Programación en C

Jesús Rodríguez Heras

4 de julio de 2017

Índice general

Capítulo 1

Elementos básicos de la programación en C

1.1. Palabras reservadas del lenguaje C

El lenguaje C, como cualquier otro lenguaje de programación, posee un conjunto de palabras clave que tienen un especial significado para el compilador y sirven para indicar al computador que realice una tarea determinada. Estas palabras no pueden utilizarse como identificadores y son las siguientes:

- **break:** Se utiliza para salir de una sentencia `switch`.
- **case:** Sirve para etiquetar los diferentes casos de una sentencia `switch`.
- **char:** Tipo de dato para declarar variables de tipo carácter. Permite guardar un único carácter perteneciente al código ASCII comprendido entre 0 y 255.
- **const:** Indica que la variable que la acompaña no puede modificar su valor durante la ejecución del programa.
- **default:** Se utiliza en la sentencia `switch` para indicar el código que se ejecutará cuando ninguna de las etiquetas de `case` se corresponde con la expresión `switch`.
- **do:** Se usa para construir bucles iterativos que, al menos, se ejecutan una vez, ya que la condición de continuación del bucle se encuentra al final de éste.
- **double:** Tipo de dato para declarar variables de coma flotante de doble precisión.
- **else:** Se utiliza con `if` para controlar el flujo de ejecución del programa.
- **extern:** Se usa para indicar al compilador que una variable se declara en otro módulo del programa.
- **float:** Tipo de datos utilizado para declarar variables de coma flotante de simple precisión.
- **for:** Se usa para realizar bucles iterativos que se ejecutan mientras se cumple una determinada condición. La condición es evaluada antes de la ejecución del bucle por lo que puede que el bucle no se llegue a ejecutar si la condición no es cumplida.
- **if:** Se utiliza para indicar que el código asociado se ejecuta solo en el caso de que se cumpla la condición impuesta.
- **int:** Tipo de dato para declarar variables de tipo entero.
- **long:** Tipo de datos utilizado para declarar variables enteras que ocupan dos veces más bytes que los enteros de tipo `short`.

- **register:** es un especificador de almacenamiento para datos enteros, que se utiliza para informar al compilador de que el acceso a los datos debe ser tan rápido como sea posible, con lo cual el compilador almacenará los datos enteros en un registro de la CPU en lugar de situarlos en memoria.
- **return:** Se utiliza para detener la ejecución de la función actual y devolver el control a la sentencia que realiza la llamada a dicha función.
- **short:** Sirve para asegurar que el tamaño de las variables enteras es tan pequeño como sea posible.
- **static:** Es un modificador del tipo de datos que indica al compilador que cree un almacenamiento permanente para la variable local a la que precede, de forma que la variable tendrá un tiempo de vida global y retendrá su valor entre las distintas llamadas a la función en la que está definida.
- **struct:** Se utiliza para representar el tipo de datos estructura, que permite agrupar variables de diferentes tipos bajo un único identificador.
- **switch:** Se usa para realizar una bifurcación múltiple dependiendo del valor de una expresión.
- **typedef:** Se utiliza para dar un nuevo nombre (crear un alias) a un tipo de dato existente.
- **union:** Es un tipo de dato que permite almacenar en la misma posición de memoria diferentes datos (que generalmente son del mismo tipo).
- **unsigned:** Se utiliza para indicar al compilador que los tipos de datos enteros son solo positivos, con lo que no es necesario bit de signo y por lo tanto el rango de valores positivos queda duplicado.
- **void:** es un tipo de dato que se utiliza para indicar la inexistencia de un valor de retorno o argumentos en una declaración y definición de una función. También se utiliza para declarar un puntero a cualquier tipo de objeto dado.
- **while:** Permite construir un bucle cuyas sentencias interiores se ejecutan hasta que una condición o expresión se hace falsa. La condición es evaluada al principio, por lo que, de no cumplirse, las sentencias asociadas no llegan a ejecutarse ni siquiera una vez.

1.2. Estructura de un programa en C

La estructura general de un programa en C es la siguiente:

- **Inicio del programa:** Contiene la lista de importación (llamadas a bibliotecas), declaraciones globales, tipos, constantes y prototipos de funciones.
- **Programa principal (`main()`):** La función `main` es la primera función que es llamada cuando comienza la ejecución del programa. Esboza lo que el programa hace, y básicamente está compuesto por llamadas a otras funciones. Todo el cuerpo de la función debe ir comprendido entre llaves `{...}`. Las llaves, son la forma utilizada por el lenguaje C para agrupar varias sentencias de modo que se comporten como una única.
- **Implementación de las funciones:** Contiene la implementación de las funciones cuyos prototipos estaban declarados al inicio del programa. Una función es una porción de código agrupada bajo un nombre o identificador que realiza una determinada tarea.

A continuación veremos un ejemplo sencillo donde se emplea dicha estructura.

Ejemplo 1.1

```
/* Primer bloque: Inicio del programa */
#include <stdio.h>                  //Ficheros de cabecera
#include <conio.h>
#include <stdlib.h>

float factorial (int n);            //Proptotipos de funciones
float e;                            //Variables globales

/* Segundo bloque: Implementacion de las funciones */
float factorial (int n) {
    int j;
    float fact;
    fact = 1;
    for (j = 1; j <= n; j++)
        fact = fact * j;
    return fact;
}

/* Tercer bloque: Programa principal main() */
int main() {
    int i, n;
    do {
        system("cls");
        printf("Calculo del factorial de e segun la formula:\n");
        printf("e = 1 + 1/1! + 1/2! + ... + 1/10! + ...\n");
        printf("Indica el numero de terminos para clacular el numero e (max 10): ");
        scanf("%d", &n);
    } while (n < 1 || n > 10);
    e = 1;
    printf("e = 1");
    for (i = 1; i <= n-1; i++) {
        printf("+ 1/%d!", i);
        e = e + 1 / factorial(i);
    }
    printf("El numero e vale %f \n", e);
    return 0;
}
```

Explicación

El primer bloque comprende la primera declaración de las bibliotecas de funciones que el programa utilizará, junto con otras directivas como los prototipos de las funciones. También contiene la declaración de las estructuras de datos que serán globales al programa.

El segundo bloque incluye el código de todas las funciones definidas por el usuario, cuyos prototipos se encuentran declarados en el primer bloque. Este bloque puede ir en la posición que vemos en el ejemplo, o después del tercer bloque, dejando el tercer bloque (función `main` en segundo lugar).

El tercer bloque está formado por la función `main` que contiene el programa principal y dentro de ella se efectúan las llamadas l resto de funciones que hay en el programa. Esta función es obligatoria en todo programa en C, a diferencia de los bloques anteriores, que aparecen en el código solamente si son necesarios.

Las líneas que comienzan por `#include` indican cuales de las bibliotecas, de las definidas en C, van a ser utilizadas en el programa.

1.2.1. Funciones de biblioteca y de usuario

Las funciones de biblioteca son aquellas que vienen definidas e implementadas en alguna de las bibliotecas de C. Por ejemplo: `printf`, `scanf`, etc.

Por otra parte, las funciones de usuario son aquellas que están definidas e implementadas por el programador y realizan una función determinada para la cual han sido diseñadas.

1.2.2. Comentarios

El lenguaje C permite incluir comentarios dentro del código de cualquier programa. Los comentarios son anotaciones que el programador hace en el programa para explicar y facilitar la comprensión del mismo, con idea de que sea fácilmente entendible por cualquier otro programador distinto al que realiza el programa (o por el propio programador).

Los comentarios ignorados por el compilador cuando convierte el código fuente a código máquina, por lo que no influyen en la ejecución del mismo y pueden ser muy útiles a la hora de corregir un programa o comprender cómo está elaborado.

En C podemos usar dos tipos de comentarios:

```
1 /* Este tipo de comentario puede ocupar
2 tantas lineas como nos interese. */
3 // Este tipo solo puede ocupar una linea
```

1.2.3. Identificadores

Un **identificador** es el nombre con el que se hace referencia a una variable, constante o función. Cada identificador propio que utilicemos en un programa debe tener un nombre y ha de cumplir las siguientes reglas:

- La longitud de un identificador puede variar entre 1 y 32 caracteres. El primer carácter ha de ser una letra o "_", pero nunca un número (los siguientes caracteres si pueden ser números).
- Un identificador no puede contener espacios en blanco.
- Un identificador no puede ser una palabra reservada para el lenguaje C ni tener el mismo nombre que una función que se encuentre en la biblioteca de C.
- El lenguaje C distingue ente mayúsculas y minúsculas.
- Es aconsejable elegir un nombre significativo para una variable o función.

1.3. Tipos, variables y constantes

1.3.1. Tipos de datos fundamentales

En C existen cinco tipos de datos simples (`void`, `char`, `int`, `float` y `double`) y cada cual tiene un rango de valores definido y ocupa un determinado tamaño en memoria.

Tipo	Tamaño (bits)	Rango
void	0	Sin valor
char	8	-128 a 127
int	16	-32768 a 32767
float	32	$-3{,}4 \cdot 10^{38}$ a $3{,}4 \cdot 10^{38}$
double	64	$-1{,}7 \cdot 10^{308}$ a $1{,}7 \cdot 10^{308}$

Tabla 1.1: Tipos de datos básicos en C.

El tipo `void` se usa generalmente en la declaración de funciones para indicar que la función no devuelve valor alguno y/o para indicar que la función no tiene parámetros.

Todos los tipos básicos, excepto el tipo `void` y `float` tienen varios modificadores que alteran el rango de datos del tipo base y son los siguientes:

- `signed`: Indica que el tipo base puede ser positivo o negativo.
- `unsigned`: Indica que el tipo base no tiene signo (solo es positivo).
- `long`: Indica que el tipo base tiene un tamaño (rango) doble.
- `short`: Indica que el tipo base tiene un tamaño (rango) corto.

1.3.2. Duración y visibilidad de variables: Modos de almacenamiento

Además del tipo de una variable, el lenguaje C permite indicar como va a ser almacenada la variable. En C existen cuatro modos de almacenamiento fundamentales:

- `auto` (automático): Es la opción por defecto y la forma en que se guardan todas las variables que no especificamos cómo queremos que sean almacenadas. No se suele usar debido a que el compilador sobreentiende que si no se ha usado es porque se elige el almacenamiento automático.
- `extern`: Se aplica a variables globales que ha sido definida con el mismo identificador en otro fichero fuente, se usa en un programa implementado en módulos.
- `static`: Las variables así declaradas dentro de un bloque conservan su valor entre distintas ejecuciones de este bloque, es decir, permanecen en memoria durante toda la ejecución del programa. Cuando se llama a una función que utiliza variables así declaradas, éstas no pierden su valor inicial.
- `register`: Las variables así declaradas son almacenadas en los registros de la CPU en lugar de en la memoria, con objeto de que su acceso se más rápido. Se utiliza cuando nos interesa disminuir el tiempo de acceso a una variable y, como consecuencia, hacer que los cálculos que se realicen con ella sean más rápidos.

Ejemplo 1.2

```
1 #include <stdio.h>
2
3 int cuenta() {
4     int valor = 1;              //variable auto
5     static int n = 0;           //variable static
6     if (n < 10)
7         n++;
8     else
```

```
9           valor = 0;
10      return valor;
11  }
12
13  int main() {
14      register int n;              //variable register
15      ...
16      do {
17          ...
18      } while (cuenta() != 0);
19      return 0;
20  }
```

Explicación

Si observamos el ejemplo, en la función `cuenta()` tenemos declaradas dos variables locales `valor` y `n`. La variable `valor` tiene un modo de almacenamiento `auto` mientras que la variable `n` está declarada como `static`. Esto implica que la variable `valor` sea creada y destruida cada vez que se ejecuta la función `cuenta()`, mientras que, la variable `n` es creada e inicializada una única vez (la primera vez que se ejecuta la función `cuenta()`) y no se destruye cuando la función finaliza.

En la función `main()` tenemos declarada una variable local `n` en modo `register` lo cual, indica al compilador que almacene dicha variable en los registros de la CPU.

1.3.3. Declaración de variables

Para poder usar una variable es necesario declararla previamente ya que en caso de que no lo esté producirá un mensaje de error de compilación. Es posible dar un valor inicial a las variables en el momento de la declaración. Si una variable no es inicializada, el valor asociado a dicha variable es impredecible (basura informática).

Ejemplo 1.3

```
1  char letra1, letra2;
2  unsigned edad;
3  int valor;
4  long num_habitantes = 157956;
5  float precio = 27.45;
6  double microcalculo;
```

Explicación

Como podemos ver en este ejemplo, el la primera línea hemos declarado dos variables que nos permiten guardar un carácter en cada una de ellas. En la segunda línea declaramos una variable para almacenar la edad de una persona, que, como no puede ser negativa, la declaramos como `unsigned`. En la tercera línea declaramos una variable para almacenar un valor entero. En la cuarta línea declaramos una variable para almacenar el número de habitantes de una ciudad (declarado como `long` porque pueden ser más de 32767 habitantes). Finalmente, en las dos últimas líneas creamos dos variables, una para almacenar el precio de un productor y otra para almacenar un valor muy pequeño.

1.3.4. Constantes

Las constantes son datos cuyo valor no puede ser alterado durante la ejecución del programa. Un ejemplo típico de constante es el número PI (3.14159) o la constante de gravitación universal G (9.81).

En el código de un programa en C pueden aparecer diversos tipos de constantes:

- **Constantes enteras:** Están formadas por una secuencia de dígitos que representan un número entero. Dichas constantes están sujetas a las mismas restricciones de rangoque las variables de tipo `int` y `long`.
- **Constantes de coma flotante:** Pueden ser de tipo `float`, `double` o `long double` (si no se indica nada se supone de tipo `double`).
- **Constantes carácter:** Una constante carácter es un carácter cualquiera encerrado entre comillas simples. El valor de una constante carácter es el valor numérico asignado a ese carácter según el código ASCII. En C no existen constantes de tipo `char`, lo que se llama aquí constantes carácter son e realidad constates enteras.
- **Cadenas de caracteres:** Una cadena de caracteres es una secuencia de caracteres delimitada por comillas dobles (").

En C existen dos formas de cerar constantes:

const

El modificador de acceso `const` se usa para indicar que una variable no puede cambiar de valor durante la ejecución del programa. Se usa para crear constante,s, únicamente podremos asignar un valor a dichas variables en el momento de la declaración.

Si intentamos modificar el valor de una variable `const` el compilador dará un mensaje de error en la compilación.

Ejemplo 1.4

```
1 const int i = 10;          //Declaramos la variable i como constante
```

define

Otra forma de crear constantes o macros es mediante la directiva del preprocesador `#define` la cual se usa para crear macros.

La sintaxis de `#define` es la siguiente:

```
#define nombre_macro expresion_macro
```

La directiva `#define` no es una sentencia, por lo tanto, no termina en ";".

Las macros se eliminan con la directiva del preprocesador `#undef`.

Ejemplo 1.5

```
1 #define TRUE 1
2 #define MENSAJE "Pulse␣una␣tecla␣para␣continuar\n"
3 #undef TRUE
4 #undef MENSAJE
```

Explicación

En la línea 1 y 2 se definen las macros `TRUE` y `MENSAJE` y en las líneas 3 y 4 se eliminan dichas macros.

Cuando creamos una macro y la usamos, el compilador, al crear el fichero objeto, sustituye dicha macro por su definición cada vez que se la encuentra.

Ejemplo 1.6

```
#include <stdio.h>

#define MENSAJE "Pulse una tecla para continuar\n"

int main() {
    printf(MENSAJE);
    return 0;
}
```

Explicación

En este ejemplo, el compilador sustituiría la palabra `MENSAJE` por su valor, por lo que al ejecutar este programa obtendríamos lo siguiente: `Pulse una tecla para continuar`.

1.4. Operaciones aritmeticológicas

Los operadores son símbolos que indican la realización de una determinada operación sobre las variables, constantes o funciones sobre las que actúan en una expresión.

Los operadores pueden clasificarse atendiendo al número de operandos a los que afectan. Según esta clasificación pueden ser **unarios** o **binarios**: los primeros afectan a un solo operando mientras que los segundos a dos.

En C podemos distinguir varios tipos deoperadores, clasificados, según el tipo de acción que realizan: aritméticos, relacionales, lógicos y a nivel de bits.

1.4.1. Operador de asignación

El operador de asignación "=" es un operador unario. La asignación es la acción que permite establecer un valor a una variable.

La forma general de la sentencia de asignación es la siguiente:

`<nombre_variable> = <expresion>`

Donde `expresion` es una constante, una variable o una combinación de ambas. El destino (o parte izquierda de la asignación) ha de ser una variable, no una función ni una constante (que sí podría estar en el lado derecho de la asignación).

1.4.2. Operadores aritméticos

- Estos operadores aritméticos se pueden aplicar a constantes, variables y expresiones.
- Cuando el operador división (/) se aplica a operandos de tipo entero o de tipo carácter, se realiza una división entera.
- El operador "%"proporciona el resto de una división entera, por lo que no puede ser usado con tipos de coma flotante (`float`, `double`). Solo se aplica a los tipos entero (`int`, `long`).
- Si los operadores de incremento (++) o decremento (−−) aparecen delante de la variable, ésta es incrementada/decrementada antes de que el valor de dicha variable sea utilizado en la expresión en la que aparece. Si por el contrario, el operador aparece detrás de la variable, ésta es incrementada/decrementada después de ser utilizada en la expresión.
- Los operadores del mismo nivel de precedencia son evaluados por el compilador de izquierda a derecha.

Operador	Acción
$+$ (unario)	Suma
$-$ (unario)	Resta
$*$ (unario)	Multiplicación
$/$ (unario)	División
% (unario)	Resto de la división (módulo)
$++$ (binario)	Incremento
$--$ (binario)	Decremento

Tabla 1.2: Operadores aritméticos.

1.4.3. Operadores relacionales

Son operadores binarios que permiten comparar unas expresiones con otras, devolviendo el valor 1 cuando es cierta la comparación y 0 en caso contrario.

La forma general de los operadores relacionales es la siguiente:

```
<expresion1> operador <expresion2>;
```

Donde `operador` es uno de los siguientes operadores:

Operador	Acción
$>$	Mayor que
$>=$	Mayor o igual que
$<$	Menor que
$<=$	Menor o igual que
$==$	Igual que
$!=$	Distinto que

Tabla 1.3: Operadores relacionales.

1.4.4. Operadores lógicos

Son operadores binarios (a excepción de la negación) que permiten combinar de forma lógica los resultados de los operadores relacionales y son los siguientes:

Operador	Acción
&&	Y
||	O
!	No

Tabla 1.4: Operadores lógicos.

La tabla de verdad de los operadores lógicos es la siguiente:

p	q	p&&q	p||q	!p
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Tabla 1.5: Tabla de verdad de los operadores lógicos.

1.5. Punteros

Una variable puntero es una variable estática como cualquier otra que, en vez de contener valores de datos, contiene valores que representan direcciones de memoria. Estas direcciones, generalmente, representan la dirección de memoria donde se encuentra otra variable. Cuando el puntero contiene la dirección de otra variable decimos que el puntero apunta a dicha variable.

Para que pueda acceder correctamente a la variable a la que apunta, el puntero debe conocer que tipo de datos almacena. Así pues, el tipo del puntero debe coincidir con el de los datos a los que apunta.

El puntero se declara como cualquier otra variable, pero poniendo un asterisco (*) antes del nombre de la siguiente forma:

```
tipo *nombre_puntero;
```

Lo veremos con el siguiente ejemplo:

Ejemplo 1.7

```
1 int *p;
```

Explicación

En este ejemplo, el asterisco le indica al compilador que `p` no va a guardar un dato de tipo `int`, sino una dirección para apuntar a una variable de tipo `int`.

Los punteros ofrecen las siguientes ventajas:

- Mediante los punteros las funciones pueden modificar sus argumentos de llamada.
- Son indispensables para soportar las rutinas de asignación dinámica de memoria.
- El uso de punteros permite mejorar la eficiencia de ciertas rutinas.
- Los punteros se usan como soporte para la creación de ciertas estructuras de datos más complejas y dinámicas tales como las listas enlazadas, árboles, etc.

1.5.1. Inicialización de punteros

Al igual que otras variables, C no inicializa los punteros cuando se declaran sino que contienen un valor "basura"que no nos sirve para lo que lo queremos usar. Por lo tanto, es necesario inicializarlos antes de usarlos o hacer que no apunte qa ningún sitio (puntero nulo).

Para hacer que un puntero no apunte a ningún sitio le asignamos un valor especial llamado `NULL`.

Ejemplo 1.8

```
int *p;
p = NULL;
```

Explicación

En este ejemplo, el puntero `p` no apunta a nada. Utilizar un puntero con un valor `NULL` es más seguro que utilizar un puntero indefinido.

1.5.2. Los operadores punteros

Existen dos operadores especiales que se usan con los punteros: * y &.

El operador & (operador de dirección), aplicado sobre el nombre de una variable, devuelve su dirección de memoria.

Así, por ejemplo:

Ejemplo 1.9

```
int x, b, *p;
p = &x;
```

Explicación

En este ejemplo, se establece la dirección de memoria de la variable `x` en el puntero `p`, por lo que `p` apunta a `x`. El puntero tiene que ser del mismo tipo que la variable a la que apunta.

Dicha dirección es la posición interna de la variable en la memoria principal del ordenador, por lo tanto no tiene nada que ver con el valor de `x`. Se puede pensar en el operador & como devolviendo "la dirección de". ASí, la operación anterior significa: asigna a `p` la dirección de `x`.

El operador * (operador de contenido) aplicado sobre una dirección de memoria (variable de tipo puntero), permite acceder al dato al que apunta dicho puntero.

Siguiendo con el ejemplo anterior, para asignar el valor de `x` a otra variable, `b`, del mismo tipo, podríamos hacer dos cosas:

```
b = x;
```

O bien:

```
b = *p;
```

Como la variable puntero `p` contiene la dirección de memoria donde está almacenada la variable `x`, entonces `*p` nos devuelve el valor que existe en dicha dirección de memoria.

Por lo tanto podemos pensar en `*p` como el contenido de la dirección con lo que la sentencia anterior significa: asigna a `b` el valor contenido en la dirección `p`.

El concepto importante es comprender que la expresión `*p` no es más que otra forma de referirse a `x`. Por lo que cualquier cosa que hagamos con una se verá reflejada en la otra debido a que en realidad son el mismo objeto.

Ejemplo 1.10

```
/* Ejemplo de manejo de punteros */
#include <stdio.h>
#include <stdlib.h>

int main() {
    int dato, num = 527;          //Variable objetivo
    int *punt;                    //Puntero
    punt = &num;                  //Asigna al puntero punt la direccion de num
    dato = *punt;                 //Asigna a dato el valor contenido en la direccion
                                  de memoria apuntada por punt
    printf("El valor de num es %d\n", num);
    printf("El valor de *punt es %d\n", *punt);
    printf("El valor de dato es %d\n", daato);
    printf("La direccion de num es %u\n", &num);
    printf("El valor de punt es %u\n", punt);
    return 0;
} /* Fin del programa */
```

El resultado en pantalla es el siguiente:

```
El valor de num es 527
El valor de *punt es 527
El valor de dato es 527
La direccion de num es 65627
El valor de punt es 65627
```

Explicación

En este ejemplo se declara una variable y un puntero. A continuación, se asigna al puntero la dirección de la variable, y termina accediendo al dato de las dos formas: directa e indirectamente. Obsérvese que en dos `printf()` usamos el modificador `%u (unsigned)` cuando queremos representar direcciones, pues son números de 16 bits sin signo.

Debemos tener en cuenta que si ejecutamos este ejemplo en otro ordenador, la dirección de memoria no tiene que ser la que aquí aparece.

1.5.3. Aritmética, asignación y comparación de punteros

Los punteros se pueden sumar, restar, incrementar y decrementar como cualquier otra variable, ya que, en el fondo, las direcciones que almacenan los punteros son números enteros.

La única particularidad que tienen es que la "unidad" con la que se opera no es siempre igual a 1 (una celda de memoria), sino que los valores se incrementan o disminuyen en unidades iguales al número de bytes que ocupe el tipo del puntero.

Ejemplo 1.11

Dada la siguiente sentencia:

```
int *p;
```

Suponiendo que el puntero `p` apunta a la dirección 1000.

Si incrementamos de la siguiente forma: `p++;`

La variable `p` toma el valor 1002 debido a que el tipo `int` ocupa 2 bytes.

1.5.4. Punteros a punteros

De la misma forma que existen variables punteros que apuntan a la dirección de memoria de una variable, podemos tener punteros que apunten a la dirección de memoria donde se encuentra otra variable puntero. Es lo que se llama una forma de indirección múltiple o encadenamiento de punteros.

En el caso del puntero normal, el valor del puntero representa la dirección de memoria donde se encuentra la variable a la que apunta. En el caso de un puntero a puntero, el primer puntero contiene como valor la dirección de memoria del segundo puntero, el cual contendrá la dirección de memoria de la variable a al a que apunta.

Para declarar un puntero a puntero tiene que aneponerse un asterisco adiccional delante del nombre de la variable de la siguiente forma:

```
int **datos;
```

Donde `datos` es un puntero a un puntero a entero.

Ejemplo 1.12

```
#include <stdio.h>

int main() {
    int n, *a, **b;
    n = 10;
    a = &n;
    b = &a;
    printf("&n: %p, n: %d \n", &n, n);
    printf("&a: %p, a: %p, *a: %d \n", &a, a, *a);
    printf("&b: %p, b: %p, *b: %p, **b: %d \n", &b, b, *b, **b);
    return 0;
}
```

El resultado en pantalla es el siguiente:

```
&n: FFD0, n: 10
&a: FFD2, a: FFD=, *a: 10
&b: FFD4, b: FFD2, *b: FFD0, **b: 10
```

1.6. Operaciones básicas de entrada-salida por consola

La entrada/salida por consola se refiere a las operaciones que se producen en el teclado (entrada estándar) y la pantalla (salida estándar) de la computadora. El lenguaje C no posee sentencias de entrada/salida sino que utiliza funciones de la librería estándar del compilador `stdio.h` donde se encuentran definidos los prototipos de estas funciones de entrada/salida.

1.6.1. Funciones para leer o escribir un carácter

Prototipos de funciones que nos permiten leer o escribir un carácter son:

```
int getche (void);
int gecth (void);
int putchar (int c);
int putch (int c);
```

Estas funciones están definidas en la librería `conio.h` y se utilizan para:

- **getche():** Lee un carácter del teclado y lo muestra en pantalla. Esta función espera hasta que se pulsa una tecla y entonces devuelve su valor. No hace falta pulsar ENTER para procesar la tecla pulsada. En el momento en que la tecla es pulsada, ésta es capturada y mostrada en pantalla.
- **getch():** Lee un carácter del teclado sin mostrarlo en pantalla. Esta función es idéntica a la función `getche()` excepto que no muestra en pantalla el carácter introducido.
- **putchar():** Imprime un carácter en la pantalla en la posición actual del cursor. Devuelve el carácter escrito si todo va bien o EOF (fin de fichero) si se ha producido un error.
- **putch():** Imprime un carácter en la ventana activa, en la posición actual del cursor. Esta función es idéntica a la función `putchar()` salvo que el carácter se imprime en la ventana activa, en lugar de la pantalla.

Ejemplo 1.13

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

int main() {
    char ch;
    printf("Introduce una letra: \n");
    ch = getche();
    printf("Has pulsado la tecla '%c'\n", ch);
    printf("Introduce otra letra: \n");
    ch = getch();
    system("pause");
    return 0;
}
```

Explicación

En este ejemplo se muestra el funcionamiento de `getche()` y `getch()` descrito anteriormente.

1.6.2. Funciones para leer o escribir una cadena de caracteres

Prototipos de funciones que nos permiten leer o escribir cadenas de caracteres son:

```
chat *gets (char *cad);
char *cgets (char *cad);
char *cputs (const char *cad);
char *puts (conts char *cad);
```

En todas ellas `cad` es una cadena de caracteres (vector de caracteres).

Estas funciones están definidas en la librería `conio.h` y se utilizan para:

- **gets():** Lee una cadena de caracteres introducida por teclado hasta que se pulsa ENTER (dicho ENTER no forma parte de la cadena).
- **puts():** Imprime en pantalla la cadena de caracteres pasada como parámetro. A la cadena pasada como parámetro se le añade un carácter de retorno de carro al final, con lo que tras imprimirlo en pantalla, posiciona el cursor en la siguiente línea.

Ejemplo 1.14

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

int main() {
    char nombre[80];
    puts ("Introduce␣tu␣nombre:␣");
    gets(nombre);
    puts("Tu␣nombre␣es:␣");
    puts(nombre);
    getch();
    return 0;
}
```

Explicación

En este ejemplo se muestra el funcionamiento de `gets()` y `puts()` descrito anteriormente.

1.6.3. Funciones para leer o escribir con formato

Además de las funciones vistas anteriormente, el estándar ANSI de C define varias funciones de entrada/salida a las cuales se puede aplicar un formato determinado. Estas funciones son `printf()` y `scanf()`.

printf()

Esta funcion imprime en la salida estándar (la pantalla, por defecto) la cadena indicada como primer argumento junto con el valor de los otros argumentos, tras formatear en dicha cadena el resto de argumentos pasados como parámetros, de acuerdo a los formatos indicados en la cadena de formato. Si se produce algún error devuelve EOF (marca de fin de fichero).

El prototipo de esta función es el siguiente:

```
int printf(çadena_de_formato", argumento1, argumento2, ...);
```

Los puntos suspensivos indican que pueden existir un número variable de argumentos y la `cadena_de_formato` está formada por caracteres y por órdenes de formato.

Las órdenes de formato indican la forma en la que se muestran los argumentos posteriores. Los principales indicadores de formato son los siguientes:
En la lista de argumentos debe haber exactamente el mismo número de argumentos que de indicadores de formato y ambos han de coincidir en su orden de aparición de izquierda a derecha.

scanf()

Esta función permite la entrada por consola de propósito general. Permite leer datos de la entrada estándar (que por defecto es el teclado) e interpretarlos y convertirlos en cualquiera de los tipos de datos que soporta el lenguaje C.

El prototipo de esta función y su funcionamiento es muy similar a la función `printf()`:

```
int scanf("%x1%x2...", &argumento1, &argumento2);
```

Donde `x1, x2, ...` son los indicadores de formato, que representan los formatos con los que espera encontrar los datos.

En la lista de argumentos debe haber exactamente el número de argumentos que de indicadores de formato y ambos deben coincidir en su orden de aparición de izquierda a derecha.

Los indicadores de formato son los mismos que los usados en la función `printf()`.

Código	En dicha posición se mostrará:
%c	Un carácter
%i o %d	Un entero
%e	Un número en notación científica
%f	Un número con decimales (coma flotante)
%g	Un número en formato %e o %f (el más corto)
%o	Un entero en formato octal
%s	Una cadena de caracteres
%u	Un entero sin signo
%x	Un entero en formato hexadecimal
%p	Un puntero
%%	El signo %

Tabla 1.6: Indicadores de formato de printf()

Ejemplo 1.15

En este ejemplo haremos uso de las funciones `printf()` y `scanf()`.

```
1  #include <stdio.h>
2  #include <conio.h>
3
4  int main() {
5      char nombre[20];
6      int edad;
7      float altura;
8
9      printf("Introduce␣tu␣nombre:␣");
10     scanf("%19s", nombre);          //Acepta 19 caracteres como maximo
11     fflush(stdin);                  //Limpia el flujo de entrada por si queda basura
12
13     printf("Introduce␣tu␣edad:␣");
14     scanf("%d", edad);
15     fflush(stdin);
16
17     printf("Introduce␣tu␣altura:␣");
18     scanf("%f", altura);
19     fflush(stdin);
20
21     printf("Tu␣nombre␣es:␣%s\n", nombre);
22     printf("Tienes␣%d␣anos␣y␣mides␣%.2f␣metros.", edad, altura);
23     return 0;
24 }
```

Explicación

En este ejemplo hay dos aspectos a tener en cuenta:

- Lo primero es la función `fflush(stdin);` la cual realiza una limpieza del flujo de entrada para que, si queda basura, el `scanf()` no tome esa basura, sino el valor que nosotros estamos introduciendo por teclado.
- El segundo se encuentra en las líneas 5 y 10 del código. En la línea 5 reservamos espacio para almacenar 20 caracteres, con lo que podremos introducir un nombre de hasta 19 caracteres debido a que en C, las cadenas de caracteres terminan con un carácter terminador nulo (\0), el cual también hay que almacenarlo.

1.6.4. Constantes de carácter con barra invertida

Además de imprimir en pantalla caracteres simples, también podemos enviar a la consola ciertos caracteres especiales. Estas constantes especiales están formadas por la barra invertida (\) seguida de un carácter. las principales constantes son las siguientes:

Código	**Significado**
\b	Espacio atrás
\f	Salto de página
\n	Salto de línea
\r	Retorno de carro
\t	Tabulación horizontal
\"	Comilla doble
\'	Comilla simple
\0	Nulo
\\	Barra invertida
\v	Tabulación vertical
\a	Alerta %
\o	Constante octal %
\x	Constante hexadecimal %

Tabla 1.7: Constantes de carácter especial.

Capítulo 2

Control de flujo del programa

2.1. Introducción

Las sentencias de un programa en C se ejecutan una detrás de otra en el orden en el que éstas aparecen en el código fuente, a menos que una sentencia de control de flujo cambie ese orden. Con el fin de dirigir el flujo de ejecución del programa, C ofrece varias sentencias de control de flujo. Estas sentencias se basan en la evaluación de una prueba condicional que determina la acción a realizar. Una prueba condicional produce o bien un valor cierto o un valor falso. En C cualquier valor distinto de cero es cierto, siendo el cero el único valor falso.

2.2. Sentencias

Una sentencia en C es una expresión (combinación de operadores y operandos) terminada en punto y coma (;).

Las sentencias más comunes son las que incluyen expresiones de asignación, expresiones de función o una combinación de ambas.

2.3. Sentencias compuestas (bloques)

Una sentencia compuesta o bloque es un grupo de sentencias agrupadas entre llaves ({ y }). Las llaves se utilizan para agrupar declaraciones y sentencias dentro de un bloque, de modo que son sintácticamente equivalentes a una sentencia simple.

Los bloques presentan la siguiente sintaxis:

```
1 {   //Comienzo del bloque
2       [directivas]          //directivas del preprocesador: #include, etc.
3       [declaraciones]       //declaraciones de variables locales
4       [sentencias]          //conjunto de sentencias
5 }   //Final del bloque
```

Analizando esta sintaxis podemos observar que:

- Pueden declararse variables dentro de un bloque. Estas variables serán locales a dicho bloque, es decir, sólo podrán usarse dentro de del bloque en el que han sido declaradas.

- Puede existir un bloque nulo, es decir, un bloque sin directivas, ni declaraciones ni sentencias.
- Un bloque no termina en punto y coma, ya que éste solo se emplea para terminar las sentencias de expresión.

Las sentencias compuestas o bloques se utilizan con frecuencia en combinación con las estructuras de selección y de repetición, cuando nos interesa que un conjunto de instrucciones se ejecuten al cumplirse o mientras que se cumple una determinada condición.

2.4. Estructuras de selección

Las estructuras de selección nos permiten dirigir el flujo de ejecución de un programa a un determinado bloque de sentencias u otro en función del resultado de la evaluación de una expresión o condición.

Las estructuras de selección que nos ofrece el lenguaje C son las siguientes:

2.4.1. `if`

Esta sentencia se utiliza para expresar decisiones. Permite ejecutar un conjunto de sentencias y otro en función de si se cumple o no una determinada condición.

La sintaxis general es la siguiente:

```
1 if (condicion)
2     sentencia1; //se ejecuta si condicion es verdadera
3 else
4     sentencia2; //se ejecuta si condicion es falsa
```

En esta sintaxis podemos destacar lo siguiente:

- `condicion` es una expresión cuyo valor es un entero (0=falso, <>0=verdadero).
- `else` es una palabra clave opcional, es decir, la parte `else` es optativa (podemos tener una setnencia `if` que no tenga asociado ningún `else`).
- `sentencia1` y `sentencia2` es una sentencia o un conjunto de sentencias (bloques).

El funcionamiento de la sentencia `if` es el siguiente:

Una vez evaluada la condición pueden suceder dos cosas:

- Si es verdadera (tiene un valor distinto de cero), se ejecuta el bloque de sentencias asociado al `if`.
- Si es falsa (expresión igual a cero), se ejecuta, si existe, el bloque de sentencias asociado al `else` (si no existe no se ejecuta nada).

Ejemplo 2.1

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int edad;
6      printf("Introduce␣tu␣edad:␣");
7      scanf("%i", &edad);
8      if(edad >= 100)
9          printf("Eres␣muy␣mayor.\n");
10     else
11         printf("Aun␣no␣has␣vivido␣un␣siglo.\n");
12     return 0;
13 }
```

Explicación

En este ejemplo podemos ver como se pide al usuario que introduzca su edad y en función de si la condición (línea 8) es verdadera o falsa mostrará en pantalla una u otra expresión.

`if` anidados

Una sentencia `if` puede tener anidada otra sentencia `if` en su interior y así sucesivamente.

En el siguiente ejemplo podemos ver un caso de `if` anidados.

Ejemplo 2.2

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      float nota;
6      float practica;
7      printf("Introduce␣la␣nota␣del␣alumno:␣");
8      scanf("%i", &nota);
9      if (nota < 5)
10     {
11         if (practica > 6)
12             nota = nota + 1;
13         else
14             nota = nota - 0.5;
15     }
16     printf("La␣nota␣final␣del␣alumno␣es␣%.2f.", nota);
17 return 0;
18 }
```

Explicación

Como vemos en el ejemplo, si un alumno suspende, puede ver incrementada su nota en un punto si en las prácticas ha sacado más de un seis. Si en las prácticas ha sacado menos de un seis, encima de que no ha aprobado, su nota se

penaliza en medio punto. Para aquellas personas que han aprobado, su nota no se ve alterada en nada por el resultado de las prácticas.

`if...else` múltiples

Es la forma más general de escribir una decisión múltiple, de forma que solo se ejecuta, de entre todas las existentes, aquella sentencia o bloque de sentencias que cumpla la condición impuesta.

El funcionamiento es el siguiente:

Las condiciones se evalúan en el orden en el que aparecen y en el momento en el que una de las condiciones se cumple (es verdadera), la sentencia o bloque asociada con ella se ejecuta, siguiendo posteriormente la ejecución del programa en la instrucción siguiente existente fuera del bloque `if` (se pasa por alto el resto de sentencias `else` restantes).

Para entenderlo bien veamos el siguiente ejemplo:

Ejemplo 2.3

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

    int main() {
    float nota;
    printf("Introduce␣tu␣nota:␣");
    scanf("%i", &nota);
    if (nota == 10)
    {
        puts("Matricula␣de␣honor.");
        puts("Enhorabuena.");
    }
    else if (nota >= 9)
    {
        puts("Sobresaliente.");
        puts("Muy␣bien␣hecho.");
    }
    else if (nota >= 7)
        puts("Notable.");
    else if (nota >= 5)
        puts("Aprobado.");
    else
    {
        puts("Suspenso.");
        puts("Lo␣siento␣mucho.");
    }
    return 0;
}
```

Explicación

En este ejemplo, solo se ejecuta aquella sentencia o bloque de sentencias asociadas a la condición que se cumple. El resto de sentencias existentes en las restantes condiciones no se ejecutan.

2.4.2. El operador ?

Podemos usar el operador ternario `?` para reemplazar una sentencia `if else`, siempre y cuando la parte asociada al `if` y al `else` sean expresiones simples. Al ser un operador ternario usa tres operandos y tiene la siguiente sintaxis:

```
condicion ? expresion1 : expresion2;
```

Una vez que la condición se evalúa tenemos:

- Si la condición es cierta, se ejecuta `expresion1` y se convierte en el valor de la expresión completa.
- Si la condición es falsa, se ejecuta `expresion2` y su valor se convierte en el valor de la expresión completa.

Un ejemplo con este operador es el siguiente:

Ejemplo 2.4

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

int main() {
    int num = 25, valor;
    printf("Adivina el numero: ");
    scanf("%d", &valor);
    valor == num ? printf("Has acertado.") : printf("No has acertado.");
    return 0;
}
```

Explicación

Como podemos observar, el operador `?` contiene dos mensajes, el primero para cuando se cumple la condición y el segundo para cuando no se cumple la condición.

2.4.3. switch

Una alternativa al uso de construcciones `if...else` múltiples es el uso de la sentencia `switch`. Esta sentencia es una estructura de decisión múltiple que comprueba si una expresión coincide con uno de los valores constantes enteros especificados en cada una de las etiquetas que contiene, y traslada el control de ejecución al conjunto de instrucciones contenidas en al correspondiente etiqueta.

La sintaxis de la sentencia `switch` es la siguiente:

```
switch(expresion_entera)
{
    case constante1: sentencia1; [break;]
    case constante2: sentencia2; [break;]
    ...
    case constanteN: sentenciaN; [break;]
    [default: sentencia;]        //Opcional
}
```

Analizando esta sintaxis podemos observar que:

- `expresion_entera` es una expresión cuyo valor es un entero.
- `constante1...constanteN` es un valor o expresión constante entera, es decir, no contiene variables.
- `default` es una etiqueta opcional.
- `sentencia1...sentenciaN` es una sentencia o un conjunto de sentencias (bloques).
- `break` es una sentencia opcional que rompe el `switch` para que no se sigan ejecutando el resto de sentencias `case`.

Cuando se ejecuta la sentencia `switch`, se evalúa `expresion_entera` y se compara con cada una de las constantes etiquetadas en los `case`. Si una de las constantes de los `case` coincide con el valor de `expresion_entera`, el control del programa es transferido a las sentencias de dicha etiqueta `case` ejecutándose todas sus sentencias y las del resto de case existentes desde ese punto hasta el final de la sentencia `switch`, a menos que una sentencia `break` rompa el `switch` y transfiera el control a la sentencia inmediatamente posterior al bloque `switch`.

Si ninguna de las sentencias `case` coincide con `expresion_entera`, se ejecutan las sentencias contenidas en la etiqueta `default` en el caso de que existiese.

Si ningún `case` coincide con `expresion_entera` y no existe la etiqueta `default`, entonces no se ejecuta ninguna acción dentro del `switch`, sino que continúa la ejecución del programa por el conjunto de sentencias existentes a continuación de la estructura `switch`.

Un uso muy común de la sentencia `switch` es la creación de un menú como vemos en el siguiente ejemplo:

Ejemplo 2.5

```
1  void ver_menu() {
2      char c;
3      printf("1. Insertar.\n");
4      printf("2. Modificar.\n");
5      printf("3. Eliminar.\n");
6      printf("Introduzca una opcion: ");
7      scanf("%c", &c);
8      switch(c)
9      {
10         case '1': insertar();    //Llamada a la funcion insertar
11             break;
12         case '2': modificar();   //Llamada a la funcion modificar
13             break;
14         case '3': elmiminar();   //Llamada a la funcion eliminar
15             break;
16         default: printf("Elija una opcion correcta.");
17     }
18 }
```

Explicación

En este ejemplo, en función de la tecla pulsada (opción elegida), se ejecutará una u otra función.

Al igual que en este ejemplo se han utilizado los `case` para llamar a otras funciones también podemos escribir código normal en función de cada `case`.

Con respecto a la sentencia `switch` hay que tener en cuenta lo siguiente:

- No siempre es posible sustituir una construcción `if...else` múltiple por una sentencia `switch` debido a que ésta solo puede comprobar la igualdad, mientras que el `if...else` múltiple puede evaluar cualquier tipo de expresión relacional o lógica.
- Todos los `case` pertenecientes al mismo `switch` deben tener valores diferentes.
- Pueden existir etiquetas `case` vacías, en ese caso se ejecutarán las sentencias existentes en la etiqueta `case` siguiente a no ser que exista una sentencia `break`.
- Una sentencia `switch` puede tener anidada otra sentencia `switch` (o cualquier otra estructura de selección o repetición) en alguno de sus `case`. En este caso, las constantes `case` existentes en el `switch` anidado pueden tener el mismo valor que la de los `case` del `switch` más exterior, ya que son independientes.

2.5. Estructuras de repetición

Las estructuras de repetición nos permiten iterar el flujo de ejecución de un programa dentro de un de terminado bloque de sentencias mientras que se verifique una determinada condición o expresión. De modo genérico, a estas sentencias se las denomina bucles y son las siguientes:

2.5.1. `while`

Esta sentencia permite ejecutar repetidamente, mientras se cumpla una determinada condición, una sentencia o bloque de sentencias y su sintaxis es la siguiente:

```
1 while (condicion) {
2     sentencia_bloque;    //Se ejecuta mientras se verifique la condicion
3 }
```

Analizando esta sintaxis podemos observar que:

- `condicion` es una expresión cuyo valor es un entero (0=falso, <>0=verdadero).
- `sentencia_bloque` es una sentencia simple, una sentencia de control condicional o iterativa, o un conjunto de sentencias (bloque).

Las llaves no son necesarias si solo existe una sentencia asociada al bucle `while`. Si son varias sentencias las que queremos que se ejecuten cuando la condición del bucle sea verdad, debemos usar las llaves para así formar un bloque de sentencias.

La sentencia o bloque de sentencias se ejecuta repetidamente mientras que se verifique `condicion` (mientras que `condicion` sea distinta de cero). La expresión de `condicion` es evaluada antes de la ejecución de las sentencias, de manera que si no se verifica la expresión, no se ejecutan las sentencias del bucle `while`. Es decir, en cada iteración, antes de ejecutar las sentencias se vuelve a evaluar `condicion` y en función del resultado de la evaluación se ejecutarán las sentencias o se terminará el bucle.

Una vez terminado el bucle, el flujo del programa continúa por las sentencias existentes a continuación del bucle `while`.

Ejemplo 2.6

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

int main() {
    int i=1;
    char tecla;
    tecla = getch();
    while (tecla != 's' && i <= 20) {
        printf("%d\n", i++);
        tecla = getch();
    }
    return 0;
}
```

Explicación

En este ejemplo, antes de imprimir un número, el programa espera la pulsación de una tecla. Si la tecla pulsada es distinta de la letra 's', imprime un valor entero, empezando por el uno e incrementa dicho valor, repitiendo este proceso una y otra vez hasta que el usuario pulse la tecla 's' o hasta que haya pulsado un total de 20 teclas.

Si nunca pulsamos la tecla 's', el bucle irá iterando una y otra vez y en cada iteración se irá incrementando en una unidad la variable `i`, por lo que cuando llevemos un total de 20 iteraciones, la variable `i` alcanzará el valor 21. Esto provocará que el bucle termine al dejar de verificarse la condición `i <= 20`.

2.5.2. `do while`

Esta sentencia es similar a la sentencia `while`, excepto que la condición se evalúa después de haberse ejecutado la sentencia o bloque de sentencias existentes en el bucle. Su sintaxis es la siguiente:

```
do {
    sentencia_bloque;
} while (condicion);
```

Analizando esta sintaxis podemos observar que:

- `condicion` es una expresión cuyo valor es un entero (0=falso, <>0=verdadero).
- `sentencia_bloque` es una sentencia simple, una sentencia de control condicional o iterativa, o un conjunto de sentencias (bloque).

Al igual que en la sentencia `while` las llaves no son necesarias cuando solo hay una sentencia. No obstante, se suelen utilizar para evitar confusiones con el bucle `while`.

Al contrario que con el bucle `while` la sentencia o bloque de sentencias se ejecuta al menos una vez, ya que la ecaluación de la condición se produce con posterioridad. Al final de cada iteración, la expresión de `condicion` es evaluada de nuevo y mientras que esta se verifique (mientras que `condicion` sea distinta de cero) se seguirá ejecutando el bucle. Una vez terminado el bucle (por dejarse de cumplir `condicion`), el flujo del programa continua por las sentencias existentes a continuación del bucle `do while`.

El bucle `do while` se usa frecuentemente junto con el `switch` para procesar menús. Podemos verlo en acción en el siguiente ejemplo:

Ejemplo 2.7

```
void ver_menu() {
    char c;
    do {
        printf("1. Insertar.\n");
        printf("2. Modificar.\n");
        printf("3. Eliminar.\n");
        printf("4. Salir.\n");
        printf("Introduzca una opcion: ");
        scanf("%c", &c);
        switch(c)
        {
            case '1': insertar();        //Llamada a la funcion insertar
                break;
            case '2': modificar();       //Llamada a la funcion modificar
                break;
            case '3': elmiminar();       //Llamada a la funcion eliminar
                break;
            case '4': break;
            default: printf("Elija una opcion correcta.");
        }
    } while (c != '4');
}
```

Explicación

Este ejemplo es una adaptación del utilizado en el apartado de la sentencia `switch`. El programa mostrará el menú hasta que no se seleccione una de las cuatro opciones. Para salir, solo hay que elegir la opción 4.

2.5.3. `for`

Esta sentencia es el bucle más flexible y utilizado en el lenguaje C. Este bucle es utilizado cuando el número de veces que debe ser repetida la sentencia o bloque del bucle está definido, o cuando las sentencias de un bucle `while` o `do while` se repitan un número indefinido de veces. Por lo tanto, cualquier sentencia `while` puede ser transformada en una sentencia `for` de forma mecánica y sin ninguna ambigüedad.

La sintaxis de la sentencia `for` es la siguiente:

```
for (inicializacion; condicion; incremento)
    sentencia_bloque;    //Se ejecuta mientras condicion es verdad
```

Analizando esta sintaxis podemos observar que:

- `inicializacion` es una expresión que normalmente inicializa el bucle.
- `condicion` es una expresión cuyo valor es un entero y determina si el bucle se ejecuta o no.
- `incremento` es una expresión que incrementa o decrementa el índice del bucle cada vez que éste se itera.
- `sentencia_bloque` es una sentencia simple, una sentencia de control condicional o iterativa, o un conjunto de sentencias (bloque).

Cualquiera de las tres partes del bucle `for` se puede omitir, aunque deben permanecer los puntos y comas (;). Si `condicion` se omite, se toma como que `condicion` siempre se verifica, por lo que en principio un bucle `for` sin `condicion` es un bucle infinito, es decir, un bucle que siempre se ejecutará, a menos que sea interrumpido por otros medios, como un `break` o un `return`.

Las expresiones de `inicializacion`, `condicion`, e `incremento` de una sentencia `for` se evalúan en el siguiente orden:

1. La expresión de `inicializacion` solo se evalúa una vez, al entrar en el `for`.
2. A continuación se evalúa `condición`.
3. Si `condicion` es verdadera, la `sentencia_bloque` del bucle se ejecuta.
4. A continuación se ejecuta la expresión de `incremento`.
5. Los pasos 2, 3 y 4 se repiten hasta que `condicion` sea falsa.

Una vez terminado el bucle, el flujo del programa continúa por las sentencias existentes a continuación del bucle `for`.

Para ver su funcionalidad observemos el siguiente ejemplo:

Ejemplo 2.8

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

int main() {
    int i, j;
    for (i = 1; i <= 10; i++)
    {
        printf("La tabla del %d es:\n",i);
        for (j = 1; j <= 10; j++)
            printf("%d x %d = %d\n", i, j, i*j);
    }
    return 0;
}
```

Explicación

En este ejemplo podemos ver como tenemos un bucle `for` dentro de otro bucle `for` que van pasando por todos los números naturales del 1 al 10 (ambos inclusive) y por pantalla se van mostrando las tablas de multiplicar de los 10 primero números naturales.

Capítulo 3

Subprogramas y modularidad

3.1. Introducción

A la hora de resolver un problema complejo, el método seguido para su resolución consiste en descomponer el problema inicial en una serie de subproblemas más simples. De esta forma, el problema puede ir resolviéndose por partes, centrándose en cada momento en uno de los subproblemas concretos a resolver.

A estas partes se les llama subprogramas y son llamados por el programa principal o por otros subprogramas. Cada subprograma debe tener un nombre o identificador. Una vez implementado, el subprograma puede ser llamado desde cualquier otro punto del programa principal por medio de una instrucción de llamada, la cual designa el nombre del subprograma correspondiente.

Los subprogramas ofrecen tres ventajas:

- **Mejoran la legibilidad del programa:** Cada subprograma, al realizar una tarea determinada y concreta, no tiene un número de líneas excesivo, por lo que es fácil de entender y programar. Además, cada subprograma puede ser desarrollado y comprobado de forma independiente uno de otro.
- **Acortan los programas:** Si el subprograma se utiliza varias veces a lo largo del programa se evita tener que escribir repetidamente las mismas instrucciones, disminuyendo así la probabilidad de introducir errores en el programa.
- **Acortan el tiempo de desarrollo:** Los subprogramas, al poder ser reutilizados en otros programas, minimizan el tiempo de corrección y desarrollo de éstos, evitando la necesidad de reprogramar y verificare tareas ya implementadas en otros programas.

Los subprogramas pueden ser de dos tipos:

- **Procedimiento:** Subprograma que realiza una determinada tarea pero no devuelve ningún valor. Los procedimientos normalmente se utilizan para estructurar un programa y para mejorar su claridad y generalidad.
- **Función:** Subprograma que realiza una determinada tarea y devuelve un valor, el cual puede ser utilizado por la sentencia que llama a la función. Las funciones se utilizan para crear operaciones y tareas nuevas no implementadas en el lenguaje de programación.

3.2. Procedimientos y funciones

Un subprograma en C, por definición, consta de una o más funciones. Todos los subprogramas en C son funciones, no existen los procedimientos como tal (lo que si existen son funciones que no devuelven ningún valor).

Normalmente una función tiene la siguiente sintaxis:

```
1 static tipo_devuelto nombre_funcion(declaracion_parametros){
2     directivas
3     declaraciones
4     sentencias;
5     return valor_devuelto;
6 }
```

Analizando esta sintaxis podemos observar que:

- `static` es una palabra clave opcional que especifica que la función es accesible únicamente dentro del fichero en el cual se ha definido (alcance de módulo). Si omitimos esta palabra clave, la función es accesible desde todos los módulos que componen el proyecto o aplicación (alcance global).
- `tipo_devuelto` declara el tipo de datos (`int`, `float`, etc) del valor que la función devolverá al ser invocada (si es que devuelve alguno). Si no especificamos el tipo, el compilador asume por defecto que el valor devuelto por la función, si lo hay, es de tipo entero (`int`). Si la función no devuelve ningún valor, se indica especificando el tipo `void`.
- `nombre_función` es el identificado que da nombre a la función. El nombre puede contener legras, dígitos y el guión bajo ('_'), pero no puede empezar por un número (el primer carácter debe ser por tanto una letra o el guión bajo).
- `declaracion_parametros` es una lista de nombres de variables separados por comas con sus tipos asociados que representa los valores que deben ser pasados a la función. los paréntesis son necesarios, aunque la función no tenga argumentos.
- Las llaves del bloque delimitan el cuerpo de la función.
- El cuerpo de la función está formado por `directivas`, `declaraciones` y `sentencias`. Ninguna de ellas es obligatoria, por lo que podemos omitir cualquiera de ellas.
- Las `sentencias` pueden ser expresiones (asignación, función) o sentencias de control (`if`, `for`, etc.).
- `return` es la palabra clave opcional, que devuelve inmediatamente el control al punto en el que se hizo la llamada a la función. Se usa para forzar una salida inmediata de la función o para devolver un valor, el indicado en la expresión `valor_devuelto` (este valor debe ser del tipo especificado en `tipo_devuelto`). Todo lo que haya en secuencia tras la sentencia `return` no se ejecuta, ya que termina la función. Una función puede tener varias sentencias `return` o ninguna.

Respecto a la lista de parámetros de una función, todos los argumentos que lo forman tienen que incluir obligatoriamente tanto el tipo como el nombre. De modo que la declaración de una función adquiere la siguiente sintaxis:

```
static tipo funcion(tipo var1, tipo var2, ..., tipo varN)
```

En cuanto a las funciones, hay que tener en cuenta las siguientes consideraciones generales:

- Las funciones tienen tiempo de vida global: existen mientras dura el programa. Las funciones ni se crean ni se destruyen mientras el programa se está ejecutando.

- Las funciones tienen alcance global: pueden ser utilizadas en cualquier parte del programa, independientemente del módulo en el que se encuentren definidos. La excepción son las funciones `static`, que sólo son visibles y accesibles en el módulo en el que están definidas (solo tienen alcance de módulo) e invisibles para el resto de módulos que componen la aplicación.
- Las funciones no pueden anidarse: una función no puede ser definida dentro de otra función. Todas las funciones se declaran y definen en un nivel externo.
- Las funciones pueden ser recursivas: una función puede llamarse a sí misma. De igual forma, una función puede llamar a cualquier función (visible) del programa.
- Las funciones no pueden ser destino de una asignación: no pueden aparecer en la parte izquierda de una sentencia de asignación.
- Todas las funciones, excepto aquellas tipo `void`, devuelven un valor. El valor devuelto es indicado explícitamente en la sentencia `return`. Mientras una función no se declara como `void` puede ser usada como operando en cualquier expresión de C. Por lo tanto una función tipo `void` no puede ser usada en una sentencia de asignación.

Para ver el funcionamiento de las funciones, veamos el siguiente ejemplo:

Ejemplo 3.1

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

int max(int a, int b);  //Protoripo de funcion

int main() {
    int x, y, maximo;
    printf("Introduce␣dos␣enteros:␣");
    scanf("%d", &x);
    scanf("%d", &y);
    maximo = max(x, y); //Llamada a la funcion max
    printf("El␣mayor␣valor␣introducido␣es␣%d", maximo);
    return 0;
}

int max(int a, int b) {
    return (a > b ? a : b);
}
```

Explicación

En la llamada a la función `max()`, el valor devuelto es asignado a la variable `maximo` que luego es impresa en pantalla en el `printf()` que le precede. En la sentencia `return` de la función `max()`, se realiza una evaluación condicional con el operador `?`.

3.2.1. Ámbito o alcance de las funciones

El ámbito o alcance de una función determina el lugar donde una función es visible y puede ser usada. Una función puede tener alcance global o alcance de módulo.

Por defecto, las funciones declaradas en un programa tienen alcance global, a menos que se declaran con la palabra clave `static` que indican al compilador que tienen alcance de módulo.

Una función de alcance global puede usarse en cualquiera de los módulos que componen la aplicación. Tan solo hay que declarar su prototipo en el resto de módulos donde se utilice.

Una función `static` (alcance de módulo) solo puede ser usada en el fichero (módulo) donde está definida y no es accesible en el resto de módulos que componen la aplicación.

3.3. Parámetros y variables locales. Variables globales

Cuando se llama a una función, puede que la función tenga que realizar cálculos internos. Por lo tanto podemos definir y utilizar variables dentro de una función con este fin. De la misma forma, si tenemos que pasarle algún tipo de información a dicha función podemos hacerlo mediante el uso de parámetros o mediante variables globales.

Las variables definidas dentro de una función son locales a ésta y no pueden utilizarse fuera de ella. A la inversa, una variable definida en la cabecera del programa es global y puede utilizarse en cualquier lugar del programa, incluyendo cualquier función o bloque.

Por lo tanto, deducimos que podemos distinguir varios tipo de variables, dependiendo del alcance, visibilidad y tiempo de vida que éstas tengan. En concreto, en C hay tres tipos de variables: variables locales, parámetros y variables globales.

3.3.1. Ámbito o alcance de las variables

El ámbito de una variable determina su accesibilidad (alcance o zona de actuación) y visibilidad en otras partes del programa. Una variable puede tener tres tipos de ámbitos:

- **Ámbito local:** Una variable de alcance local es aquella que es declarada dentro de un bloque delimitado por llaves o de una función. Únicamente es visible y puede ser usada en el interior del bloque o función en el cual está definida.
- **Ámbito de módulo:** Una variable con alcance de módulo es aquella que es declarada como `static` fuera de cualquier bloque o función. Es visible puede ser usada en el módulo en el cual está definida, pero no puede usarse en los demás módulos que componen la aplicación. De hecho en los demás módulos pueden existir variables globales que tengan el mismo nombre o identificador.
- **Ámbito global:** Una variable con alcance global es aquella que es declarada al principio de un módulo, fuera de cualquier bloque o función. Es visible y puede ser usada tanto en el módulo en el cual está declarada, como en el resto de módulos que componen la aplicación. Para ello, en cada módulo donde se quiera usar, hay que redeclararla al principio del fichero usando la palabra clave `extern`. El tipo de almacenamiento `extern` indica al compilador que la variable está definida con el mismo identificador en otro fichero fuente (módulo) del programa y que por tanto no tiene que reservar de nuevo espacio en memoria para ella.

3.3.2. Variables locales

Las variables locales son aquellas que están definidas dentro de una función o de un bloque Las variables locales no pueden usarse fuera del bloque en el que están definidas (solo pueden ser usadas por las sentencias que están dentro del mismo bloque). Además solo existen durante la ejecución del bloque de código en el que se declaran: una variable local se crea al entrar en su bloque y se destruye al salir. Por tanto, sus contenidos se pierden una vez que el flujo del programa deja el bloque.

La excepción a esta regla viene determinado por las variables locales estáticas (`static`) cuyos contenidos no se pierden una vez que el flujo del programa deja el bloque donde están (ámbito local, pero tiempo de vida global). Las variables locales de tipo `static` se utilizan cuando interesa que la variable conserve el valor entre cada llamada a la función o bloque donde están.

Dentro de un mismo programa podemos tener varias variables locales con el mismo nombre, siempre y cuando estén definidas en bloques o funciones diferentes.. Al ser locales a sus respectivos bloques no son visibles unas a otras por lo que no presentan ningún conflicto ni ambigüedad.

De la misma forma, dentro de un mismo programa podemos tener variables locales con el mismo nombre que variables globales. En ese caso, todas las referencias a ese nombre de variable dentro de la función donde se declara como local se referirá a la local y no afectará a la variable global ya que el compilador asume que la variable a la que nos referimos siempre es la local.

Observemos el siguiente ejemplo que hace uso de variables globales y locales:

Ejemplo 3.2

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

void ver();

int a= 1, b = 1, c = 1;

int main() {
    printf("A: %d B: %d C: %d\n", a, b, c);      //a, b, c globales
    ver();
    printf("A: %d B: %d C: %d\n", a, b, c);      //a, b, c globales
    return 0;
}

void ver() {
    int a = 5;                                    //a local
    c++;                                          //Incrementa c global
    printf("a: %d b: %d c: %d\n", a, b, c);      //a local, b, c globales
    int c = 50; //c local
    c++;                                          //Incrementa c local
    printf("a: %d b: %d c: %d\n", a, b, c);      //a, c local, b global
}
```

La salida generada por el programa, tras su ejecución, es la siguiente:

```
A: 1 B: 1 C: 1
a: 5 b: 1 c: 2
a: 5 b: 1 c: 51
A: 1 B: 1 C: 2
```

Explicación

En este programa se muestran claramente los dos casos comentados anteriormente:

- Podemos usar en una función una variable local con el mismo nombre que una variable global, sin ninguna ambigüedad.

- Podemos declarar variables en cualquier parte del bloque o función. Es por esto que debemos tenenr en cuenta que hasta que el lujo del programa no llegue hasta la sentencia donde se realiza la declaración, el compilador no crea realmente esa variable. Es por ello por lo que en la función `ver()`, el primer incremento de la variable `c` se refiere a la variable global, mientras que a partir del instante en el que el flujo dl programa lega hasta la sentencia donde se crea la variable `c` local, todas las referencias posteriores de la variable `c` en la función `ver()` se refieren a la variable local y no a la local.

3.3.3. Parámetros

Los parámetros son los valores que se le pasan a la función al ser llamada. los parámetros son el medio de comunicación entre la sentencia de la llama a la función y la función que es llamada.

Los parámetros funcionan dentro de la función como si de variables locales se trataran: se crean al entrar en la función y se destruyen al salir de ésta. Tienen por tanto un tiempo de vida y ámbito local.

Los parámetros que se escriben en la sentencia de llamada a la función se llaman parámetros reales o argumentos. Los que aparecen en la descripción de la función se llaman parámetros formales.

La relación entre los parámetros formales y los reales (argumentos) vienen dada por el orden en que aparecen. Los parámetros formales son sustituidos por los parámetros reales (argumentos) y el orden de sustitución es el de la llamada. Al emparejarse los parámetros reales y los formales deben coincidir en número y en tipo, aunque no es necesario que coincidan los nombres de los identificadores. Si al realizar una llamada, los parámetros reales (argumentos) pasados a la función no son del mismo tipo que los parámetros formales declarados en la función, el compilador no da´ra mensaje de error, pero se producirán resultados inesperados.

3.3.4. Variables globales

Al contrario de las variables locales, las variables globales son accesibles y visibles en todo el programa y pueden utilizarse en cualquier lugar de éste, incluyendo cualquier función. La excepción viene dada por las variables globales estáticas, las cuales solo son accesibles y visibles en el módulo en el cual están definidas.

Las variables globales mantienen sus valores durante toda la ejecución del programa (tiempo de vida global). Las variables globales se declaran fuera de cualquier función, generalmente al principio del programa. Cualquier función puede acceder a ellas y modificar su valor independientemente del lugar donde se encuentren. No es por tanto necesario pasar una variable global como parámetro a una función ya que esta puede acceder directamente a ella. Por tanto las variables globales representan una alternativa al uso de parámetros. Sin embargo, se debe evitar el abuso de las variables globales por varias razones:

- Ocupan memoria durante toda la ejecución del programa)no solo cuando se necesitan.
- El uso de variables globales puede conducir a errores de programación debido a efectos laterales (podemos accidentalmente cambiar el valor de una variables global debido a que se puede usar en cualquier función o bloque del programa).
- El uso de variables globales en lugar de parámetros hace que las funciones no sean genéricas al depender de variables que no están definidas dentro de sí misma.

3.4. Tiempo de vida

El tiempo de vida de un objeto (variables, funciones, etc.). puede ser global o local.

Tiempo de vida global significa que una vez que el objeto ha sido definido, su espacio y posición en memoria, así como sus valores, solo se conservan durante la ejecución del bloque de código en el cual el objeto está definido. Una vez que el bloque de código ha terminado de ejecutarse, la posición de memoria que ocupa se libera y el valor del objeto queda indefinido. Un objeto con tiempo de vida local solo existe mientras el flojo del programa permanece en el bloque o función en el cual está definido. Una vez que el flujo del programa pasa a otro lugar el objeto es destruido.

El tiempo de vida de un objeto viene determinado por cómo y dónde está declarado el objeto. Por lo que tenemos que tener en cuenta las siguientes reglas:

- Una función tiene tiempo de vida global: Una vez que la función está definida permanece en memoria hasta que el programa termina.
- Las variables globales tienen un tiempo de vida global. Generalmente estas variables suelen ponerse al principio del programa o incluidas en un fichero de cabecera.
- Las variables locales tienen tiempo de vida local. La excepción viene determinada por las variables locales `static`, las cuales se crean en el momento en el que el flujo del programa llega a ala función o bloque donde están definidas y no se destruyen hasta que el programa (no la función o bloque donde están) finaliza.

3.5. Paso de parámetros a las funciones

3.5.1. Parámetros por valor y por referencia

Los parámetros son los valores que se le pasan a la función al ser llamada. Los parámetros pueden ser de dos tipos: parámetros por valor o parámetros por referencia en función de si se pasa el valor del argumento o la dirección de memoria del argumento.

- Los parámetros por valor o copia son aquellos para los que los cambios en el valor del parámetro formal no afecta al del parámetro real o argumento.
- Los parámetros por variable o referencia son aquellos para los que los cambios en el valor del parámetro formal afecta al del parámetro real o argumento.

En la llamada por valor se copia el valor del argumento en el correspondiente parámetro formal de la función, por lo que, al transferirse únicamente una copia del valor del argumento, los cambios en los parámetros de la función no afectan a la variable usada como argumento en la llamada.

En la llamada por referencia, se transfiere en la llamada la dirección de memoria del argumento al parámetro. Dentro de la función, la dirección se utiliza para acceder a la variable usada como argumento en la llamada, de manera que lo que se hace con el parámetro formal afectará a la variable usada como argumento.

3.5.2. Creación de una llamada por referencia

El lenguaje C usa el método de llamada por valor para pasar argumentos. Para forzar una llamada por referencia es necesario hacer uso de los punteros. Se pueden pasar punteros a las funciones de la misma forma que se puede pasar cualquier otro tipo de dato. Para ellos, debemos declarar los correspondientes parámetros como tipo puntero.

Para observar esto, veamos el siguiente ejemplo:

Ejemplo 3.3

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

int triple(int a);        //Prototipo de funciones
void cambia(int *a);

int main() {
    int x = 5;            //Variable local
    printf("El valor de x es %d.\n", x);
    printf("El triple de x es %d.\n", triple(x));
    printf("El valor de x es %d.\n", x);
    cambia(&x);
    printf("El valor de x es %d.\n", x);
    return 0;
}

int triple(int a) {
    a = a * 3;
    return a;
}

void cambia(int *a) {
    *a = *a + 2;
}
```

Tras la ejecución del programa, el resultado mostrado en pantalla es el siguiente:

```
El valor de x es 5
El triple de x es 15
El valor de x es 5
El valor de x es 7
```

Explicación

En este ejemplo, en la llamada a la función `triple()`, no se transfiere a la función la variable x, sino una copia de dicho valor, que es 5, el cual es tomado por el parámetro `a` de la función. Este parámetro `a` es local a la función y es distinto a la variable `x`. Cuando se realiza la asignación dentro de la función, el único valor que se modifica es el de la variable `a` y no el de la variable `x` de la función `main()`. Sin embargo, cuando se realiza la llamada a la función `cambia()`, se realiza una llamada por referencia ya que se transfiere el parámetro `a` de la función a la dirección de memoria de la variable `x`, en lugar de su valor. Dentro de la función `cambia()`, a través del operador `*`, se accede a la dirección de memoria de la variable `x`, almacenada en `a`, y se incrementa en dos unidades el valor almacenado en dicha dirección de memoria. De esta forma conseguimos modificar el valor de la variable `x`. Es por esto que la última sentencia `printf()` muestra el valor 7, en lugar de 5.

3.6. Prototipos de funciones

Como comentamos anteriormente, el tipo devuelto por defecto por una función es un entero (`int`). Por lo que, cuando no es así, debemos indicar en la función explícitamente qué tipo de dato es el que devuelve dicha función.

Antes de que se pueda usar una función que devuelva un tipo no entero, se debe hacer saber su tipo al resto del programa. Si en el programa se llama a una función que devuelve un tipo no entero y esa llamada se produce antes

de la declaración de la función, el compilador supondrá que esa función llamada devuelve un tipo entero y generará erróneamente el código de la llamada a al función.

Este problema puede ser evitado definiendo la función antes de que sea llamada. Es decir, si en el programa una función llama a otra, lo que tendremos que hacer es definir antes la función que es llamada y posteriormente definir y escribir la función que llama a la anterior. Teniendo esta precaución a la hora de escribir un programa evitamos el problema anterior.

Para evitar dicho problema están los prototipos o declaraciones de funciones. Un prototipo de función básicamente informa al compilador del número de tipos de datos de los argumentos de una función así como del tipo de valor que ésta devuelve. Los prototipos de funciones consisten en la declaración del número y tipo de los argumentos de la función antes de su definición.

La declaración del prototipo de una función tiene dos propósitos principales:

- Evitar las conversiones erróneas de los valores cuando los argumentos y parámetros no son del mismo tipo.
- Evitar errores en el número y tipo de argumentos usados al llamar y/o definir una función.

Los prototipos de funciones permiten que el compilador pueda llevar a cabo una fuerte comprobación de tipos y hacer que éste realice las conversiones adecuadas (si el tipo de los argumentos y el de los parámetros no coinciden) cuando éstas sean posibles, así como informar de cualquier conversión ilegal cuando ésta no sea posible. Ademas permiten que el compilador pueda detectar si en una llamada a una función faltan o sobran argumentos, ya que gracias al prototipo, el compilador conoce exactamente el número de parámetros que espera la función.

Los prototipos van al principio del programa, debiendo aparecer antes de que se produzca alguna llamada a las funciones que declaran. La forma general de un prototipo es la siguiente:

```
tipo_devuelto nombre_funcion(tipo var1, tipo var2, ..., tipo varN);
```

Como podemos observar es exactamente como la primera línea de la definición de función, terminada en punto y coma.

El prototipo de una función que no tiene parámetros tiene la palabra clave `void` como lista de parámetros.

Para entender la importancia de los prototipos, veamos el siguiente ejemplo:

Ejemplo 3.4

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

int min(int x, int y);  //Prototipo de funcion

int main() {
    float a = 29.57, b;
    printf("Introduce un numero: ");
    scanf("%f", &b);
    printf("El numero mas pequeno es %d\n", min(a, b));
    return 0;
}

int min(int x, int y) { //Defunicion de funcion
    return (x < y ? x : y);
}
```

Explicación

Como podemos observar, cuando llamamos a la función `min()`, los argumentos que pasamos son de tipo `float`, cuando la función espera que le pasemos enteros. Gracias a que se ha especificado el prototipo de la función, el compilador realiza previamente una conversión y convierte los valores tipo `float` a `int`: el valor de `a`, 29.57, se convierte en 29 y el de `b` a su respectivo valor entero.

3.7. Macros como alternativa a las funciones

Las macros pueden representar a veces una alternativa al uso de funciones. Es más, el uso de macros en lugar de funciones puede llevar a una mayor velocidad de ejecución del código ya que no hay que perder tiempo en llamar a una función. No obstante debemos tener en cuenta que no es lo mismo una macro y una función y por lo tanto debemos conocer sus diferencias. El lenguaje C permite definir y eliminar constates y macros mediante las directivas `#define` y `#undef`.

La directiva `#define` define un identificador y una cadena asociada. El compilador, más concretamente, el preprocesador, sustituirá en el fichero fuente cualquier aparición del identificador por su cadena asociada.

3.7.1. `#define` para constantes

Cuando la directiva `#define` se usa para definir constantes, sus sintaxis es la siguiente:

```
#define NOMBRE_CONSTANTE expresion_constante
```

Analizando esta sintaxis podemos observar que:

- `NOMBRE_CONSTANTE` es un identificador (por convenio, con los caracteres en mayúsculas, aunque puede definirse en minúsculas o como se quiera).
- `expreison_constante` es cualquier expresión constante válida. No puede constener variables, aunque si constantes definidas previamente.

Para ver su uso, observemos el siguiente ejemplo:

Ejemplo 3.5

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define PI 3.1416        //No debe terminar en punto y coma
5
6  int main() {
7      float area, radio;
8      printf("Introduce el radio del circulo :");
9      scanf("%f", &radio);
10     area = 2 * PI * radio;
11     printf("El area del circulo de radio %.2f es %.2f.\n", radio, area);
12     return 0;
13 }
```

Explicación

En este ejemplo, el preprocesador sustituirá toda aparición de `PI` por el valor asignado: 3.1416 y el programa se ejecutará correctamente. Es importante observar que la sentencia `#define` no termina en punto y coma, ya que si así fuera, sustituiría el valor de `PI` por "3.1416;", lo que produciría errores.

Nota 3.1 Para eliminar las constantes usamos la directiva `#undef NOMBRE_CONSTANTE`.

3.7.2. `#define` para macros

Cuando la directiva `define` se usa para definir macros, su sintaxis es la siguiente:

```
#define nombre_macro(lista_argumentos)(expresion_macro)
```

Analizando esta sintaxis podemos observar que:

- `nombre_macro` es el identificador o nombre dado a la macro. No debe haber ningún espacio en blanco entre el nombre de la macro y el paréntesis de la lista de argumentos.
- `lista_argumentos` es una lista de argumentos separados por comas.
- `expresion_macro` es una expresión que implementa la macro haciendo uso de los identificadores de la lista de argumentos.

Para ver su uso, observemos el siguiente ejemplo:

Ejemplo 3.6

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

#define CUBO(x) (x)*(x)*(x)      //No debe terminar en punto y coma

int main() {
    int x;
    printf("Introduce un numero para elevarlo al cubo:");
    scanf("%d", &x);
    printf("El cubo de %d es %d \n", x, CUBO(x));
    return 0;
}
```

Explicación

En este ejemplo, el preprocesador sustituirá toda la aparición de `CUBO` por su expresión definida anteriormente y el programase ejecutará correctamente.

Nota 3.2 Para anular las macros usamos la directiva `#indef nombre_macro`.

3.8. Módulos: definición

Un módulo es una colección de declaraciones (constantes, tipos, funciones, etc.) definidas, en principio en un ámbito de visibilidad cerrada, es decir, ocultas a toda acción o declaración ajena al propio módulo.

Una de las ventajas que ofrece el lenguaje C es que permite la posibilidad de escribir un programa en múltiples ficheros o módulos. Es decir, no solo podemos dividir el programa en diferentes funciones cada una de las cuales resuelve un problema determinado sino que, además, las funciones que componen un programa pueden agruparse en uno o varios ficheros o módulos. De esta forma, un programa puede residir en uno o más archivos fuente (módulos), cada uno de los cuales contiene una serie de funciones y variables relacionadas.

Estas funciones existentes en un determinado módulo pueden ser visibles al resto del programa o no dependiendo si las funciones son declaradas como estáticas (`static`) o no. Las funciones estáticas solo son visibles y accesibles en el módulo en el cual están definidas. Las funciones normales (no estáticas) son visibles y accesibles en todo el programa, es decir, pueden ser usadas en cualquiera de los módulos que componen la aplicación.

De la misma forma, las variables globales existentes en un módulo pueden ser visibles y accesibles o no al resto de módulos que componen la aplicación. Las variables globales declaradas como `static` solo pueden ser usadas en el módulo en el cual están definidas. Para que una variable pueda ser usada en todos los módulos que componen una aplicación hay que declararla como global en uno de ellos y declararla como global `extern` en el resto de módulos. El modificador de almacenamiento `extern` indica al compilador que se hace referencia a una variable definida en otro módulo.

Por lo tanto, en un programa compuesto de varios módulos, aquellas funciones que sean necesarias para la implementación de un módulo pero que sean indiferentes para el resto de módulos del programa, serán declaradas estáticas (`static`) al igual que aquellas variables usadas en el módulo para la implementación de éste.

En cambio, aquellas funciones y variables que sean necesarias para el resto del programa serán declaradas globales para que puedan ser utilizadas por el reto de módulos que componen la aplicación.

Con el uso de los módulos se da un paso más en el proceso de creación de una aplicación por un grupo de personas. Cada persona o grupo de ellas puede dedicarse a implementar un módulo específico, en el que incluirán funciones relacionadas, las cuales podrán ser visibles o no al resto de módulos.

De hecho, nada impide que, por ejemplo, cada módulo sea desarrollado por un programador o grupo diferente: cuando otro grupo tuviese que utilizar varias funciones externas implementadas en otro módulo, ni siquiera tendría que conocer como realmente están implementadas, sino qué es lo que hacen. Simplemente deben especificar en su módulo el prototipo de las funciones que necesiten definidas en otro módulo para poder utilizarlas.

Los módulos pueden ser vistos como cajas negras en las que el programador conoce qué variables y qué funciones implementan esos módulos y qué es lo que hacen, pero no cómo lo hacen.

Las razones para dividir un programa en diferentes módulos son las siguientes:

- Se facilita la edición: Un archivo muy grande es difícil de editar, mientras que uno más pequeño es más manejable y fácil de mantener.
- La compilación se realiza más rápidamente: Un cambio pequeño en un programa escrito en un único módulo requiere que se recompile todo el programa entero. En cambio, en un programa dividido en múltiples módulos, solo se compilan los módulos en los que se han producido cambios, en lugar de compilarlos todos, con lo que la compilación se realiza más rápidamente.
- La programación es más estructurada: Al situar funciones y variables relacionadas en un mismo módulo aumentamos la comprensión del programa y por lo tanto disminuimos los errores de programación y hacemos el código más fácil de depurar.
- Aumentamos y promovemos la reusabilidad del software: Al colocar las funciones más usadas en ficheros separados del programa principal, nos permite poder enlazar el código objeto de este módulo al de cualquier nuevo programa, sin necesidad de tener que rediseñar, reescribir y recompilar el código, haciendo la programación más rápida y eficiente.

En el entorno integrado de desarrollo del compilador de C, los programas de múltiples módulos se llaman proyectos. Cada proyecto se asocia a un fichero de proyecto, que determina que módulos son parte del proyecto.

Un proyecto en C es por tanto, un sistema software de múltiples módulos. El proyecto normalmente comprende un fichero proyecto, uno o más ficheros de cabecera (`.h`) y do o más ficheros fuente (`.c`).

3.9. Sección de includes: Ficheros de cabecera

Muchas de las funciones incluidas en la biblioteca estándar de C trabajan con sus propios tipos de datos y estructuras. Estas estructuras y tipos, junto con los prototipos de las funciones de la biblioteca estándar están definidas en los ficheros de cabecera proporcionados por el compilador.

Como ocurre con cualquier otra función, las funciones de la biblioteca estándar de C que se utilicen en un programa deben estar identificadas por su correspondiente prototipo. Por tanto, para poder utilizar las funciones de biblioteca de C es necesario insertar sus correspondientes archivos de cabecera en nuestros programas.

Los ficheros de cabecera tienen la extensión `.h`, aunque pueden tener cualquier extensión. Estos archivos, además de contener prototipos de funciones, contienen los siguientes elementos:

1. Ciertas definiciones, macros y constantes que utilizan las funciones (`#define`).
2. Referencias externas, es decir, variables `extern` que están definidas en otros módulos.
3. Enumeraciones (`enum`) y declaraciones de estructuras (`struct`).

Los archivos de cabecera se insertan en el programa fuente con la directiva `#include` y a continuación el nombre o ruta completa del fichero. Un fichero fuente puede tener tantas directivas `#include` como necesite y se ponen al principio del fichero fuente.

Además de los ficheros de cabecera suministrados por el compilador (`stdio.h`, `stdlib.h`, etc), el usuario puede definir y crear los suyos propios con prototipos y funciones que él mismo haya implementado.

La directiva `#include` tiene dos tipos de sintaxis:

```
#include <nombre_fichero.h>

#include "nombre_fichero_con_ruta_acceso.h"
```

La diferencia entre la primera y la segunda se encuentra en el directorio de búsqueda de los archivos de cabecera.

Cuando se escribe de la primera forma el compilador busca el fichero en el directorio estándar de librerías. Pero cuando se escribe de la segunda forma, el compilador busca el fichero en la ruta de acceso especificada, si ésta se indica, o en el directorio actual y posteriormente, en caso de que no lo encuentre, en los directorios por defecto y en el directorio estándar de librerías.

El uso de ficheros de cabecera ofrece las siguientes ventajas:

- Cuando es necesaria una nueva declaración o definición en el sistema, o hay que redefinir alguna de las existentes, ésta solo debe ser añadida o modificada en el fichero de cabecera y no en los módulos que lo utilizan. Los módulos, al contener la directiva `#include` automáticamente quedan actualizados.
- Cuando un fichero objeto es utilizado por más de un proyecto de software, el fichero de cabecera contiene todas las declaraciones y definiciones necesarios para que ese módulo pueda ser incluido y utilizado en los nuevos ficheros fuente, simplificando así el desarrollo y mantenimiento de programas.

3.10. Compilación condicional

El compilador de C contiene varias directivas que permiten que se compilen o no determinadas partes del código fuente en función de una condición. La compilación condicional se usa principalmente para:

- Crear diferentes versiones del mismo programa. Mediante la compilación condicional podemos compilar ciertas líneas de código o no dependiendo de la versión particular que deseemos generar. De esta forma, podemos tener un programa en el que una versión se ejecute con un determinado hardware y otra con otro diferente.
- Depurar y corregir programas. Si sospechas que una parte del programa es la causante de ciertos errores, podemos temporalmente omitirla de la compilación para ver si verdaderamente es la causante de los errores.

En la siguiente tabla se muestra el uso de las directivas de compilación condicional:

Directiva	**Función**
`#if`	Realiza un test sobre las constantes que siguen. Si el test es verdadero (distinto de cero), el código se incluye en la compilación. Si es falso el código no se compila.
`#endif`	Termina el segmento del código especificado para la compilación condicional utilizando la directiva `#if`.
`#else`	Es opcional y se usa en combinación de la directiva `#if` y `#endif`. El códigoincluido a continuación del `#else` se compila si no se verifica la condición `#if`.
`#elif`	Es opcional y equivale a un "`#else #if`"
`#ifdef`	Comprueba si una constante ha sido definida (con la directiva `#define`). Si lo ha sido, no se compila el código que viene a continuación, en caso contrario sí se compila.
`#ifndef`	Comprueba si una constante no ha sido definida (con la directiva `#define`). Si no lo ha sido, se compila el código que viene a continuación, en caso contrario no se compila.

Tabla 3.1: Directivas de compilación condicional.

Un ejemplo muy simple es el siguiente:

Ejemplo 3.7

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #define MAX 100
4
5  int main() {
6  #if (MAX < 100)
7      printffff("Esta linea esta mal escrita.");   //Sentencia erronea
8  #else
9      printf("Esta linea esta bien escrita.");
10 #endif
11     return 0;
12 }
```

Explicación

Este programa no produce ningún error al compilar ya que, aunque en la línea 7 encontramos una sentencia mal escrita, al comparar `MAX` con 100 y no cumplirse la condición salta al bloque asociado a la directiva `#else`. Por lo que el programa escribirá por pantalla la siguiente línea:

```
Esta linea esta bien escrita.
```

Capítulo 4

Estructuras de datos complejas y asignación de memoria

4.1. Vectores estáticos

Un vector estático (o array) es una colección ordenada de variables del mismo tipo que ocupan posiciones contiguas de memoria y que se referencian utilizando un nombre común. La dirección más baja corresponde al primer elemento y la más alta al último. Para acceder a un elemento específico de un vector se usa un índice. El vector que se usa más frecuentemente es el de caracteres (cadena de caracteres).

La palabra "ordenada" no significa que los elementos sigan un determinado orden. Quiere decir que cada elemento tiene asociado un número secuencial (llamado subíndice) que identifica su posición en el vector.

4.1.1. Vectores unidimensionales

El formato general para la declaración de un vector es el siguiente:

```
tipo nombre[tamaño];
```

Analizando esta sintaxis podemos observar que:

- `tipo`: es el tipo base del vector y determina el tipo de datos de cada elemento del vector.
- `nombre`: es el identificador del vector.
- `tamaño`: es el número de elementos que componen el vector.

Veamos como se realiza la declaración de un vector:

Ejemplo 4.1

```
int float[10];
```

Explicación

Esta sentencia realiza dos operaciones al mismo tiempo:

1. Reserva espacio en memoria para almacenar 10 valores `float` (es decir, 10x4 bytes).
2. Crea un puntero llamado `notas` que apunta automáticamente al primer elemento del vector. Este puntero es una variable que contiene la dirección de memoria donde comienza el vector.

Con respecto a los vectores debemos tener en cuenta lo siguiente:

- En C todos los vectores usan el cero como índice del primer elemento. Así, en el ejemplo anterior, el primer elemento del vector es `notas[0]` y el último es `notas[9]`.
- El lenguaje C no realiza comprobación de límites en los vectores. Esto quiere decir que podemos indexar un vector por encima del número de elementos que contiene sin provocar ningún mensaje de error en tiempo de compilación o ejecución. De esta forma, si sobrepasamos el final del vector durante una operación de asignación, entonces corremos el riesgo de asignar valores a otras variables diferentes o a un trozo de código del programa o al sistema operativo. Como programadores, somos responsables de asegurarnos que los vectores sean lo suficientemente grandes para guardar los datos que necesitamos y de realizar las comprobaciones de límite cuando sea necesario.

4.1.2. Cadena o vector de caracteres

El tipo de vector más usado en C es el vector de caracteres. Es un vector que contiene ne cada celda un carácter del código ASCII. A los vectores de caracteres se les denominan cadenas de caracteres o `strings`.

En C, una cadena de caracteres se implementa mediante un vector de caracteres terminado en un nulo ("\0"), que es un cero que indica el final de la cadena. Por esta razón se debe declarar vectores de caracteres con un carácter más que la cadena que vaya a guardar.

La declaración de una cadena de caracteres se realiza de la siguiente forma:

```
char cadena[11];
```

En dicha declaración se permite guardar una cadena de hasta 10 caracteres puesto que el último es para el nulo de final de cadena ("\0").

Todas las funciones para el uso de las cadenas de caracteres se encuentran definidas en la librería `string.h`.

4.1.3. Vectores multidimensionales: Matrices

El lenguaje C también permite crear vectores con dos o más dimensiones. El formato general de una declaración de un vector multidimensional es el siguiente:

```
tipo nombre[tamaño1][tamaño2]...[tamañoN];
```

Pero el más usado es el vector bidimensional ya que es tratado como una tabla.

Ejemplo 4.2

Para declarar un vector "notas" de enteros bidimensional de tamaño 15x4 escribiremos:

```
int notas[15][4];
```

Y para acceder al 4º elemento de la undécima fila del vector notas escribiremos:

```
notas[10][3];
```

Nota 4.1 Los índices son una unidad inferior a los que hemos señalado con anterioridad debido a que el primer elemento de cada vector se encuentra en el índice "0". Recordemos que el primer elemento de un vector siempre está en la posición "0" como ya se dijo con anterioridad.

4.1.4. Inicialización de vectores

El lenguaje c permite la inicialización de los vectores en el momento de la declaración, aunque no permite la de los vectores locales (a menos que se declaren como `static`). El formato general de una inicialización de un vector es el siguiente:

```
tipo nombre[tamaño1][tamaño2]...[tamañoN] = lista_valores;
```

Donde `lista_valores` es una lista separada por comas de constantes, las cuales son del mismo tipo que las del tipo base del vector.

Veamos un ejemplo de inicialización de vectores:

Ejemplo 4.3

```
int i[5] = 0, 1, 4, 9, 16;
```

Esta declaración es equivalente a la siguiente:

```
int i[5];
i[0] = 0;
i[1] = 1;
i[2] = 4;
i[3] = 9;
i[4] = 16;
```

Los vectores de caracteres se pueden declarar como en el ejemplo anterior o de la siguiente forma abreviada:

```
char nombre[tamaño] = "cadena";
```

De esta forma, el compilador agrega automáticamente el "\0" al final de la cadena.

Por lo tanto tanto la primera como la segunda forma son equivalentes tal como veremos en el siguiente ejemplo:

Ejemplo 4.4

```
char saludo[5] = "hola";
char saludo[5] = 'h', 'o', 'l', 'a', '\0';
```

Los vectores multidimensionales se declararan de la misma forma.

Veamos un ejemplo:

Ejemplo 4.5

```
int datos[5][3] = 1, 1, 1, 2, 4, 5, 3, 9, 27, 4, 16, 64, 5, 25, 125;
```

Para facilitar la comprensión de esta inicialización podemos ponerlo así:

```
int datos[5][3] = {
1, 1, 1,
2, 4, 8,
3, 9, 27,
4, 16, 64,
5, 25, 125};
```

Explicación

En este ejemplo se inicializa la matriz "datos" con los 5 primeros números naturales, sus cuadrados y sus cubos.

El lenguaje C también permite inicializar vectores sin indicar el tamaño. En una sentencia de inicialización de vectores, si no se especifica el tamaño de los mismo, el compilador creará automáticamente un vector lo suficientemente grande como para albergar todos ls inicializadores presentados.

Además de ser menos tedioso, el método de inicialización de vectores sin tamaño evita que cometamos errores de reserva de espacio a la hora de la inicialización.

4.2. Funciones específicas para el uso de cadenas

Las funciones para trabajar con cadenas de caracteres se encuentran definidas en la librería `string.h` y son las siguientes:

Prototipo de función	Acción
`char *strcpy(char *c1, const char *c2)`	Copia la cadena `c2` sobre la cadena `c1`.
`char *strcat(char *c1, const char *c2)`	Concatena la cadena `c1` con la cadena `c2`.
`char *strchr(const char *c1, int ch1)`	Busca el carácter `ch1` en la cadena `c1`. Devuelve un puntero a la primera ocurrencia de `ch1` en `c1`. Si no lo encuentra devuelve `NULL`.
`int strcmp(const char *c1, const char *c2)`	Compara la cadena `c1` con al cadena `c2`. Devuelve un valor que es <0 si `c1` es menor que `c2`; >0 si `c1` es mayor que `c2` y == 0 si `c1` es igual a `c2`.
`int strlen(const char *c1)`	Devuelve la longitud de la cadena `c1` sin contar el terminador nulo "\0"
`char *strupr(char *c1)`	Convierte la cadena `c1` a mayúsculas.
`char *strlwr(char *c1)`	Convierte la cadena `c1` a minúsculas.
`char *strset(char *c1, int ch1)`	Rellena toda la cadena `c1` con el carácter `ch1`.
`char *strstr(const char *c1, const char *c2)`	Encuentra la primera coincidencia de la subcadena `c2` dentro de `c1`. Devuelve `NULL` si no la encuentra.

Tabla 4.1: Principales funciones de la librería string.h.

Veamos un ejemplo donde se usan algunas de dichas funciones:

Ejemplo 4.6

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

int main() {
    char nombre[30], frase[80];
    printf("Escribe␣tu␣nombre:␣");
    scanf("%s", nombre);          //Al ser una cadena no lleva &
    strcpy(frase, nombre);
    if (!strcmp(nombre, frase)) {
        printf("Las␣cadenas␣son␣iguales.\n");
    }
    printf("Mi␣nombre␣es␣%s\n", nombre);
    strcat(frase, "es␣listo");   //Concatena "es listo" con frase
    printf("%s\n", frase);
    printf("La␣cadena␣frase␣tiene␣%d␣caracteres.\n", strlen(frase));
    strcpy(nombre, "");          //Borra la cadena frase
    return 0;
}
```

4.3. Asignación estática y dinámica de memoria

Los programas pueden usar variables globales o locales. Las variables (globales y locales) se almacenan en posiciones fijas de memoria y todas las unciones pueden utilizar las variables globales. Sin embargo, las variables locales existen sólo mientras están activas las funciones en las que están declaradas.

Estas variables se definen cuando se compila el programa. El compilador reserva espacio de memoria para estas variables en tiempo de compilación (antes de comenzar la ejecución del programa) y para ello, el compilador necesita conocer cuántas y de qué tipo son las variables a asignar. Estas variables estáticas sirven para describir estructuras de datos cuya forma y tamaño se conocen de antemano, pudiéndose acceder a sus elementos de una manera estándar.

Sin embargo, no siempre es posible conocer con antelación a la ejecución cuánta memoria se debe reservar al programa. Otras veces necesitamos estructuras de datos que puedan variar de forma y tamaño durante su existencia o ejecución del programa, es decir, necesitamos que sus elementos individuales puedan crearse y conectarse dinámicamente.

Esto se consigue mediante el uso de las variables dinámicas. Estas variables se crean y se destruyen dinámicamente durante la ejecución del programa. Las variables dinámicas no tienen nombre y sólo podemos acceder a ellas mediante punteros.

Las zonas dinámicas de memoria pueden ser reservadas y liberadas durante la ejecución del programa a nuestro antojo, sin embargo, el puntero es gestionado por el sistema ya que es una variable estática más, por tanto, es muy importante no confundir puntero con variable dinámica asociada a dicho puntero.

Así pues, las variables estáticas son aquellas cuya reserva de memoria se hace en tiempo de compilación. no podemos liberar el espacio de memoria que ocupan, ni cambiar su tamaño o volver a reservarlas de nuevo en tiempo de ejecución. El sistema es el que se encarga de asignar las direcciones de memoria correspondientes, de liberarlas y reservarlas, sin que podamos influir en estas acciones mediante instrucciones de nuestro programa.

Las variables dinámicas en cambio, son variables cuya reserva de memoria se hace en tiempo de ejecución. Son variables que se crean cuando nosotros queremos y que podemos destruir a nuestro antojo.

Las diferencias entre la asignación estática y la dinámica se describen en la siguiente tabla:

Asignación estática	**Asignación dinámica**
La reserva de memoria se realiza en tiempo de compilación.	La reserva de memoria se realiza en tiempo de ejecución.
La memoria asociada a cada una de las variables es fija y no varía durante la ejecución del programa.	La memoria asociada a cada una de las variables no es fija y puede variar dinámicamente en función de sus necesidades.
Necesitamos conocer de antemano que cantidad de memoria debe necesitar la variable.	No es necesario conocer de antemano qué cantidad de memoria necesitamos ya que podemos obtener más según la vayamos necesitando.
Necesitamos conocer de antemano la forma (fija) de la estructura de datos que vamos a necesitar.	La forma de las estructuras de datos creadas puede variar durante la ejecución del programa.

Tabla 4.2: Diferencias entre asignación estática y asignación dinámica.

Es decir, en la asignación estática de memoria, el compilador reserva para cada una de las variable, un espacio de memoria igual al tipo de datos con el que ha sido declarado. Lo que se obtuvo durante la compilación es todo lo que se tiene (el área de datos estáticos de un programa se establece en el momento de la compilación y no puede cambiar durante la ejecución del mismo). Esto significa que de antemano tenemos que conocer qué tipo de datos queremos almacenar y qué cantidad de datos vamos a necesitar.

En cambio, la asignación dinámica de memoria, no se realiza en tiempo de compilación, sino en tiempo de ejecución. La ventaja que ofrece la asignación dinámica de memoria es que nos permite una mayor flexibilidad a la hora de crear estructuras de datos y además, nos permite aprovechar la memoria disponible de forma más eficiente, ya que sólo usamos aquella que necesitamos. Mediante la asignación dinámica de memoria,m el programa puede crear o destruir, en tiempo de ejecución, espacio de memoria para sus variables en función de sus necesidades.

4.3.1. Funciones de asignación dinámica en C

En C se utilizan las funciones `malloc()` y `free()` para asignar y liberar memoria. La función `malloc()` asigna memoria, inicializa el vector con la basura que haya en dicha región de memoria y devuelve un puntero al comienzo de esa memoria. Si queremos que el vector quede inicializado sin basura, usaremos la función `calloc()`, la cual también devuelve un puntero al comienzo de dicha memoria.

Cuando se termina de utilizar un bloque de memoria, se puede liberar el espacio con la función `free()`, de manera que esa memoria liberada queda disponible para poderla asignar nuevamente cuando haga falta.

Los prototipos de estas tres funciones son:

`malloc():`

`void *malloc(size_t tamaño);` Siendo `tamaño` el número de bytes requeridos.

`calloc():`

`void *calloc(size_t tamaño);` Siendo `tamaño` el número de bytes requeridos.

`free():`

```
void free(void *p);
```

Dichas funciones están definidas en la librería `alloc.h` o `stdlib.h`. Tanto la función `malloc()` como `calloc()` devuelven `NULL` si no hyay suficiente memoria para satisfacer la petición.

El argumento de la función `free()` con un puntero no válido o no inicializado puede causar daños e incluso puede causar un "cuelgue" del programa o del propio sistema operativo.

Ejemplo 4.7

```
1 char *p;
2 p = (char *) malloc(50);
```

Explicación

Este ejemplo asigna 50 bytes de memoria al vector `p`.

Ejemplo 4.8

```
1 int *p;
2 p = (int *) malloc(25 * sizeof(int));
```

Explicación

Este ejemplo reserva espacio para 25 enteros.

Como podemos observar en este último ejemplo, para asegurar la portabilidad del programa, como los tamaños de los datos pueden cambiar de un compilador a otro, podemos usar el operador de tiempo de compilación `sixeof(tipo)`, el cual devuelve el tamaño en bytes que ocupa el tipo de datos indicado como argumento.

Ejemplo 4.9

```
sizeof(int)
```

Nos devuelve cuanto ocupa un elemento de tipo `int` en nuestro compilador.

Para que la asignación de memoria sea correcta debemos comprobar el valor devuelto por las funciones `malloc()` y `calloc()`. El modo correcto de asignar memoria y comprobar que el puntero obtenio es válido es de la siguiente forma:

```
int *p;
p = (int *) malloc(100 * sizeof(int));
if (p == NULL) {
    printf("No hay memoria suficiente\n");
    /* Aqui vendria la rutina de tratamiento del error */
}
```

4.4. Vectores dinámicos

En muchas situaciones de programación, es imposible saber cómo de grande se necesita un vector, por lo que no es posible usar un vector estático de tamaño predefinido, ya que en ese caso, las dimensiones deben quedar establecidas en tiempo de compilación y no se pueden cambiar durante la ejecución del programa.

La solución está en crear vectores dinámicos. Mientras que los vectores estáticos se crean en tiempo de compilación de manera que su memoria permanece asignada durante toda la ejecución del programa, los vectores dinámicos se crean en tiempo de ejecución de manera que su memoria se asigna solo cuando se ejecuta la sentencia que define su tamaño.

En el siguiente ejemplo, podemos ver como crear y usar un vector dinámico:

Ejemplo 4.10

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

int main() {
    int n, i;
    float suma = 0, *notas;
    printf("Cuantas notas vas a introducir?");
    scanf("%d", &n);
    notas = malloc(n * sizeof(float));  //Reservamos memoria para n notas
    if (notas == NULL)  //Comprobamos que la peticion de memoria ha tenido exito
        printf("Error de asignacion de memoria");
```

```
12      else {
13          for (i = 0; i < n, i++) {
14              printf("Introduce la nota del alumno %d: ", i + 1);
15              scanf("%f", &notas[i]);
16              suma = suma + notas[i];
17          }
18          for (i = 0; i < n; i++)
19              printf("La nota del alumno %d es %.2f \n", i+1, notas[i]);
20          printf("La nota media de los %d alumnos es %.2f \n", n, suma/n);
21          free(notas);              //Liberamos la memoria asignada anteriormente
22      }
23      reutrn 0;
24  }
```

Explicación

Como podemos observar en el programa, para prevenir un uso accidental de un puntero nulo, comprobamos el punteo `notas` para asegurarnos que la petición de memoria ha tenido éxito y que `malloc()` ha devuelto un puntero válido. Además, para asegurar la portabilidad de este programa a computadoras reales de tamaño diferente, hemos usado el operador `sizeof` para calcular el número de bytes que se necesitan para un vector de n reales.

4.5. Consideraciones

4.5.1. Paso de cadenas y vectores a funciones

El lenguaje C permite usar un vector como argumento de una función. Para pasar vectores unidimensionales a funciones, en la llamada a la función se pone el nombre del vector sin índice. Esto pasa la dirección del primer elemento del vector a la función. En C no se puede pasar un vector completo como argumento a una función (debido al tiempo prohibitivo que entrañaría tener que copiar el vector por completo), en su lugar, se pasa automáticamente un puntero, es decir, se pasa el vector por referencia, así todos los cambios que hagamos al vector dentro de la función, afectarán al vector original.

Para ilustrar esto, veamos el siguiente ejemplo:

Ejemplo 4.11

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4   void ver(char c[20], int tope);
5   void cambia(char c[20], int salto);
6
7   int main() {
8       char letras[20];
9       int i, j = 10;
10      for (i = 0; i < 20; i++)
11          letras[i] = 'a' + 1;
12      ver(letras , j);
13      cambia(letras , j);
14      return 0;
15  }
16
```

```
17 void ver(char c[20], int tope){
18     int i;
19     for (i = 0; i < tope; i++)
20         printf("%c", c[i]);
21     printf("\n");
22 }
23
24 void cambia(char c[20], int salto) {
25     int i;
26     for (i = 0; i < 20; i++)
27         c[i] = c[i] + salto;
28 }
```

Al ejecutar el programa, el resultado mostrado en pantalla es el siguiente:

```
abcdefghij
fhgijklmno
```

Explicación

Si nos fijamos, tanto en la función `cambia()` como en la función `ver()` pasamos como argumento un vector y una variable. La variabla es pasada por valor, es decir, transferimos a las funciones el valor de la variable, no la variable. En cambio, el vector es pasada por referencia, es decir, tranferimos a la función no una copia del vector, sino la dirección del vector (en concreto, la dirección del primer elemento del vector).

Al ser pasado por referencia, cualquier modificación que sufra dicho vector en la función, afectará al vector argumento. De ahí se explica que en la segunda llamada a la función `ver()`, los resultados obtenidos son diferentes, ya que en la llamada a `cambia()` hemos alterado el valor de los elementos del vector.

De esta forma, cuando usamos un vector como argumento de una función, el compilador solo pasa la dirección del vector uy no una copia del vector. En C, un nombre de vector sin ningún índice, es un puntero al primer elemento del vector. Por lo tanto, a la hora de implementar la función que recibirá dicho vector como argumento, debemos declarar su correspondiente parámetro como de un tipo puntero compatible.

Hay tres formas de declarar un parámetro topo puntero:

- Declarando el parámetro como un vector indicando su tamaño.
- Declarando el parámetro como un vector sin tamaño.
- Declarando el parámetro como un puntero.

Veamos otro ejemplo del paso de vectores a funciones:

Ejemplo 4.12

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main() {
5     float nota[10];
6     int i;
7     for (i = 0; i < 10; i++) {
```

```
8           printf("Introduce la nota del alumno %d: ", i + 1);
9           scanf("%f", &nota[i]);
10      }
11      aprobados(nota);
12  }
```

Como vemos en este ejemplo, tenemos que implementar la función `aprobados` que recibe como parámetro un vector. Vamos a ver a continuación tres formas distintas de implementar esta función y de declarar que recibe un vector como parámetro:

1. Declarando el parámetro como un vector indicando su tamaño:

```
1  void aprobados(float datos[10]) {
2      int i;
3      printf("Han aprobado los siguientes alumnos:\n");
4      for (i = 0; i < 10; i++)
5          if (datos[i] >= 5)
6              printf("Alumno %d: Nota: %5.2f\n", i+1, datos[i]);
7      }
8  }
```

2. Declarando el parámetro como un vector sin tamaño:

```
1  void aprobados(float datos[]) {
2      int i;
3      printf("Han aprobado los siguientes alumnos:\n");
4      for (i = 0; i < 10; i++) {
5          if (datos[i] >= 5)
6              printf("Alumno %d: Nota: %5.2f\n", i+1, datos[i]);
7      }
8  }
```

3. Declarando el parámetro como un puntero, accediendo como si fuera un vector:

```
1  void aprobados(float *datos) {
2      int i;
3      printf("Han aprobado los siguientes alumnos:\n");
4      for (i = 0; i < 10; i++) {
5          if (datos[i] >= 5)
6              printf("Alumno %d: Nota: %5.2f\n", i+1, datos[i]);
7      }
8  }
```

Explicación

Las tres formas anteriormente vistas usan el mismo código en su interior. Aún así, la forma más común de declarar que se recibe un vector es usando un puntero (tercera forma definida anteriormente). En esta implementación, aunque hemos declarado que se recibe un puntero, luego en el código, lo hemos tratado como si fuera un vector, ya que como hemos dicho antes, el C permite indexar cualquier puntero usando [] como si el puntero fuera un vector. Recordemos que en realidad el nombre de un vector es un puntero al comienzo de éste.

4.5.2. Vectores multidimensionales dinámicos

De la misma manera que creamos vectores dinámicos, podemos crear vectores multidimensionales dinámicos (matrices dinámicas), pero es necesario usar una función para acceder a ellos, ya que debe haber una forma de definir el tamaño de todas las dimensiones, excepto la de más a la izquierda.

Para realizar esto, simplemente debemos crear una función que tenga un parámetro declarado con los límites adecuados de la matriz y pasar el puntero de la matriz dinámica a dicha función.

Para ilustrarlo, observemos el siguiente ejemplo:

Ejemplo 4.13

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

int potencia(int base, int exponente);
void crea_tabla(int t[10][3]);
void ver_tabla(int t[10][3]);

int main() {
    int *t;
    t = malloc(10 * 3 * sizeof(int));    //Reservamos memoria para 10 x 3 enteros
    if (t == NULL)
        printf("Error de asignacion de memoria");
    else{
        crea_tabla(t);                   //t es un puntero a enteros
        ver_tabla(t);
        free(t);                         //Liberamos la memoria ocupada por t
    }
    return 0;
}

void crea_tabla(int t[10][3]) {
    int i, j;
    for (i = 0; i < 10, i++) {
        for (j = 0; j < 3; j++)
            t[i][j] = potencia(o + 1, j + 1);
    }
}

void ver_tabla(int t[10][3]) {
    int i, j;
    printf("%10s %10s %10s\n", "N", "N2", "Cubo(N)");
    for (i = 0; i < 10; i++) {
        for (j = 0; j < 3, j++)
        printf("%10d", t[i][j]);
    }
    printf("\n");
}

int potencia(int base, int exponente) {
    int i, pot = 1;
    for (i = 0; i < exponente; i++)
```

```
42          pot = pot * base;
43      return pot;
44  }
```

Explicación

En este ejemplo se crea una tabla con 10 elementos, su cuadrado y su cubo. Al definir el parámetro de la función con las dimensiones deseadas de la matriz, conseguimos que el compilador de C maneje el puntero `t` como si de una matriz bidimensional se tratara (por lo que al compilador concierne, la función `crea_tabla()` y `ver_tabla()` se tiene una matriz de 10 x 3 enteros).

4.5.3. Vectores de punteros

Al igual que se crean vectores de cualquier tipo de datos, se pueden crear vectores de punteros. Así, por ejemplo, si se necesita reservar muchos punteros a varios valores diferentes, se puede declarar un vector de punteros. Un vector de punteros es un vector que contiene elementos punteros, cada uno de los cuales apunta a un tipo de datos específico.

El formato general para la declaración de un vector de punteros es:

```
tipo *nombre_vector[tamaño];
```

Para asignar una dirección de una variable a un elemento en concreto del vector de punteros, se escribirá:

```
nombre_vector[elemento] = &nombre_variable;
```

Para devolver el valor al que apunta ese elemento del vector, se escribirá:

```
*nombre_vector[elemento]
```

De la misma forma, se podría declarar un puntero a un vector de punteros de la siguiente forma:

- `(*lista)[] /* puntero a un vector */`
- `*(*lista)[] /* puntero a un vector de punteros */`
- `int *(*lista)[] /* puntero a un vector de punteros a int */`

Si se quiere pasar un vector de punteros a una función, se puede utilizar el mismo método que se utilizaba para pasar cualquier vector a una función (declarando el parámetro de la función como un vector sin índice, como un puntero, etc.).

Por ejemplo:

Ejemplo 4.14

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* Mediante vector de punteros */
5  void mostraar(int *a[]);
6
7  /* Mediante punteros a punteros */
8  void mostrar2(int **a);
9
```

```
10 int main() {
11     int *t[3], a[] = {2, 4, -1};
12     t[0] = &a[0];
13     t[1] = &a[1];
14     t[2] = &a[2];
15     mostrar(t);
16     mostrar2(t);
17     return 0;
18 }
19
20 void mostrar(int *a[]) {
21     int i;
22     for (i = 0; *a[i] > 0; i++)
23         printf("%d\n", *a[i]);
24 }
25
26 void mostrar2(int **a) {
27     for ( ; **a > 0; (*a)++)
28         printf("%d\n", **a);
29 }
```

Explicación

Como se observa en este ejemplo, se han implementado dos versiones, una declarando el parámetro como un vector de punteros sin tamaño y otra declarando el parámetro con un puntero a puntero. Recuérdese que en C, un vector de punteros es básicamente lo mismo que una colección de punteros a punteros.

4.5.4. Relación entre vectores y punteros

Los vectores y los punteros están fuertemente relacionados en C. Se pueden direccionar vectores como si fueran punteros y punteros como si fueran vectores. En esencia, el nombre de un vector es un puntero al comienzo de éste. Por lo tanto, para trabajar con los elementos de un vector se puede utilizar la notación de subíndices o la notación de punteros.

De la misma forma, si definimos un puntero del mismo tipo que un vector al que queremos que apunte, podemos asignar dicho vector al puntero y utilizar éste como si fuere un alias para reconocerlo y manipularlo.

Así pues, el lenguaje C permite dos métodos de acceso a los elementos de un vector: mediante indexación del vector o mediante punteros.

Véase el siguiente ejemplo:

Ejemplo 4.15

```
1 /* Indexar cadena como un vector */
2 char *strupr(char * cadena) {
3     int i;
4     for (i = 0; i < strlen(cadena); i++)
5         candea[i] = toupper(cadena[i]);
6     return cadena;
7 }
8
9 /* Recorrer cadena como puntero */
```

```
10 char *strupr(char *cadena) {
11     char *p;
12     p = cadena;
13     /* En C todas las cadenas terminan en nulo. Asi, *cadena
14     sera nulo cuando se llegue al final de la cadena */
15     while (*cadena != '\0'){
16         *cadena = toupper(*cadena);
17         cadena++;
18     }
19     return p;
20 }
```

Explicación

En este ejemplo vemos la implementación de la función de la biblioteca estándar `strupr()`, que permite convertir a mayúsculas la cadena pasada como parámetro, en el que se ilustran dos métodos de acceso (uno que indexa un vector y otro con punteros). Ambas versiones son totalmente equivalentes. Se debe hacer so de aquella que resulte más fácil de entender, aunque la versión con punteros es mucho más eficiente y rápida. Pero no por ello debemos usar siempre una versión con punteros, sino usar aquella que resulte más cómoda para el programador.

Si se quiere acceder al vector estrictamente en oren ascendente o descendente, entonces los punteros son más fáciles de usar y más rápidos. Sin embargo, si se quiere acceder al vector aleatoriamente, entonces es mejore la indexación ya que es más fácil de programar y entender.

4.6. Estructuras

Los vectores pueden definirse como una colección ordenada de variables del mismo tipo, las cuales se referencian a través de un nombre común. Con ello, se observa la imposibilidad de manejar en un único vector grupos de elementos con tipos diferentes de datos cada uno.

La solución a este problema es utilizar una estructura. Una estructura es una colección de variables o elementos denominados miembros, cada uno de los cuales puede ser de un tipo de dato diferente, que se referencian bajo el mismo nombre. La estructura se define por la palabra calve `struct`. Al igual que el identificador de un vector, el identificador de una estructura representa a todos sus componentes, pero a diferencia de un vector, sus miembros no necesitan ser del mismo tipo.

Las estructuras ayudan a organizar datos debido a que permite que un grupo de variables relacionadas se les trate como una unidad en lugar de como entidades separadas. Generalmente todos los miembros de una estructura están relacionados lógicamente unos con otros.

El formato general para la declaración de una estructura es el siguiente:

```
1 struct nombre_estructura {
2     tipo1 campo1;
3     ...
4     tipoN campoN;
5 };
```

Analizando esta sintaxis podemos observar que:

- `nombre_estructura`: Es el nombre con el que se identifica la estructura creada.
- `campo1...campoN`: Son los diferentes campos o variables contenidas en la estructura.

- `tipo1...tipoN`: Son los tipos de datos de cada una de las variables o campos de la estructura.

Cuando se declara una estructura, se está definiendo un tipo complejo de variables compuesto por los elementos de la estructura.

Por lo tanto, debemos tener en cuenta lo siguiente:

- Las variables declaradas dentro de una estructura (miembros) pueden tener el mismo nombre que una variable ordinaria o que otro miembro de otra estructura diferente sin que ello suponga un conflicto, puesto que siempre se puede distinguir por el contexto.
- Una declaración `struct` define un tipo. Una declaración de estructura que no está seguida por una lista de variables no reserva un espacio de almacenamiento sino que describe una plantilla o la forma de una estructura. Una vez definida la estructura con un nombre, el nombre puede ser usado posteriormente para crear variables de ese tipo.
- Una estructura se puede inicializar con una lista de inicializadores constantes, al igual que los vectores.
- Las estructuras se pueden copiar y asignar como unidad (enteras), pasar a funciones como argumento y ser devueltas por éstas. También podemos copiar, asignar y pasar a funciones elementos individuales de la estructura.
- Las estructuras no se pueden comparar. Si quisiéramos saber si dos estructuras son iénticas o no, la única solución sería comparar uno a uno cada uno de sus miembros.
- Las estructuras pueden anidarse, es decir, una estructura puede contener como campo miembro otra estructura (estructura anidada).

Para ilustrar esto, veamos el siguiente ejemplo:

Ejemplo 4.16

```
1  /* Definicion de la estructura direccion */
2  struct direccion {
3      char calle[30];
4      char ciudad[20];
5      char provincia[20];
6  };
7
8  /* Definicion de la estructura datos_persona */
9  struct datos_persona {
10     char nombre[35];
11     struct direccion dir;
12     int edad;
13 };
14
15 /* Declaracion de variables de tipo datos_persona */
16 struct datos_persona persona1, persona2;
```

Explicación

En este ejemplo hemos representado la información de una persona mediante una estructura que contiene una estructura anidada para su dirección.

Para hacer referencia a un miembro de la estructura se usa la siguiente sintaxis:

```
nombre_estructura.nombre_miembro
```

Es decir, el nombre de la variable de estructura seguida del operador punto '.' y del nombre del elemento, referencia a ese elemento individual de la estructura.

4.6.1. Vectores de estructuras

Al igual que podemos declarar vectores de cualquier tipo de datos, podemos declarar vectores de estructuras. Implemente debemos definir primero la estructura y luego declarar una variable vector de dicho tipo, como ilustra el siguiente ejemplo:

Ejemplo 4.17

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

/* Definicion de la estructura direccion */
struct direccion {
    char calle[30];
    char ciudad[20];
};

/* Definicion de la estructura datos_persona */
struct datos_persona {
    char nombre[35];
    struct direccion dir;
    int edad;
};

int main() {
    int i, n;
    char tecla;
    /* Declaracion de un vector de tipo datos_persona */
    struct datos_persona personas[5];
    for (i = 0; i < 5; i++) {
        printf("Nombre: ");
        scanf("%s", personas[i].nombre);
        printf("Calle :");
        scanf("%s", personas[i].dir.calle);
        printf("Ciudad: ");
        scanf("%s", personas[i].dir.ciudad);
        printf("Edad :");
        scanf("%d", personas[i].edad);
    }
    do {
        do {
            system("cls");
            printf("Introduzca el numero de la persona a visualizar: ");
            scanf("%d", &n);
        } while (n < 0 || n > 4);
        printf("Nombre: %s\n", personas[n].nombre);
        printf("Calle: %s\n", personas[n].dir.calle);
```

```
40          printf("Ciudad: %s\n", personas[n].dir.ciudad);
41          printf("Edad: %d\n", personas[n].edad);
42          printf("Desea ver los datos de otra persona(s/n)?");
43          tecla = getch();
44      } while (tecla == 's');
45      return 0;
46  }
```

Explicación

En este ejemplo podemos ver como el vector `personas` contiene 5 estructuras del tipo `datos_persona` que se completan en el bucle `for` con los datos introducidos por el usuario y luego se imprimen en el doble bucle `do while`.

4.6.2. Paso de estructuras a funciones

El lenguaje C permite pasar estructuras completas a funciones. Para ello, debemos declarar el parámetro de la función del mimo tipo que la estructura que se pasa. La mejor forma de hacer esto es definiendo la estructura globalmente y luego usar su nombre para declarar variables y parámetros de esa estructura.

Cuando se pasa una estructura a una función, se pasa por valor, es decir, lo que se pasa a la función es una copia de la estructura, no la estructura original, con lo que los cambios realizados en los contenidos de la estructura dentro de la función no afectan para nada a la estructura original utilizada como argumento. Si queremos que los cambios realizados en los contenidos de la estructura en una función afecten a la estructura original, habrá que pasarlo explícitamente por referencia (usando punteros), pasando su dirección de memoria mediante el operador &, al igual que sucede con las variables de los tipos básicos.

Ahora bien, silo que se pasa es un vector de estructuras a una función, lo que se pasa es un puntero al primer elemento del vector, con lo que los cambios efectuados dentro de la función afectan al vector original usado como argumento en la llamada.

Para ilustrar todo esto, veamos el ejemplo anterior implementado mediante funciones:

Ejemplo 4.18

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  /* Definicion de la estructura direccion */
5  struct direccion {
6      char calle[30];
7      char ciudad[20];
8  };
9
10 /* Definicion de la estructura datos_persona */
11 struct datos_persona {
12     char nombre[35];
13     struct direccion dir;
14     int edad;
15 };
16
17 /* Prototipo de funciones */
18 void cargar_datos(struct datos_persona p[]);
19 void ver_datos(struct datos_persona p);
```

```
20
21 int main() {
22     int i, n;
23     char tecla;
24     /* Declaracion de un vector de tipo datos_persona */
25     struct datos_persona personas[5];
26     cargar_datos(personas);
27     do {
28         do {
29             system("cls");
30             printf("Introduzca␣el␣numero␣de␣la␣persona␣a␣visualizar:␣");
31             scanf("%d", &n);
32         } while (n < 0 || n > 4);
33         ver_datos(personas[n]);
34         printf("Desea␣ver␣los␣datos␣de␣otra␣persona(s/n)?");
35         tecla = getch();
36     } while (tecla == 's');
37     return 0;
38 }
39
40 void cargar_datos(struct datos_persona p[]) {
41     int i;
42     for (i = 0; i < 5; i++) {
43         printf("Nombre:␣");
44         scanf("%s", p[i].nombre);
45         printf("Calle␣:");
46         scanf("%s", p[i].dir.calle);
47         printf("Ciudad:␣");
48         scanf("%s", p[i].dir.ciudad);
49         printf("Edad␣:");
50         scanf("%d", p[i].edad);
51     }
52 }
53
54 void ver_datos(struct datos_persona p) {
55     printf("Nombre:␣%s\n", p.nombre);
56     printf("Calle:␣%s\n", p.dir.calle);
57     printf("Ciudad:␣%s\n", p.dir.ciudad);
58     printf("Edad:␣%d\n", p.edad);
59 }
```

Explicación

Este ejemplo hace lo mismo que el anterior pero para la carga de datos y la presentación de los mismos utiliza dos funciones en comparación de cómo lo hacía el ejemplo anterior.

4.6.3. Paso de elementos de estructuras a funciones

Además de pasar una estructura completa a una función, podemos pasar solo un elemento de la estructura. Al igual que antes, al pasar un elemento de una estructura a una fucion solo pasamos una copia de ésta, con lo que los cambios efectuados en la función no afectan para nada al elemento de la estructura usado como argumento.

Si consideramos el ejemplo anterior, las siguientes llamadas solo pasan un elemento determinado de la estructura:

```
funcion1(personas[2].nombre);    /* Pasa solo el campo nombre */
funcion2(personas[2].edad);      /* Pasa solo el campo edad */
funcion3(personas[2].dir);       /* Pasa solo el campo dir que es otra estructura */
funcion4(personas[2].dir.calle);/* Pasa solo el campo calle */
```

Los prototipos de estas funciones serán los siguientes:

```
void funcion1(char *cadena); // O void funcion1(char cadena[]);
void funcion2(int n);
void funcion3(struct direccion d);
void funcion4(char cadena[]);// O void funcion4(char *cadena);
```

Ahora bien, si lo que queremos es pasar un elemento por referencia, para que los cambios efectuados en la función afecten al elemento pasado como argumento, debemos pasar la dirección de memoria de dicho elemento:

```
funcion1(personas[2].nombre);    /* Pasa solo el campo nombre */
funcion2(&personas[2].edad);     /* Pasa solo el campo edad */
funcion3(&personas[2].dir);      /* Pasa solo el campo dir que es otra estructura */
funcion4(personas[2].dir.calle);/* Pasa solo el campo calle */
```

Los prototipos de estas funciones serán los siguientes:

```
void funcion1(char *cadena);  // O void funcion1(char cadena[]);
void funcion2(int *n);
void funcion3(struct direccion *d);
void funcion4(char cadena[]); // O void funcion4(char *cadena);
```

Podemos observar como el operador & debe preceder al nombre de la estructura en la llamada a la función. Además, vemos cómo cuando uno de los elementos de la estructura que pasamos es una cadena de caracteres, no hace falta utilizar el operador & ya que los vectores se pasan siempre por referencia y no por valor.

4.6.4. Punteros a estructuras

De la misma forma que podemos usar punteros a cualquier tipo de variable, podemos usar punteros a variables de estructuras. Al igual que los demás punteros, lo punteros a estructuras s e declaran anteponiendo un asterisco '*' delante del nombre de la variable de estructura.

La siguiente declaración declara un puntero a una estructura de tipo `strct datos_persona`:

```
struct datos_persona *p;
```

La dirección de una variable de estructura, se obtiene, al igual que para cualquier otro tipo de variable, anteponiendo el operador & delante del nombre de la variable. Por ejemplo:

Ejemplo 4.19

```
strict datos_persona persona, *p;
p = &persona;
```

Explicación

En este ejemplo se coloca la dirección de la estructura `persona` en el puntero `p`.

Cuando se usan punteros a estructuras no se debe usar el operador punto '.' para acceder a un elemento de la estructura a través del puntero. En su lugar se debe usar el operador '->'. Por ejemplo:

Ejemplo 4.20

```
p->nombre;        /* Hace referencia al campo nombre */
p->edad;          /* Hace referencia al campo edad */
p->dir.calle;     /* Hace referencia al campo calle */
```

Para ilustrar todo esto, veamos otro ejemplo con una versión del programa anterior implementado con punteros a estructuras:

Ejemplo 4.21

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

/* Definicion de la estructura direccion */
struct direccion {
    char calle[30];
    char ciudad[20];
};

/* Definicion de la estructura datos_persona */
struct datos_persona {
    char nombre[35];
    struct direccion dir;
    int edad;
};

/* Prototipo de funcion */
void cargar_datos(struct datos_persona *p);

int main() {
    int i;
    /* Declaracion de un vector de tipo datos_persona */
    struct datos_persona personas[5];
    for (i = 0; i < 5; i++)
        cargar_datos(&personas[i]);
    return 0;
}

void cargar_datos(struct datos_persona *p) {
    printf("Nombre: ");
    scanf("%s", p->nombre);
    printf("Calle :");
    scanf("%s", p->dir.calle);
    printf("Ciudad: ");
    scanf("%s", p->dir.ciudad);
    printf("Edad :");
    scanf("%d", p->edad);
}
```

Explicación

Este ejemplo hace lo mismo que los otros pero está implementado con punteros a estructuras por lo que se hace obligatorio el uso del operador '->' para acceder a un elemento de la estructura a través de un puntero.

4.7. typedef

La palabra clave `typedef` permite definir explícitamente un nuevo nombre de tipo. Realmente no se crea un nuevo tipo sino que se define un "alias" para un tipo existente. Esto permite usar nombres más descriptivos para los tipos de datos que usamos y para las estructuras que hallamos creado.

La sintaxis de la sentencia `typedef` es la siguiente:

```
typedef tipo alias;
```

Analizando esta sintaxis podemos observar que:

- `tipo`: Es cualquier tipo de datos existente en C o la definición de una estructura.
- `alias`: Es el nuevo nombre para ese tipo.

Veamos un ejemplo:

Ejemplo 4.22

```
1  /* Definicion de la estructura direccion */
2  typedef struct {
3      char calle[30];
4      char ciudad[20];
5  } direccion;
6
7  /* Definicion de la estructura datos_persona */
8  typedef struct {
9      char nombre[35];
10     direccion dir;
11     int edad;
12 } datos_persona;
13
14 /* Declaracion de un vector de tipo datos_persona */
15 datos_persona personas[5];
```

Explicación

Estas estructuras son las mismas que las usadas en los ejemplos anteriores con la diferencia de que en ellas se ha usado `typedef` para la definición de dichas estructuras, que es el uso más habitual de dicha palabra clave.

Capítulo 5

Ficheros

5.1. Introducción

5.1.1. La necesidad de archivar la información

El sentido que utilizamos en este capítulo de fichero es el de fichero en disco o archivo.

En estos apuntes, hasta ahora se ha visto que los programas almacenan y tratan información, pero solo de modo interno, en lo que se llama almacenamiento principal o primario. El almacenamiento primario generalmente está implantado físicamente en la memoria RAM del ordenador. El almacenamiento interno tiene las siguientes características:

- Cuando el programa deja de ejecutarse, sus datos internos desaparecen por completo. La información que fue introducida desaparece y cuando se vuelva a ejecutar el programa deberán introducirse de nuevo los datos.
- Es de muy rápido acceso. La información puede ser accedida de un modo muy rápido para su consulta o tratamiento.

El nuevo modo de almacenamiento que estudiamos en este capítulo es el del almacenamiento secundario o externo. EL almacenamiento en archivos, implantado generalmente en disco, tiene las siguientes características:

- Es permanente. Una vez terminada la ejecución de un programa, los datos almacenados en disco en forma de archivos, permanecen. Los datos introducidos en una ejecución anterior del programa pueden almacenarse en arcchivos para la próxima ejecucion, incluso apagando el ordenador.
- Su acceso es mucho más lento que el acceso a los datos en la memoria RAM del ordenador.
- La capacidad del almacenamiento secundario es mucho mayor que la del almacenamiento primario.

Los dispositivos del ordenador que físicamente acogen el almacenamiento primario suele ser, principalmente la memoria RAM. Los dispositivos que físicamente acogen el almacenamiento secundario suelen ser soportes magnéticos y ópticos, es decir discos o cintas.

5.1.2. Definiciones de archivos

Un fichero o archivo podría definirse como un elemento de almacenamiento de datos sobre un medio permanente que presenta las siguientes características:

- Es una estructura de datos dinámica, en tanto que su tamaño puede cambiar durante la ejecución del programa.
- Permite el almacenamiento permanente de la información.
- No tiene un tamaño fijo preestablecido ni un máximo, salvo la propia capacidad de todo el soporte físico (generalmente el disco).
- Tiene un acceso lento a la información, si lo comparamos con los accesos a memoria principal.

En la mayor parte de los lenguajes de programación, el concepto de archivo o fichero se define como un conjunto de datos estructurados en una colección e entidades elementales o básicas del mismo tipo base de datos. En el lenguaje C, la visión de los ficheros o archivos se verá como una colección de caracteres (ficheros de texto) o de bytes (ficheros binarios).

5.2. Tipos de archivos

5.2.1. En cuanto al método de acceso

El modo de acceso a los archivos depende principalmente del soporte empleado para los mismos y del modo físico en el que se ha organizado su información.

Básicamente, los dos tipos de acceso que pueden darse son:

- Acceso secuencial: Se accederá a cada elemento del archivo uno tras otro en el mismo orden en el que se situaron.
- Acceso directo: Permite acceder a un elemento determinado sin tener que acceder previamente a otros precedentes.

5.2.2. En cuanto al tipo que almacenan

Frecuentemente, los lenguajes de programación recurren a especializaciones, matices o peculiaridades en cuanto a la forma de utilizar archivos en función de los tipos de elementos que va a almacenar. La clasificación en este sentido sería:

- Archivos de datos: Almacenan un tipo de datos. Este tipo de datos puede ser simple o estructurado, así un archivo de datos pude ser pro ejemplo de enteros, reales, caracteres, bytes, estructuras, etc. Un archivo no puede ter un tipo dinámico sino estático.
- Archivos binarios: No tienen tipo. Son de acceso muy rápido y sobre ellos es posible implementar los archivos de datos. Algunos lenguajes de programación solo disponen de este tipo de archivos.

5.3. Operaciones con ficheros

5.3.1. Declaración de ficheros lógicos

La declaración de un fichero consiste en dar un identificador y decir que el tipo de sus datos es puntero a `FILE` `(FILE *)`. Para que un programa pueda leer o escribir en un fichero, necesita utilizar puntero a fichero. El puntero identifica el archivo en disco y utiliza la secuencia asociada a él mediante las funciones que emplean buffer. Por lo tanto, la declaración de un fichero sería como sigue:

```
FILE * identificador;
```

Un puntero a un fichero es un puntero a una información que define varios aspectos sobre el fichero como son:

- Nombre.
- Estado.
- Posición actual del localizador

El tipo de datos `FILE` viene definido en la librería `stdio.h`.

En esencia, el puntero al archivo identifica un archivo de disco específico, siempre que se produzca la asociación de fichero lógico con fichero físico. El puntero al fichero no debe ser alterado nunca por el código, pues es usado por la mayoría de las unciones del sistema de archivos.

5.3.2. Asignación y apertura de ficheros

El identificador utilizado para la declaración del fichero es el nombre con el que nos referimos al fichero en nuestro programa. Sin embargo, hay que asociar o asignar a dicho identificador lógico el nombre físico que tiene en el dispositivo.

Por otro lado, la operación de apertura consiste en abrir el fichero para que éste pueda ser utilizado. En la apertura de un fichero hay que indicar si se abre para:

- Lectura: Extraer información del archivos. No altera el contenido previo del fichero.
- Escritura: Escribir el fichero desde el principio. Si el fichero no se encontrase vacío (fue utilizado con anterioridad), una operación de apertura para escritura borraría automáticamente la información anterior. Si el fichero no existe, lo crea.
- Lectura y escritura: Se van a realizar operaciones de ambos tipos. Como veremos, puede darse el caso tanto que el fichero físico aya exista como que no.
- Añadir al final: Para añadir información al final del archivo. Si no existe el archivo, se creará.

Además de identificar qué tipo de operaciones vamos a realizar con el fichero que estamos abriendo, debemos indicar si abrimos el fichero en modo texto o en modo binario.

Si se abre en modo texto, las secuencias retorno de carro/salto de línea se convierten a caracteres de salto de línea en la lectura. En la escritura, ocurre lo contrario: los caracteres de salto de línea se convierten a retorno de carro/salto de línea. En los archivos binarios no tienen lugar estas conversiones.

La función de apertura tiene el siguiente prototipo:

```
FILE *fopen(const char *nombre_fichero, cont char *modo);
```

Donde `nombre_fichero` es una cadena de caracteres que representa el nombre del fichero físico que deseamos abrir, pudiendo llevar una especificación de encaminamiento (ruta de directorio).

Si se produce algún error en la apertura (disco protegido contra escritura, lleno, nombre de fichero incorrecto, etc) la función devolverá nulo (`NULL`).

En cuanto al `modo` de apertura, un fichero solo puede ser usado para aquel modo de operación para el que se abrió y no para otro. Nos vamos a centrar en hacer uso de los que se expresan en la siguiente tabla:

Modo	Significado
"r"	Abre un archivo de texto para lectura.
"w"	Crea un archivo de texto para escritura.
"a"	Abre un archivo de texto para añadir.
"r+"	Abre un archivo de texto para lectura/escritura.
"w+"	Crea un archivo de texto para lectura/escritura.
"a+"	Abre o crea un archivo de texto para lectura/escritura.
"rb"	Abre un archivo binario para lectura.
"wb"	Crea un archivo binario para escritura.
"ab"	Abre un archivo binario para añadir.
"r+b"	Abre un archivo binario para lectura/escritura.
"w+b"	Crea un archivo binario para lectura/escritura.
"a+b"	Abre o crea un archivo binario para lectura/escritura.

Tabla 5.1: Modos de apertura.

La forma adecuada de declarar un fichero de texto es la siguiente:

```
1 FILE *f;
2
3 if ((f = fopen("datos.txt", "w")) == NULL)
4     printf("No se puede arbir el fichero.\n");
5 else {
6 /* Resto de codigo */
7 }
```

5.3.3. Cierre de ficheros

El cierre de un fichero debe realizarse siempre que se deje de utilizar un fichero (bien porque ya no lo requiere el programa o porque termina la ejecución del mismo). Su sintaxis es la siguiente:

```
int flcose(FILE *f);
```

Siendo `f` el fichero lógico a cerrar. La función `fclose()` devuelve 0 si no hay error y `EOF` si se produce algún error (por ejemplo, haber retirado el disquete, haber perdido la conexión de red, etc).

En las operaciones de escritura, las mismas no se llevan acabo de forma automática, sino hasta que se llena un sector o se indica la operación de cierre del mismo. Cuando un programa finaliza, los ficheros abiertos son cerrados automáticamente, aún cuando no se haya ejecutado instrucción de cierre para los mismos, pero existen casos en los que si no se realiza una operación de cierre específica pude llevar a la pérdida de información. Éstas y otras circunstancias remarcan la necesidad de cerrar expresamente el fichero después de su uso.

Si un fichero ha sido abierto en modo escritura y posteriormente precisa ser leído por el propio programa, lo correcto es realizar antes de la apertura en modo lectura, un cierre del mismo.

Los ficheros tienen al final de los mismos una marca que indica que ya no hay más elementos tras ése. Esa marca es `EOF` (End Of File). Ese valor de `EOF` es un valor entero, que en ficheros binarios podría confundirse con algún valor válido del fichero. Por ello, para no entrar en una casuística distinta dependiendo de si estamos manejando ficheros de texto o binarios, para comprobar el estado de fin de un fichero (si se ha llegado o no al `EOF`) nos apoyaremos siempre en la función `feof()`, cuyo prototipo es el siguiente:

```
int feof(FILE *f);
```

Devuelve, aplicado a un fichero `f`, un valor distinto de 0 si se ha alcanzado el final de dicho fichero o 0 en caso contrario.

Cuando un fichero se cierra se añade automáticamente la marca de `EOF` a la posición siguiente a la última útil. Las operaciones de apertura para añadir también reubican adecuadamente la marca `EOF`.

5.3.4. Lectura

Las operaciones de lectura consisten en la copia de la información contenida en un elemento del fichero sobre una variable del programa localizada en memoria principal. Destacamos las siguientes funciones:

- **`getc()`:** Lee un carácter de un fichero abierto. Su prototipo es:

  ```
  int getc(FILE *f);
  ```

  Siendo `f` el fichero previamente abierto con `fopen()` en modo compatible con la operación de lectura. Cada vez que empleemos la instrucción `getc()` leeremos un carácter o bien `EOF` si llegamos al final del fichero.

- **`fgets()`:** Su prototipo es:

  ```
  char *fgets(char *cad, int n, FILE *f);
  ```

  Lee del fichero `f` los `n-1` caracteres almacenándolos en la cadena `cad`, añadiendo el carácter de fin de cadena "\0". Si se encuentra un salto de línea antes de llegar a la longitud de `n-1` caracteres indicada, se detiene la lectura (se lee en este caso hasta el fin de la línea).

- **`fcanf()`:** Su prototipo es:

  ```
  int fscanf(FILE *f, const char *formato, lista-arg);
  ```

  La función `fscanf()` funciona de forma análoga a la función `scanf()`, descrita con anterioridad en el capítulo 1 de estos apuntes. Como única diferencia tiene que la lectura de la información no se realiza desde `stdin` (teclado), sino de la información existente en el fichero `f`. `lista_arg` es la lista de argumentos.

  La función devuelve el número de argumentos realmente leídos (a los que se les asigna valores), pudiendo devolver el valor `EOF` si se trata de leer más allá del final del archivo.

- **`fread()`:** Su prototipo es:

  ```
  size_t fread(void *destino, size_t tam_elem, size_t numelem, FILE *f);
  ```

  Donde `destino` es un puntero a la zona de memoria donde escribir los datos leídos del fichero `f`. La forma cómo se indica qué cantidad de bytes se desea leer se expresa a través de los parámetres `tam_elem` (número de bytes que ocupa el elemento que deseamos leer) y de `numelem` (número de elementos que deseamos leer conjuntamente a través de esta operación de lectura).

  La función devuelve el número de elementos leídos, que puede ser menor que `numelem` si se produce un error o e llega al final del fichero.

5.3.5. Escritura

Las operaciones de escritura consisten en la copia de la información contenida en una variable del programa sobre un elemento del fichero. Destacamos las siguientes instrucciones:

- **`putc()`:** Escribe caracteres en un fichero abierto con `fopen()` en modo compatible con escritura. Su prototipo es:

  ```
  int putc(int c, FILE *f);
  ```

  Siendo `f` el fichero y `c` el carácter que queremos escribir. La función `putc()` devuelve `EOF` si se produce un error en la escritura.

- **`fputs()`:** Su prototipo es:

  ```
  int fputs(const char *cad, FILE *f);
  ```

  Esta función sirve para escribir la cadena de caracteres `cad` en el fichero `f`.

- **`fprintf()`:** Su prototipo es:

  ```
  int fprintf(FILE *f, const char *formato, lista_arg)
  ```

  Esta función permite escribir en el fichero `f`, los valores de los argumentos `lista_arg` según el formato especificado en `formato`.

  La función `fprintf()` es similara la función `printf()` estudiada en el capítulo 1, excepto que la salida, en vez de producirse hacia `stdout` (pantalla) se produce hacia el fichero `f`. Por ello, los posibles formatos y uso de la instrucción son idénticos a los expuestos para `printf()`.

- **`fwrite()`:** Su prototipo es:

  ```
  size_t fwrite(const void *origen, size_t tam_el, size_t num_el, FILE *f);
  ```

  Escribe en el fichero `f` la información contenida en la zona de memoria apuntada por `origen`. El número de bytes a escribir en el fichero se obtiene conjuntamente del tamaño indicado en `tam_el` (tamaño del elemento a escribir) y del número de elementos indicados en `num_el`. La función devuelve el número de elementos escritos, que puede ser menor que `num_el` si se produce un error.

Veamos un ejemplo para ilustrar algunas de las funciones comentadas:

Ejemplo 5.1

```
1  #include <stido.h>
2  #include <stdlib.h>
3
4  int main() {
5      FILE *f;
6      FILE *f2;
7      char nombre[30];
8      char c;
9
10     printf("Introduzca␣el␣nombre␣del␣fichero␣a␣copiar:␣");
11     gets(nombre);
12     if ((f = fopen(nombre "r")) == NULL)
13         printf("No␣se␣puede␣abir␣el␣fichero.\n");
14     else {
15         f2 = fopen("Copia.txt", "w");
16         c = getc(f);
17         while (!feof(f)){
```

```
18              putc(c, f2);
19              c = getc(f);
20          }
21          fclose(f);
22          fclose(f2);
23      }
24      return 0;
25  }
```

Explicación

Este ejemplo copia de un fichero a otro carácter a carácter.

5.3.6. Acceso directo

El acceso directo expresa la posibilidad de acceder a un elemento determinado sin tener que acceder previamente a otros precedentes. Este caso se puede realizar a través de la función `fseek()`, dado que dicha función permite colocar el indicador de posición del fichero en la posición que se le indique, siempre que la misma sea correcta. El acceso directo se empleará preferentemente en ficheros binarios, pues existen casos en los que dicha operación no funciona correctamente con ficheros de texto, en el proceso de traducción de caracteres, llegando a producir errores en la localización o posicionamiento. Su prototipo es el siguiente:

```
int fseek(FILE *f, long num_bytes, int origen);
```

Analizando esta sintaxis podemos observar que:

- `f` es el fichero que tengamos abierto usando `fopen()`.
- `num_bytes` es el número de bytes que queremos desplazarnos desde la posición indicada por `origen`, siendo `origen` uno de estos posibles valores:
  - `SEEK_SET` (o valor 0): Indica el principio del fichero.
  - `SEEK_CUR` (o valor 1): Indica la posición actual del fichero.
  - `SEEK_END` (o valor 2): Indica la posición desde el final del fichero.

Veamos a continuación un ejemplo con cada una de dichos valores:

Ejemplo 5.2

```
fseek(f, 250, SEEK_SET);
```

Explicación

Se sitúa en la posición (byte) 250 desde el comienzo del fichero.

Ejemplo 5.3

```
fseek(f, 250, SEEK_CUR);
```

Explicación

Se sitúa en la posición (byte) 250 desde la posición que ocupa actualmente en el fichero.

Ejemplo 5.4

```
fseek(f, 250, SEEK_END);
```

Explicación

Se sitúa en la posición (byte) 250 a partir del final del fichero.

La función `fseek()` devuelve 0 cuando tiene éxito y un valor distinto de 0 cuando se produce un error.

En base a la operación de posicionamiento, se pueden desarrollar las operaciones de lectura y escritura en cualquier posición de un fichero y por tanto, llevar a cabo los accesos directos.

Lectura en acceso directo

Al igual que en acceso secuancial, las operaciones de lectura en acceso directo consissten en copiar la información contenida en un elemento del fichero sobre una variable del programa localizada en memoria principal, pero en este caso indicaremos necesariamente la posición del elemento del fichero desde la que deseamos realizar la lectura. Para ello, actuamos del siguiente modo:

- Uso de `fseek()` para posicionarnos.
- Uso de `fread()` para leer.

Escritura en acceso directo

De igual modo que en el acceso secuencial, se trata de copiar la información contenida en una variable del programa sobre un elemento del fichero, pero en este caso debemos indicar la posición del fichero en al que deseamos situar el elemento.Para ello, actuamos del siguiente modo:

- Uso de `fseek()` para posicionarnos.
- Uso de `fwrite()` para escribir.

Es importante destacar que la escritura en una posición del fichero no inserta, sino que sobrescribe el contenido del elemento de esa posición.

Veamos el siguiente ejemplo:

Ejemplo 5.5

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

typedef struct{
    char nombre[20];
    float nota;
}ficha;

int main() {
    ficha f;
    FILE *p;
    int i;
    char opcion;
    if (!(p = fopen("Datos.bat", "rb+"))) {
        printf("No se puede abrir.");
        p = fopen("Datos.bat", "wb+"); //Lo creamos
    }
    else {
        do {
            printf("1.-Mostrar ficha\n");
            printf("2.-Escribir ficha\n");
            printf("3.-Salir\n");
            opcion = getch();
            switch (opcion) {
                case '1':
                    printf("Indique la posicion: ");
                    scanf("%d", &i);
                    if (fseek(p, (i-1) + sizeof(f), SEEK_SET) == 0) {
                        if (fread(&f, sizeof(f), 1, p) == 1) {
                            printf("Nombre: %s\n", f.nombre);
                            printf("Nota: %f\n", f.nota);
                        }
                        else
                            printf("Dicha posicion no existe.\n");
                    }
                    else
                        printf("No existe esa posicion\n");
                    break;
                case '2':
                    printf("Indique la posicion: ");
                    scanf("%d", &i);
                    if (fseel(p, (i-1) * sizeof(f), SEEK_SET) == 0) {
                        printf("Nombre: ");
                        scanf("%s", f.nombre);
                        printf("Nota: ");
                        scanf("%f", &f.nota);
                        fwrite(&f, sizeof(f), 1, p);
                        printf("Operacion realizada.\n");
                    }
                    else
                        printf("Error en posicionamiento");
                    break;
```

```
53                  }
54              } while (opcion != '3');
55              fclose(p);
56          }
57          return 0;
58  }
```

Explicación

En este ejemplo se maneja una serie de fichas con notas de los alumnos para ejemplificar el uso de las funciones anteriormente vistas.