tanto, al mostrar la cadena de caracteres representada por la variable `cadena` sólo aparecerá el texto "Uno". Y al mostrar la cadena de caracteres representada por la variable `cptr2`, aparecerá el texto "dos, tres".

También se pueden comparar dos punteros para comprobar si son iguales o si uno es mayor que el otro. Estas operaciones con punteros se pueden utilizar para recorrer arrays. A continuación, se muestra un ejemplo avanzado de uso de esta técnica para invertir un array de caracteres sin utilizar el operador `[]`.

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

int main() {
  char cadena[] = "Hola";
  char * cptr1 = cadena;        // Apunta a la letra 'H'
  char * cptr2 = cadena + 3;    // Apunta a la letra 'a'
  while ( cptr1 < cptr2 ) {
    char aux = *cptr1;
    *cptr1 = *cptr2;
    *cptr2 = aux;
    cptr1 = cptr1 + 1;
    cptr2 = cptr2 - 1;
  }
  printf( "%s\n", cadena );
}
```

Fig. 79. Función que invierte una cadena de caracteres usando punteros

Arrays de punteros a carácter y los argumentos del programa

Los elementos de un array pueden ser punteros a cualquier Tipo de Datos. Por ejemplo, para almacenar varios textos se puede crear un array de punteros a carácter, y hacer que cada uno de ellos apunte a una cadena de caracteres literal.

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

char * mensajes[] = { "Hola", "Muy buenas" , "Saludos cordiales" };

int main() {
  n = rand() % 3;    // Obtiene un número aleatorio entre 0 y 2
  printf( "%s\n", mensajes[n] );
}
```

Fig. 80. Ejemplo de uso de un array de punteros

Usando un array de punteros a carácter en lugar de un array bidimensional de caracteres, cada cadena ocupará justo la cantidad de memoria que necesita. En un array bidimensional de caracteres todas las filas tienen el mismo tamaño, que puede ser demasiado para algunos textos y demasiado pequeño para otros.

Un ejemplo paradigmático de uso de este tipo de arrays de punteros a cadenas de caracteres literales es el de los parámetros de la función principal. Aunque es posible definir la función `main` sin parámetros como se ha hecho hasta ahora, normalmente se define con dos parámetros. Y través de ellos se reciben los argumentos usados al ejecutar la aplicación.

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

int main( int argc, char * argv[] ) {
  printf("Se ha ejecutado %s con los %d siguientes argumentos:\n",argv[0],argc-1);
  for ( int i = 1 ; i < argc ; i = i + 1 ) {
    printf( "%s\n", argv[i] );
  }
}
```

Fig. 81. Acceso a los argumentos utilizados al ejecutar el programa

El parámetro `argc` contiene el número de argumentos usados en la ejecución de la aplicación incluyendo al propio programa, por lo que su valor mínimo es 1. El segundo parámetro es un array de punteros a carácter, y cada uno apunta al comienzo de una cadena de caracteres literal con el texto usado como argumento.

Sin embargo, un puntero que apunte a una cadena de caracteres literal no se puede tratar exactamente igual que un array de caracteres. Mientras que en un array de caracteres es posible modificar sus elementos, el contenido de una cadena de caracteres literal apuntada por un puntero puede consultarse, pero no modificarse.

```
#include <stdio.h>

int main( int argc, char * argv[] ) {
  char texto[] = "Hola";
  char * pchar = "Hola";
  texto[0] = 'C'; // Ahora texto contiene “Cola”
  pchar[0] = 'C'; // Error durante la ejecución
}
```

Fig. 82. Diferencia entre un array de caracteres y un puntero a una cadena de caracteres literal

Arrays bidimensionales en memoria dinámica

Como un puntero puede tratarse como si fuera un array, un array de punteros se puede ver como un array de arrays. Así pues, en tiempo de ejecución y usando memoria dinámica, es posible crear una estructura de datos con forma de array bidimensional del tamaño necesario. La siguiente es una de las formas de hacerlo.

```
#include <stdlib.h>

typedef int * Fila;

int main() {
  int filas = 3;
  int columnas = 10;
  Fila * matriz = malloc( filas * sizeof ( Fila ) );
  for ( int i = 0 ; i < filas ; i = i + 1 ) {
    matriz[i] = malloc( columnas * sizeof( int ) );
  }
  // Rellenar de ceros
  for ( int i = 0 ; i < filas ; i = i + 1 ) {
    for ( int j = 0 ; j < columnas ; j = j + 1 ) {
      matriz[i][j] = 0;
    }
  }
  // Liberar memoria
  for ( int i = 0 ; i < filas ; i = i + 1 ) {
    free( matriz[i] );
  }
  free( matriz );
}
```

Fig. 83. Ejemplo de creación, uso y liberación de un array bidimensional en memoria dinámica

El tipo `Fila` se define como un alias de puntero a entero. De este modo se simplifica la declaración de la variable `matriz`, que se declara como puntero a `Fila`. Esto es lo mismo que decir que es un array con un número indefinido de filas. Así pues, tras determinar el número concreto de filas y columnas a crear, se reserva memoria para tantas filas como sea necesario. Nótese que `sizeof( Fila )` es equivalente a `sizeof( int * )` pero el código queda más claro de la primera forma. Seguidamente, con el array de punteros a enteros ya reservado en memoria, se procede a dar valor a cada uno de esos punteros. Y, teniendo en cuenta que cada fila debe tener tantos enteros como indique la variable `columnas`, para cada celda del array `matriz` se vuelve a usar `malloc` para reservar espacio suficiente para guardar todos los enteros de cada fila.

Posteriormente, la variable `matriz` se puede tratar como si fuera un array bidimensional estándar. Cada celda de la primera dimensión representa un array, por eso puede indexarse con los corchetes de la segunda dimensión. La única diferencia con un array bidimensional, declarado en el área de datos o en la Pila del Sistema, es la organización de los datos en memoria. En este caso, las filas de datos no tienen por qué estar situadas de forma consecutiva en memoria.

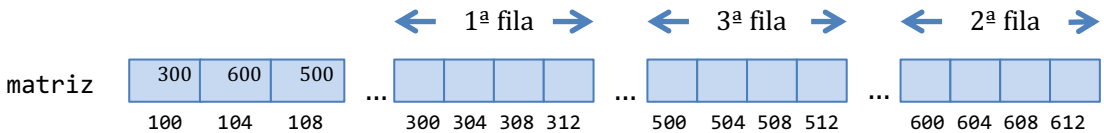

Fig. 84. Esquema de una posible distribución en memoria dinámica de un array bidimensional

Finalmente, se libera cada fila y después el array de filas en que consiste `matriz`.

Ficheros

Para trabajar con ficheros la biblioteca estándar de C incluye el Tipo de Datos llamado `FILE` y las funciones `fopen`, `fprintf`, `fscanf`, `fgets`, `fputs`, `feof` y `fclose`. El Tipo de Dato y las funciones están declarados en el fichero de cabecera <stdio.h>

El modo de trabajo es simple. Antes de poder leer o escribir en un fichero hay que abrirlo, y una vez que se ha terminado de usar se debe cerrar.

Para abrir un fichero se usa la función `fopen` indicando el nombre del fichero y el modo en el que va a ser usado. Tanto el nombre del fichero como el modo son cadenas de caracteres. El nombre puede incluir una ruta absoluta o relativa, y entre los modos de uso posible se encuentran los siguientes:

- `"r"`: el fichero se abre sólo para leer en él.
- `"w"`: el fichero se abre sólo para escribir en él.
- `"a"`: el fichero se abre para añadir nuevo contenido al final del mismo.
- `"r+"`: el fichero se abre para leer y escribir en él.
- `"w+"`: el fichero se abre para leer y escribir en él.
- `"a+"`: el fichero se abre para leer y añadir al final del mismo.

La función `fopen` devuelve un puntero a una estructura de tipo `FILE`. Este puntero representa el fichero y se debe usar como argumento en las llamadas al resto de funciones. Y, además:

- Si se produce un error al abrir el fichero la función devuelve `NULL`.
- Si se intenta abrir un fichero que no existe con los modos `"r"` o `"r+"` la función también devuelve `NULL`.
- Los modos `"w"` y `"w+"` crean el fichero si éste no existe, y si existe borran su contenido.
- Los modos `"a"` y `"a+"`crean el fichero si éste no existe, pero no borran su contenido en caso de que tenga alguno.

Para escribir en el fichero se usa `fprintf` y para leer `fscanf`. Ambas funcionan igual que `printf` y `scanf` pero realizan las operaciones en el fichero que reciben como primer argumento. Y cuando se ha terminado de usar el fichero hay que cerrarlo con `fclose`. De no hacerlo se pueden perder datos.

El ejemplo de la Fig. 85 muestra cómo crear un fichero en el que se escribe una línea con el nombre de una persona, un número, que puede ser su DNI, y una letra. Como puede verse, es importante comprobar el valor devuelto por `fopen`.

```
#include <stdio.h>

int main() {
  FILE * fp = NULL;

  fp = fopen( "datos.txt", "w" );
  if ( fp == NULL ) {
    printf( "Error abriendo o creando datos.txt\n" );
    return 0;
  }
  fprintf( fp, "Juan 43381037 S\n" );
  fclose( fp );
}
```

Fig. 85. Ejemplo de creación y escritura de un fichero

El siguiente programa lee el fichero creado con el ejemplo anterior.

```
#include <stdio.h>

int main() {
  FILE * fp = NULL;
  char name[80];
  int dni;
  char letra;

  fp = fopen( "datos.txt", "r");
  if ( fp == NULL ) {
    printf( "El fichero datos.txt no existe\n" );
    return 1;
  }
  int res = fscanf( fp, "%s %d %c", name, &dni, &letra );
  if ( res != 3 ) {
    printf("Error leyendo el fichero datos.txt \n" );
    return 1;
  }
  printf("Datos leidos: <%s> <%d> <%c>\n", name, dni, letra );
  fclose( fp );
}
```

Fig. 86. Ejemplo de programa que lee un fichero de texto

El programa de la Fig. 86 comienza intentando abrir el fichero en modo lectura. Si no lo consigue muestra un error y termina el programa devolviendo 1. El valor devuelto por la función `main` es interpretado por el Sistema Operativo como un código de error, siendo cero el indicador de que todo fue bien. Normalmente, la función `main` devuelve cero, aunque no se ponga `return` al final de la misma, pero cuando se detecta un error conviene terminar informando del mismo.

Una vez que el fichero está abierto, se usa `fscanf` para leer los tres datos que se supone que tiene guardados. Pero siempre se debe comprobar el valor que devuelve para detectar errores de formato en el fichero que se está leyendo.

Por otro lado, cuando no se conoce cuántos datos puede haber en un fichero, pero sí el formato en el que están, se puede usar la función `feof`, que devuelve 1 si se ha alcanzado el final del fichero. Por ejemplo, para leer todos los números enteros incluidos en un fichero se necesitaría un bucle similar al siguiente:

```
...
while ( !feof( fp ) )  {
  int res = fscanf( fp, "%d", &dato );
  if ( res != 1 ) {
    printf("Error leyendo el fichero datos.txt \n" );
    return 1;
  }
  printf("Dato leido: <%d>\n", dato );
}
...
```

Fig. 87. Ejemplo de programa que lee todos los datos de fichero

Como puede verse, el bucle repite iteraciones mientras que `feof` devuelve falso.

Por último, aunque `fprintf` y `fscanf` permiten trabajar con cadenas, las funciones `fgets` y `fputs` pueden resultar más cómodas. Además, por la forma en que actúan son capaces de evitar que se produzcan algunos errores típicos cuando los textos a leer son más grandes de lo esperado.