

Manual comprensible de C

Francisco Rodríguez Melgar

Ingeniero informático

19 de diciembre de 2022



«Como es más lo que ignoras que lo que sabes, no hables mucho.»

—Raimundo Lulio



Índice

1. Introducción	1
1.1. ¿Qué es la programación?	1
1.2. ¿Cómo se programa?	1
2. Preparación del entorno	3
2.1. Sistema operativo	3
2.2. Instalar el compilador	6
3. Tu primer programa; ¡dile hola al mundo!	8
3.1. Editor de texto	10
4. Primeros pasos	11
4.1. Variables	11
4.2. Imprimir cosas	15
4.3. Operadores	16
4.3.1. Cástring: conversiones explícitas	19
4.3.2. Ejemplo final de programa con operadores	20
5. Alterando el flujo normal del programa	22
5.1. Sentencias condicionales	22
5.1.1. Operaciones lógicas	22
5.1.2. Bifurcando el programa: el if	25
5.2. Bloques de código y alcance	30
5.3. Otras instrucciones de salto: switch y goto	31
5.4. Repetir instrucciones: bucles	33
5.4.1. El bucle while	34
5.4.2. El bucle for	35
5.5. Interrupción de bucles	36
6. Estructuras de datos	39
6.1. El array	39
6.2. La estructura	41
6.3. Listas de inicialización	43
6.4. Ejercicios de la sección	45
7. Funciones	47
7.1. Separación entre declaración y definición	50
7.2. Las funciones y los arrays	51
7.3. Ejercicios de la sección	52
8. La memoria	54
8.1. Sistemas numéricos posicionales: decimal, binario y hexadecimal	55
8.2. El mapa de memoria	57
8.3. Punteros	61
8.3.1. Aritmética de punteros	63
8.3.2. El puntero a char	65
8.3.3. El puntero nulo	66
8.4. Alojamiento y liberación de memoria	67
8.5. Composición de punteros	71
8.6. Ejercicios de la sección	77
9. Modificadores de tipos: constancia y signo	78



9.1. Ejercicios de la sección	83
10. Comunicar tu programa con el exterior	84
10.1. Ejercicios de la sección	95
11. Cómo escribir programas legibles y claros	97
11.1. Estilo de código	104
11.1.1. Tabulación	106
11.1.2. Espaciado entre símbolos	108
11.1.3. Estilo de las llaves	109
11.1.4. Declaración de variables, estructuras y enumerados	110
11.1.5. Convenciones de nombres	111
12. Lo que hay detrás de la compilación	112
12.1. Preprocesado	112
12.2. Compilación de objetos	118
12.3. Enlazado	123
13. Funciones de la biblioteca estándar	125
13.1. Manejo de memoria	125
13.2. Manejo de cadenas de texto	129
13.2.1. Especificadores posicionales	136
13.3. Manejo de errores y manual	138
13.4. Ejercicios de la sección	140
14. Lógica avanzada	142
14.1. Operadores a nivel de bit	142
15. Algoritmos	148
15.1. Recursividad	149
15.2. Algoritmos de ordenación	152
15.3. Búsqueda	160
16. Algoritmos genéricos	163
17. Ejemplo completo de programa	173
18. Anexo A: soluciones a ejercicios	180



Índice de figuras

1.	Ejemplo de sistema de ficheros	4
2.	Configurar ver las extensiones	8
3.	Diagrama de flujo: resolución de sistema de ecuaciones con condicional	22
4.	Diagrama de flujo: programa que imprime los números del 1 al 100	34
5.	Diagrama de flujo de un bucle <code>for</code>	35
6.	Módulo de memoria RAM	54
7.	Mapa de memoria vacío	57
8.	Regiones del mapa de memoria de un proceso	59
9.	Mapa de memoria de un vector de vectores (doble puntero)	74
10.	Mapa de memoria de un doble array	75
11.	Puntero a puntero a <code>char</code> constante	170
12.	Puntero a puntero constante a <code>char</code>	171



Índice de tablas

1.	Tipos básicos de C	11
2.	Especificadores de formato en C	15
3.	Imprimir caracteres especiales en C	15
4.	Operadores matemáticos básicos	17
5.	Conversiones de tipos en C	19
6.	Ejemplo de operación lógica	23
7.	Tablas de la verdad de las operaciones lógicas	23
8.	Operadores de comparación de C	24
9.	Ejemplo de una cuadrícula con valores	55
10.	Ejemplo del estado de la pila en una ejecución	61
11.	Tiempos de ejecución de los distintos algoritmos	159
12.	Tiempos de ejecución de los distintos algoritmos	172



Índice de programas

1.	Hola Mundo en C	11
2.	Creación y asignación de variables	12
3.	Asignación de variables entre ellas	13
4.	Ejemplo final de variables	13
5.	Asignaciones inválidas	14
6.	Ejemplo de impresión.	16
7.	División entera contra división decimal	17
8.	Operadores de incremento y decremento	18
9.	Ejemplo de cásting	19
10.	Cálculo de ecuación lineal	20
11.	Primer programa con operaciones lógicas	24
12.	Primer programa con operaciones de comparación	25
13.	Esquema básico de instrucción if	25
14.	Programa del aspersor con operadores lógicos y de comparación	26
15.	Programa de resolución de ecuaciones lineales con condicionales	26
16.	Programa ejemplo de estructura if-else	27
17.	Programa de resolución de ecuaciones lineales con condicionales	28
18.	Ejemplo de alcance de variables declaradas	30
19.	Ejemplo de redifinición de variable	30
20.	Ejemplo de programa con una variable global	31
21.	Ejemplo de programa con un switch	32
22.	Ejemplo de programa con un goto	33
23.	Ejemplo de programa con un bucle while	34
24.	Ejemplo de programa con un bucle do-while	35
25.	Estructura de un bucle for	36
26.	Ejemplo de programa con un bucle for	36
27.	Ejemplo de interrupción de un bucle con una variable auxiliar	36
28.	Interrupción de un bucle con la instrucción break	37
29.	Ejemplo de algoritmo de año bisiesto	38
30.	Ejemplo de algoritmo con continue	38
31.	Ejemplo de declaración de array	39
32.	Ejemplo de uso de array	39
33.	Ejemplo de uso de array	40
34.	Ejemplo de uso de array bidimensional	40
35.	Declaración, instanciación y uso de un <i>struct</i>	41
36.	Ejemplo de resolución de distancia entre puntos	42
37.	Ejemplo de resolución de distancia entre puntos usando estructuras	43
38.	Inicialización con llaves	44
39.	Inicialización con llaves de struct con selección de campos	44
40.	Combinación <i>struct</i> con array	45
41.	Declaración de una función en C	47
42.	Definición de una función en C	47
43.	Ejemplo de función en C	48
44.	Invocación de función en C	49
45.	Demostración de que una función recibe copias de sus argumentos	49
46.	Declaración no separada de definición	50
47.	Declaración separada de definición	51
48.	Uso de arrays en funciones	52
49.	Ejemplo de uso de punteros	62
50.	Declaración de punteros	63
51.	Arrays como punteros	64



52.	Aritmética de punteros	64
53.	Ejemplo práctico de aritmética de punteros	65
54.	Arrays de char	66
55.	Uso de punteros a NULL	67
56.	Ejemplo de reserva dinámica	69
57.	Diferencia entre sizeof con punteros y arrays	71
58.	Reserva, uso y liberación de un vector de vectores	72
59.	Uso de un array como una estructura unidimensional	76
60.	Uso de una constante numérica	78
61.	Uso de punteros constantes como argumentos de función	79
62.	Estructura con punteros constantes – Funciones de manipulación	81
63.	Estructura con punteros constantes – Funciones de recuperación de información	82
64.	Estructura con punteros constantes – función main	82
65.	Uso de tipos sin signo	83
66.	Declaración de una función main que reciba argumentos	84
67.	Utilización de los argumentos de un programa	84
68.	Programa que suma argumentos	85
69.	Ejemplo básico de scanf	86
70.	Ejemplo avanzado de scanf	86
71.	Declaración de la función fopen	87
72.	Declaración de la función fread	88
73.	Declaración de la función fwrite	88
74.	Declaración de la función fclose	88
75.	Ejemplo de manejos de ficheros	89
76.	Ejemplo de lectura de fichero con <i>buffer</i>	91
77.	Ejemplo de uso de funciones para mover la cabeza de lectura	93
78.	Declaración de la función remove	94
79.	Ejemplo de programa que usa la función remove	95
80.	Definición de un tipo a partir de una estructura	97
81.	Diferentes combinaciones de struct con typedef	98
82.	Estructura anónima y efímera	98
83.	Redefinición de un tipo básico	99
84.	Ejemplo de definición de tipo a partir de un tipo constante	100
85.	Definición de un tipo personalizado a partir de un array	100
86.	Ejemplo básico de tipo enumerado	101
87.	Enum combinado con array de nombres	102
88.	Ejemplo final de enumerados	103
89.	Ejemplo de programa escrito con un mal estilo	104
90.	Ejemplo de programa escrito con un buen estilo	105
91.	Impresión larga	107
92.	Impresión larga	107
93.	Muchos argumentos	107
94.	Cómo acortar líneas con muchos argumentos	108
95.	Acortamiento de una declaración de función	108
96.	Ejemplo de llaves estilo K&R	109
97.	Ejemplo de llaves estilo Allman	109
98.	Declaración de variables en una misma línea	110
99.	Declaración de miembros de un <i>struct</i> en una sola línea	110
100.	Ejemplo de tipo enumerado con typedef	111
101.	Ejemplo de función descriptiva	111
102.	Ejemplo de función no descriptiva	111
103.	Example of program with comments	113
104.	Creación de macros	114



105. Uso de <code>define</code> con arrays	114
106. Uso de macro con argumentos	115
107. Ejemplo de error por una macro	115
108. Uso de macros con cadenas de texto	116
109. Macro para convertir a string	116
110. Conversión de macros numéricas a string	117
111. Ejemplo de directiva <code>include</code> , archivo principal	117
112. Ejemplo de directiva <code>include</code> , archivo incluido	118
113. Uso de directivas <code>ifdef</code> e <code>ifndef</code>	118
114. Archivo de cabecera	119
115. Archivo de definiciones con cabecera incluida	120
116. Archivo principal con cabeceras incluidas	120
117. Ejemplo de redefinición – <code>point.h</code>	121
118. Ejemplo de redefinición – <code>point.c</code>	121
119. Ejemplo de redefinición – <code>circle.h</code>	121
120. Ejemplo de redefinición – <code>circle.c</code>	122
121. Ejemplo de redefinición – <code>main.c</code>	122
122. Ejemplo de <i><code>include guard</code></i>	123
123. Declaración de la función <code>calloc</code>	125
124. Declaración de la función <code>realloc</code>	125
125. Utilización de <code>realloc</code>	126
126. Declaración de la función <code>memset</code>	127
127. Utilización de la función <code>memset</code>	128
128. Declaración de <code>memcpy</code>	128
129. Utilización de la función <code>memcpy</code>	129
130. Ejemplo de uso de <code>strcmp</code>	130
131. Propia versión de <code>strlen</code>	130
132. Declaración de la función <code>strlen</code>	131
133. Definición de <code>strdup</code>	131
134. Ejemplo básico de <code>sprintf</code>	132
135. Ejemplo de uso avanzado de <code>sprintf</code>	133
136. Ejemplo de uso de <code>snprintf</code>	135
137. Ejemplo de impresión con argumento repetido	137
138. Ejemplo de impresión con argumento repetido y especificador posicional	138
139. Ejemplo de programa que usa la variable <code>errno</code>	139
140. Ejemplo del uso del tipo <code>bool</code>	142
141. Implementación de opciones con operaciones a nivel de bit	145
142. Ejemplo de bajada de una bandera	146
143. Comparación de algoritmos recursivos e iterativos	150
144. Función para el cálculo de la sucesión de Fibonacci	151
145. Implementación del algoritmo de la burbuja	152
146. Implementación de <code>swap</code> y uso en algoritmo de la burbuja	153
147. Algoritmo de selección	153
148. Algoritmos auxiliares al de inserción	155
149. Algoritmo de inserción	155
150. Implementación alternativa de ordenación por inserción	156
151. Implementación de <i>Quick Sort</i>	158
152. Implementación del algoritmo de búsqueda binaria	161
153. Inversión de arrays de cualquier tipo	163
154. Imprimir arrays de varios tipos	165
155. Ejemplo primero de puntero a función como argumento	166
156. Llamada a una función que recibe un puntero a función	166
157. Definición de función de impresión genérica	167



158. Definición de tipos puntero a función	168
159. Ejemplo final de función que recibe un puntero	168
160. Definición de <code>bubble_sort</code> genérico	169
161. Función auxiliar de comparación de <i>strings</i>	170
162. Cómo medir el tiempo	172
163. Ejemplo final de programa – <code>person.h</code>	173
164. Ejemplo final de programa – <code>person.c</code> definiciones	175
165. Ejemplo final de programa – <code>person.c</code> manipulación	176
166. Ejemplo final de programa – <code>person.c</code> recuperación	177
167. Ejemplo final de programa – <code>main.c</code>	178
168. Solución al ejercicio 1	180
169. Solución al ejercicio 2	181
170. Solución al ejercicio 3	182
171. Solución al ejercicio 4	183
172. Solución al ejercicio 5	183
173. Solución al ejercicio 6	184
174. Solución al ejercicio 7	184
175. Solución al ejercicio 8	184
176. Solución al ejercicio 9	186
177. Solución al ejercicio 12	188
178. Solución al ejercicio 13 – reserva	189
179. Solución al ejercicio 13 – impresión	189
180. Solución al ejercicio 13 – liberación	189
181. Solución al ejercicio 13 – función <code>main</code>	190
182. Solución al ejercicio 13 – función <code>main</code>	190
183. Solución al ejercicio 15	191
184. Solución al ejercicio 16	192
185. Solución al ejercicio 17	194
186. Solución al ejercicio 18	195
187. Solución al ejercicio 19	196
188. Solución al ejercicio 20	197
189. Solución al ejercicio 21	198
190. Solución al ejercicio 23 – <code>tagged_point.h</code>	199
191. Solución al ejercicio 22 – <code>tagged_point.c</code>	200
192. Solución al ejercicio 22 – <code>main.c</code>	201



1. Introducción

En este documento quiero poder explicar, al menos, los fundamentos básicos del lenguaje de programación C. Además, espero ofrecer motivos para aprenderlo, e intentar transmitir al lector parte de la belleza que me inspira. Bajo este primer epígrafe explicaré primero lo que es la programación, qué tipo de lenguajes existen y ofrecer una explicación de cómo estructuraremos este documento.

1.1. ¿Qué es la programación?

Siendo este un manual no muy avanzado, es posible que aún no hayas tenido experiencia en programación. Si ése es el caso, en esta sección te explicaré de manera somera qué es «programar». Programar es, según el diccionario de la Real Academia Española, «Elaborar programas para su empleo en computadoras», no se quedaron calvos los académicos con esta definición, pero nos sirve para empezar, programar es elaborar programas, dicho esto, para saber qué es programar, debemos responder a la pregunta de qué es un programa.

Un programa es el conjunto de instrucciones que un ordenador sigue para realizar una tarea concreta. Por ejemplo, si tú fueras un ordenador, e hiciéramos un programa para que compraras un café en una máquina expendedora, el programa que tú, como ordenador viviente, seguirías se parecería a lo siguiente:

1. Levántate
2. Camina hasta la máquina de café
3. Escoge el café que deseas tomar
4. Mira el precio de ese café
5. Introduce el precio de ese café en la ranura para monedas
6. Pulsa el botón del café deseado
7. Espera a que se haga
8. Recoge el café, ¡y ten cuidado de no quemarte!

Dicho así, sería genial poder decirle a tu ordenador, o a cualquier ordenador, algo como «resuelve esta ecuación diferencial» o «predice el tiempo que hará mañana». Tristemente, es aquí donde entra en juego la pericia del programador, pues los ordenadores no entienden el lenguaje de las personas, ni saben lo que es el tiempo, ni lo que es un café. Los ordenadores sólo entienden de operaciones matemáticas (y no muchas) y de operaciones lógicas (si no sabes lo que es la lógica, como ciencia, no te preocupes). El programador debe ser capaz de convertir una tarea compleja en instrucciones que un ordenador pueda comprender.

Así que, finalmente, podríamos decir que programar es: «expresar tareas complejas expresadas en lenguaje humano en términos de tareas simples que un ordenador pueda entender».

1.2. ¿Cómo se programa?

Ahora que sabemos **qué** es programar, veamos **cómo** se hace, en términos generales. Siguiendo con la metáfora que establecí en el apartado anterior, un «programa», tal y como lo entendemos, es un archivo de texto -o varios- donde se expresan esas instrucciones para que el ordenador haga algo. Como he dicho antes, los ordenadores entienden un número de instrucciones muy pequeño, y, de hecho, sólo las entienden en código binario. Un ordenador almacena en su memoria (lo que se conoce como memoria RAM, coloquialmente) las instrucciones que debe ejecutar. En esa memoria sólo se almacenan *bits*, es decir, un valor en un sistema numérico que sólo tiene dos cifras: el uno y el cero.



Si aplicamos la definición del apartado anterior, para programar, deberíamos escribir los programas en ceros y unos. Para que te hagas una idea, el programa Firefox, el navegador de Internet, ocupa unos 500 KB, o lo que es lo mismo: medio millón de *bytes*. Un byte son ocho bits. Eso quiere decir que el programador que, hipotéticamente, escribiera Firefox, habría tenido que escribir un archivo continuo de cuatro millones de ceros y unos. Es lógico pensar que eso no es así.

Desde los primeros tiempos de la informática moderna se han ideado **lenguajes formales** que explican de una manera entendible para el ser humano cómo debe ser un programa, pero que permiten crear un programa en esos ceros y unos. Aquí es donde entran los distintos nombres de lenguajes que quizás has oído nombrar: C, C++, C#, Java, Rust... Todos esos lenguajes se diferencian en que son maneras distintas (cada una con sus ventajas e inconvenientes) de componer un programa que, después de un proceso, el ordenador pueda entender. Este proceso es la **compilación**. Compilar un programa es convertirlo desde ese lenguaje intermedio que los humanos podemos entender (y que tú vas a aprender a escribir, espero que con mi ayuda) a un lenguaje puramente de ordenadores. El código escrito en esos lenguajes se llama **código fuente**, o, en inglés: *source code*, porque es la fuente de donde sacaremos (compilaremos) nuestros programas. En general no voy a hacer la distinción entre programa (programa compilado) y código fuente. Los distinguiremos por el contexto.

Así que si añadimos a lo que ya teníamos esta nueva información, podemos decir que programar es: «escribir un archivo en un lenguaje de programación que pueda ser compilado a un programa que el ordenador pueda ejecutar directamente».



2. Preparación del entorno

Quizás estás ya impaciente, o quizás hace mucho que dejaste de leer, pero creo que esa introducción era pertinente, al menos para quien no supiera lo que era programar en el nivel más básico. Ahora vamos a hablar de cómo preparar un **entorno** para programar. El entorno es el conjunto de herramientas que vamos a utilizar para escribir y compilar nuestros programas. El problema de C es que es un lenguaje muy «cercano al ordenador», ¿qué quiere decir esto? Que es más difícil de entender y de escribir para las personas, por lo que la preparación que vas a tener que hacer para programar en C es un poco más complicada que si usaras otros lenguajes. Así que vamos por partes.

2.1. Sistema operativo

Si estás leyendo este manual, o esta parte, entiendo que no has explorado la programación anteriormente. Empecemos por el principio. Como C es uno de esos lenguajes más fáciles de entender para el ordenador que otros, debemos preguntarnos qué sistema operativo tenemos. Si no le has hecho nada a tu ordenador, lo más probable es que tengas un ordenador con Windows. Lo ideal sería que instalaras Linux, o que crearas una máquina virtual con Linux, o, al menos, que utilices el Subsistema de Linux para Windows.

Como explicar todas las alternativas haría el manual muy extenso y, además, me obligaría a hacer distinciones en cada uno de los apartados que vienen a continuación, voy a suponer que utilizarás el subsistema de Linux para Windows o, por sus siglas en inglés (*Windows Subsystem for Linux*, WSL). El primer paso es instalar la característica de Windows que nos permite hacer esto. Para hacer esto, pulsa la tecla de Windows en tu teclado y la letra R, después, escribe `optionalfeatures` y pulsa intro. Se te abrirá una ventana pequeña donde deberás buscar el elemento: «Subsistema de Windows para Linux», selecciona la casilla de la izquierda y dale a aceptar. Reinicia cuando se te sugiera. Después irás a la tienda de Windows (Microsoft Store) y buscarás «Ubuntu», elige el primer resultado e instálala. Después, ve al menú inicio y busca la aplicación (Ubuntu), ábrela. Tardará un momento en instalarse. Después te pedirá un nombre de usuario, y que escribas dos veces una contraseña. ¡Recuérdalos! Cuando termines, estarás delante de una ventana negra con un texto que será: `{usuario}@{nombre de la máquina}:~$`. Enhorabuena, ya tienes instalado un Linux que puedes usar en Windows.

Esta ventana negra que sólo contiene letras se llama terminal, y es una manera de interactuar con el ordenador que se lleva usando desde hace décadas. En vez de hacer clic en iconos, escribirás comandos en la terminal y le darás a enter. Te voy a dar una serie de comandos básicos y de conceptos para que sepas cómo utilizarla. En una terminal está en todo momento en un «directorio de trabajo», por ejemplo, si escribes `pwd` y le das a enter, te dirá en qué directorio (carpeta) estás ahora mismo. A continuación te muestro un ejemplo de cómo se vería:

```
usuario@DESKTOP-U80A808:~$ pwd
/home/usuario
```

Con el comando `cd` te mueves de directorio de trabajo. Todo directorio tiene dentro dos directorios especiales, el directorio punto (`.`) y el punto punto (`..`). El primero simboliza el mismo directorio, es decir, si haces `cd .` nunca harás nada, y el segundo el directorio superior, por ejemplo:

```
usuario@DESKTOP-U80A808:~$ pwd
/home/usuario
usuario@DESKTOP-U80A808:~$ cd ..
usuario@DESKTOP-U80A808:/home$ pwd
/home
usuario@DESKTOP-U80A808:/home$
```



Para que te quede una idea más sólida de lo que es un sistema de ficheros, voy a explicártelo más detenidamente. En Windows, todos tus archivos y directorios están en unidades que tienen asignadas letras y acaban en dos puntos: (:). Por ejemplo, la ruta más habitual para el escritorio es `C:\users\<nombre de usuario>\Escritorio` en sistemas Windows en español. Por otro lado, en Linux esto no es así, en este sistema todos los directorios nacen desde el directorio raíz (/). Nota, si no lo has hecho ya, que en Windows las rutas de los directorios se simboliza mediante barras invertidas (\) y en Linux con barras inclinadas normales (/). Cada unidad en Windows suele ser un disco físico: un pendrive, un disco duro mecánico, un SSD... En Linux, cuando insertas una unidad o disco, simplemente se **monta** en un directorio que cuelga del raíz. Montar simplemente significa que el sistema de ficheros del disco que hemos añadido (todas sus carpetas y contenido) se «enganchan» a los directorios que ya teníamos en un punto (en un directorio) como si fueran contenido de una carpeta.

Estos directorios funcionan como una serie de puntos unidos por conexiones. Cada directorio o archivo es un punto, conectado al directorio de los que cuelga y teniendo conectados a él los directorios y archivos que están en él. Esto se suele representar en un dibujo como este que te presento ahora:

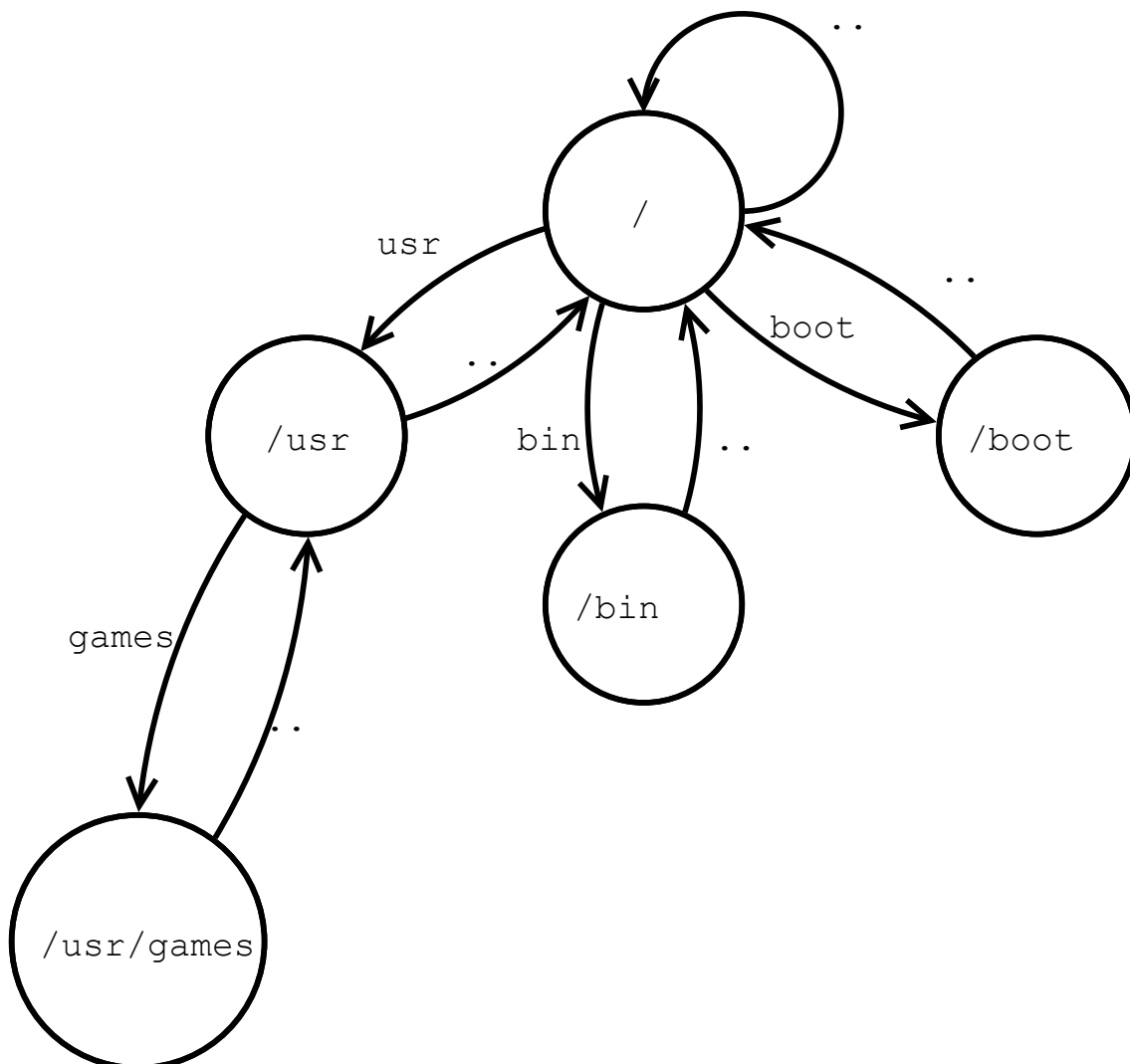


Figura 1: Ejemplo de sistema de ficheros



En esta figura, cada directorio tiene enlazados aquéllos dentro de él, y todos ellos muestran a dónde enlaza el directorio `..` que está dentro de ellos. Como puedes ver, para llegar a una ruta como `/usr/games` sólo tenemos que ir dando «saltos» desde un directorio al siguiente. Si quisiéramos volver hacia atrás, podemos usar los directorios ficticios que apuntan al padre de cada uno. El caso del raíz es especial, porque su directorio padre es él mismo. En Windows el sistema es igual, pero no existe una raíz, sino varias, siendo éstas las unidades de disco. Por otro lado, no tenemos por qué hacer `cd` directorio a directorio, podemos introducir rutas completas. Una ruta, si no lo has deducido ya, es simplemente la sucesión de directorios que llevan, desde uno dado, a otro. Hay dos tipos de rutas:

1. Rutas absolutas: Empiezan por barra inclinada (`/`) e indican una ruta desde el directorio raíz hasta uno concreto, por ejemplo: `/home/john/music/Beethoven_symphony.mp4` sería una ruta absoluta.
2. Rutas relativas: Son aquéllas cuyo inicio no es la raíz, sino el directorio de trabajo. Por ejemplo: `music/Beethoven_symphony.mp4` es una ruta relativa que sólo apuntará a un archivo que existe si en el directorio de trabajo actual existe un directorio llamado `music` y, dentro de éste, un archivo llamado `Beethoven_symphony.mp4`.

Eso que sale cada vez que pulsas enter se llama *prompt*, y en general te muestra tu nombre de usuario (en mi caso: «usuario»), la máquina donde estás (en mi caso: «DESKTOP-U8OA808») y después el directorio de trabajo, si cabe. Yo voy a sustituir el *prompt* por únicamente el símbolo del dólar en los ejemplos que te muestre, para que quepa el texto mejor. La primera vez que abriste la terminal, ésta mostraba una virgulilla (`~`) porque ese es un alias de tu directorio *home*, que es una ruta de los sistemas Linux donde el usuario almacena sus archivos personales. En general esa ruta está en `/home/<nombre de usuario>`.

Ahora vamos a aprender a ver lo que hay en un directorio, el comando que hace eso se llama `ls`, si lo tecleas y pulsas enter, verás que... que no sale nada. Eso es porque no hemos creado ningún archivo ni directorio en nuestra *home*, el comando que crea archivos es `touch`, escribe `touch prueba.txt` y pulsa enter, si haces `ls`, verás que ahora aparece.

```
$ touch prueba.txt
$ ls
prueba.txt
$
```

`ls` tiene muchas opciones, las opciones en comandos de Linux se ponen con un guion delante, así que si te dicen que debes escribir, «`ls` con las opciones `a` y `l`», debes escribir `ls -l -a` o, juntando todas las opciones en el mismo guion: `ls -la`. Debería pasar algo como esto:

```
$ ls -la
total 8
drwxr-xr-x 1 usuario usuario 512 Jul  8 19:05 .
drwxr-xr-x 1 root    root    512 Jul  7 22:37 ..
-rw-r--r-- 1 usuario usuario 220 Jul  7 22:37 .bash_logout
-rw-r--r-- 1 usuario usuario 3771 Jul  7 22:37 .bashrc
drwxr-xr-x 1 usuario usuario 512 Jul  7 22:37 .landscape
-rw-rw-rw- 1 usuario usuario  0 Jul  8 18:25 .motd_shown
-rw-r--r-- 1 usuario usuario 807 Jul  7 22:37 .profile
-rw-rw-rw- 1 usuario usuario  0 Jul  8 19:05 prueba.txt
```



Como verás, hay muchos archivos que tú no has creado. Esto es porque la opción `a` hace que `ls` nos muestre **todos los archivos**, incluidos los ocultos, que son los que empiezan por punto (`.`). Y la opción `l` hace que salgan en una lista, y con información sobre ellos. Los directorios ocultos empiezan con punto, y esto provoca que cuando haces un `ls` normal no aparezcan, convenientemente, los directorios punto y punto-punto. Si esto te intimida, no te preocupes, es normal, y tengo buenas noticias, el WSL ve los directorios que tienes en tu sistema Windows, así que puedes crear una carpeta en tu escritorio y trabajar con el explorador de Windows, para crear o eliminar los archivos.

2.2. Instalar el compilador

Ahora que ya has instalado una distribución de Linux que puedes usar, vamos a instalar el compilador. Para ello ejecuta los comandos que salen a continuación y, si te preguntan si quieres continuar, di que sí.

```
$ sudo apt update
#Aquí saldrá mucho texto, ignóralo.
$ sudo apt install build-essential
.....
After this operation, 189 MB of additional disk space will be used.
Do you want to continue? [Y/n] y
.....
```

Ahora ya deberías tener instalado el compilador de C, que se llama GCC (por sus siglas en inglés: *GNU Compiler Collection*). Para ver si es cierto, haz lo que sale a continuación.

```
$ gcc -v
Using built-in specs.
COLLECT_GCC=gcc
COLLECT_LTO_WRAPPER=/usr/lib/gcc/x86_64-linux-gnu/9/lto-wrapper
OFFLOAD_TARGET_NAMES=nvptx-none:hsa
OFFLOAD_TARGET_DEFAULT=1
Target: x86_64-linux-gnu
Configured with: ../src/configure -v --with-pkgversion='Ubuntu 9.3.0-17
ubuntu1~20.04' --with-bugurl=file:///usr/share/doc/gcc-9/README.
Bugs --enable-languages=c,ada,c++,go,brig,d,fortran,objc,obj-c++,
gm2 --prefix=/usr --with-gcc-major-version-only --program-suffix=-9
--program-prefix=x86_64-linux-gnu- --enable-shared --enable-linker
-build-id --libexecdir=/usr/lib --without-included-gettext --enable
-threads=posix --libdir=/usr/lib --enable-nls --enable-clocale=gnu
--enable-libstdcxx-debug --enable-libstdcxx-time=yes --with-default
-libstdcxx-abi=new --enable-gnu-unique-object --disable-vtable-
verify --enable-plugin --enable-default-pie --with-system-zlib --
with-target-system-zlib=auto --enable-objc-gc=auto --enable-
multiarch --disable-werror --with-arch-32=i686 --with-abi=m64 --
with-multilib-list=m32,m64,mx32 --enable-multilib --with-tune=
generic --enable-offload-targets=nvptx-none=/build/gcc-9-HskZEa/gcc
-9-9.3.0/debian/tmp-nvptx/usr,hsa --without-cuda-driver --enable-
checking=release --build=x86_64-linux-gnu --host=x86_64-linux-gnu
--target=x86_64-linux-gnu
Thread model: posix
gcc version 9.3.0 (Ubuntu 9.3.0-17ubuntu1~20.04)
```




Si te sale un texto parecido, enhorabuena, ya has instalado el compilador de C. Ahora vamos, (por fin, estarás pensando) a empezar a aprender los conceptos del lenguaje.



3. Tu primer programa; ¡dile hola al mundo!

Vamos a navegar a esa carpeta que tienes en tu escritorio, las unidades de tu ordenador (los «discos» C, D, E...) se presentan en tu WSL en los directorios dentro de /mnt/, así, C: sería: /mnt/c. A continuación te dejo el ejemplo de cómo navegar en Linux a una carpeta llamada hola_mundo en tu escritorio de Windows. Puedes crearla ahora.

```
$ cd /mnt/c/Users/Usuario/Desktop/  
$ cd hola_mundo  
$ pwd  
/mnt/c/Users/Usuario/Desktop/hola_mundo  
$
```

Ahora que ya estás en la carpeta, ábrela en el explorador de Windows, porque nos queda una última configuración, en la ventana del explorador de archivos, arriba, pincha en la pestaña «vista» y marca la casilla donde pone «extensiones de nombre de archivo». Te dejo una foto de cómo debe quedar:

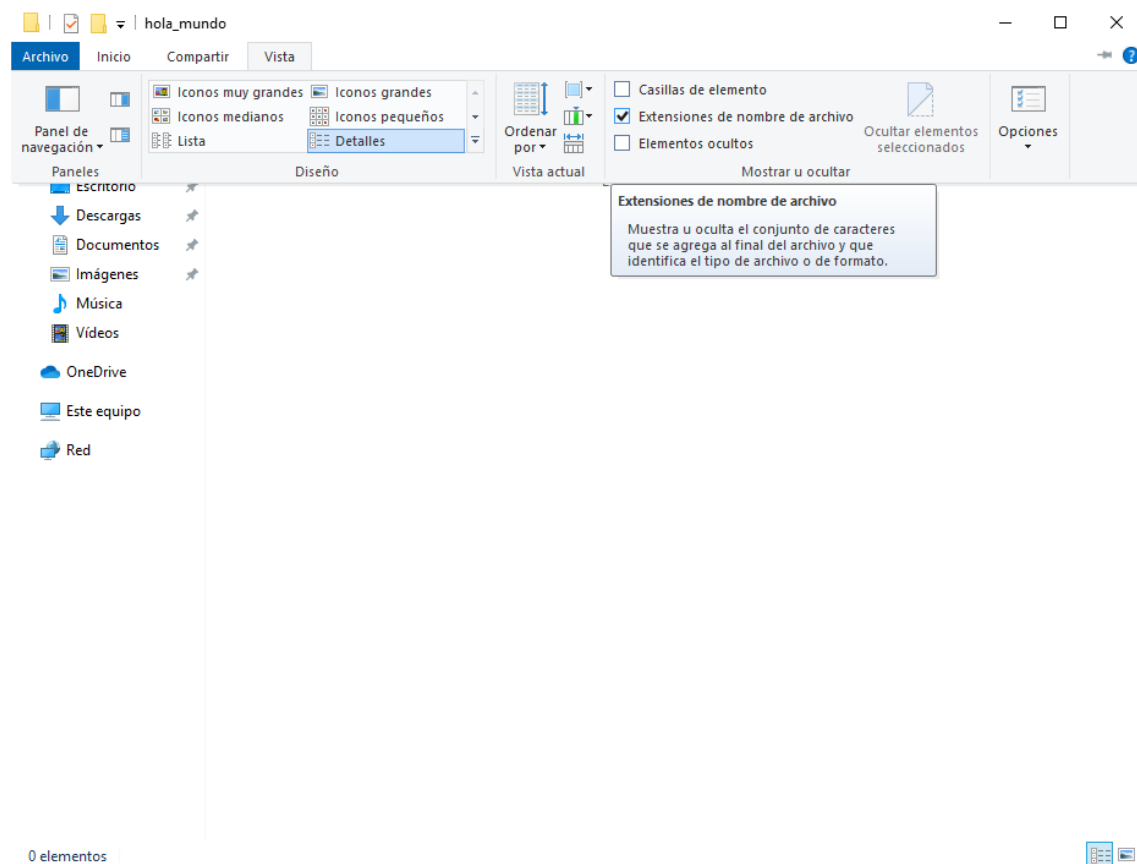


Figura 2: Configurar ver las extensiones

Ahora, con el método que prefieras (la terminal o el ratón), crea en esa carpeta un archivo llamado hola_mundo.c. Y ábrelo con el bloc de notas de Windows, y cuando digo el bloc de notas, no me refiero ni al wordPad ni a Word, sino al bloc de notas. En ese documento vamos a escribir esto:



```
#include <stdio.h>

int main(void)
{
    printf("¡Hola, mundo!\n");
}
```

Te aclaro que la línea que sale desplazada a la derecha es porque tiene espacios delante, puedes poner dos, cuatro u ocho, yo recomiendo cuatro en términos generales. Después guárdalo, y acude a la terminal, deberías tener el fichero en el directorio. Haz un `ls` para comprobarlo. Te recuerdo que si has cambiado de directorio de trabajo o has cerrado la terminal y vuelto a abrirla, deberás volver a navegar a este directorio para poder ver sus archivos y trabajar con él. Ahora vamos a compilar nuestro primer programa. Para eso, estando en el directorio que hemos creado (`hola_mundo`) invocaremos a `gcc`. Debería verse así:

```
$ ls
hola_mundo.c
$ gcc -o hola_mundo.elf hola_mundo.c
```

Al hacer esto, habrá aparecido un archivo llamado `hola_mundo.elf` en la carpeta. ¡Enhorabuena!, ese es tu primer programa en C. ¿Qué hace? Imprime «¡Hola, mundo!». Si haces doble clic en él verás que Windows no sabe abrirlo, porque es un ejecutable para Linux, por ello, en la terminal, escribe `./hola_mundo.elf`. Si recuerdas lo que dije antes, una ruta que no empieza en el directorio raíz (/) es una ruta relativa, y el directorio punto **es el directorio de trabajo**. Cuando a una terminal le indicas un comando y pulsas enter, busca un programa con ese nombre en unos directorios que vienen configurados en tu sistema operativo. Si quieres ejecutar cualquier otro programa (u otras cosas que se pueden ejecutar, pero no vienen al caso ahora) debes indicar una ruta, relativa o absoluta. Para que la terminal sepa que estamos introduciendo una ruta y no un comando, la empezamos por `./`, es decir, este mismo directorio. Cuando escribas la orden y le des a enter, debería verse lo siguiente:

```
$ ./hola_mundo.elf
¡Hola, mundo!
```

Ahora que ya has compilado tu primer programa, quizás te hayas quedado un poco frío, porque no entiendes qué hace, y es el momento en que yo adquiriera un compromiso contigo, el lector, y tú conmigo. Ese compromiso es que, por tu parte, cuando no entiendas algo que aparece en los programas, confíes en que en algún momento explicaré qué es. Por mi parte, mi compromiso es que procuraré hacerlo lo antes posible.

De momento, te voy a explicar qué es el comando que hemos usado para compilar nuestro programa: `gcc` es, como ya te dije, el compilador de C, la opción `o` (recuerda que te expliqué lo que eran las opciones cuando te expliqué cómo usar `ls`) indica que lo que viene a continuación es el nombre del programa que queremos crear y, finalmente, el nombre de nuestro fichero de código fuente. Si detrás de la opción `-o` escribieras otro nombre, el programa resultante se llamaría así, de aquí en adelante la mayoría de los ejemplos se compilarán como `main.exe`.



3.1. Editor de texto

Tu primer programa lo has abierto con el Bloc de notas, pero editar código fuente con eso es bastante insufrible. En parte porque como verás más adelante lo normal es editar archivos de código fuente con un editor que coloree las palabras más importantes (más adelante verás cuáles) y te ayude con cosas como saber dónde terminan las llaves o paréntesis que has abierto. Si quieres editar tu código, hay una gran variedad de editores para ello, aquí te dejo una lista. En el manual no haré referencia a comandos específicos del editor. Algunos editores que puedes usar son:

1. Visual Studio Code.
2. Atom.
3. Sublime Text.
4. Notepad++.



4. Primeros pasos

Ahora que ya sabes lo que es un fichero de código fuente te voy a presentar cómo vamos a incluir los fragmentos de código fuente en el manual. Revisitemos el programa `hola_mundo.c`.

```
1 #include <stdio.h>
2 int main(void)
3 {
4     printf("¡Hola, mundo!\n");
5 }
```

Programa 1: Hola Mundo en C

Como puedes ver, las líneas están numeradas, y hay palabras que salen en color azul, y otras en rojo. Las palabras que salen en azul son las llamadas palabras clave, (en inglés *key words*), de momento, quédate con que todos tus programas deben contener las líneas 1, 2, 3 y 5, y que entre la 3 y la 5 escribirás las instrucciones que compondrán tus archivos de código.

4.1. Variables

¡Por fin empieza lo bueno! Una de las primeras cosas que hacen falta en un programa son **variables**, una variable simboliza un espacio en la memoria del ordenador en que guardamos cosas. Cuando creamos una variable, le decimos al ordenador «reserva espacio en tu memoria para guardar un dato». C es un lenguaje de los que se llaman tipados, es decir, que cada variable tiene un tipo, y éste no puede cambiar una vez que esa variable se crea. En C hay una serie de tipos básicos que te presento en una práctica tabla.

Nombre	Tamaño (en bytes)	Rango	Uso
char	1	$[-128, 127]$	Un carácter de texto o un byte
short	2	$[-32\,768, 32\,767]$	Números en ese rango (generalmente puertos de red)
int	4	$[-2\,147\,483\,648, 2\,147\,483\,647]$	Tipo general para números enteros
float	4	$[\pm 3,4 \cdot 10^{-38}, \pm 3,4 \cdot 10^{38}]$	Números decimales de precisión simple
double	8	$[\pm 1,79 \cdot 10^{-308}, \pm 1,79 \cdot 10^{308}]$	Números decimales de precisión doble

Tabla 1: Tipos básicos de C

Una tabla así puede ser abrumadora, pero es sencillo. Cuando declaramos (creamos) una variable, debemos decir de qué tipo es. Me gusta decir que las variables son como cajas y que, según su tipo, en esa caja entran unas cosas u otras. Para declarar una variable, debes poner su tipo, y un nombre, ¡y no olvides terminar la línea en punto y coma (;)! Además de declararlas, debemos aprender a darles un valor. Eso se llama «asignar un valor» y se hace con el símbolo de igual (=). Para darle un valor se pone el nombre de la variable, el símbolo de igual y el valor, veamos unos ejemplos, y te explico algunas salvedades.



A propósito del nombre: el nombre de una variable se compone de letras, números y guiones bajos. El nombre de una variable no debería estar escrita en mayúsculas, pero puede contenerlas. No puede empezar por número y **no deberías empezarla por un guion bajo**. En general, puedes usar dos notaciones para escribir nombres en C (y en cualquier lenguaje de programación):

1. *Camel Case*: Si un nombre contiene varias palabras, se deben escribir juntas, con la primera palabra de cada una en mayúscula. Por ejemplo: `betterValue` o `targetNumber`. Se llama así porque esas mayúsculas recuerdan a las jorobas de un camello.
2. *Snake Case*: Las palabras aparecen separadas por guiones bajos, en minúscula totalmente, por ejemplo: `better_value` o `target_number`. Se llama así porque la forma de los nombres escritos en este modo recuerda a una serpiente que ha comido animales y tiene bultos en su cuerpo.

```
1 #include <stdio.h>
2 int main(void)
3 {
4     char letter = 'a';
5     char byte = 120;
6     short shorty = 5520;
7     int money_i_want;
8     float money_i_have = 3.22F;
9     double money_you_have = 52.55;
10
11     money_i_want = 450000;
12     shorty = 11111;
13 }
```

Programa 2: Creación y asignación de variables

Los ordenadores no entienden de letras, sólo de números, así que, en el fondo, cuando le dices que `letter` es igual a `'a'`, le dices que es igual al número que significa esa letra. Nota que para poner el valor de un `char` debes usar comillas simples, en el teclado español, es lo que sale al pulsar la tecla de la interrogación de cierre. La correspondencia entre letras y números está escrita en la tabla ASCII, si quieres leerla, te dejo un enlace a un sitio donde la puedes consultar. Si miras allí, verás que la letra `a` es el valor 97. Después asignamos a otro `char` un valor numérico, en la línea 7 puedes ver que declaramos una variable sin darle un valor. Esto es perfectamente correcto, pero, ¡cuidado!, **una variable a la que no le das valor tiene un valor aleatorio**. Darle valor a una variable por primera vez se llama «inicializarla». Por este motivo muchos profesores te dirán que siempre que declares una variable le des un valor inmediatamente.

En las líneas 11 y 12 le damos valor a variables que hemos declarado anteriormente, y quiero detenme en el caso de la línea 12. Al principio puede parecer que al escribir `shorty = 5520;` estamos enunciando una igualdad matemática, es decir, que eso siempre será así, pero en C no trabajamos con «leyes», sino con instrucciones, así que no debes leer esa línea como «`shorty` es igual a 5520» sino como «He asignado a `shorty` el valor 5520». Es decir, has metido en esa «caja» un 5520, pero nada te impide sacar ese valor y meter otro número como estamos haciendo.

Los valores que escribe el programador en el código se llaman «literales». Creo que el nombre no necesita explicación. Cada literal tiene un tipo determinado, al igual que las variables. Más adelante veremos por qué eso es importante. De momento, recuerda que se llama literal a los valores que el programador escribe en el código explícitamente. Por ejemplo, los números (como 5520) o las letras (`'a'`). Además de literales, podemos asignarle a una variable el valor de una **expresión**, una expresión es cualquier texto escrito en lenguaje C que tenga un valor. De momento, nos vamos a limitar a asignar unas variables a otras. Por ejemplo:



```
1 #include <stdio.h>
2 int main(void)
3 {
4     int a = 3;
5     int b = 2;
6
7     a = b;
8 }
```

Programa 3: Asignación de variables entre ellas

En la línea 7 podemos ver como asigno a la variable `a` el valor de `b`, es decir, ahora `a` valdrá 2. Como ver esto así es difícil, voy a incluir en el siguiente ejemplo una serie de líneas que contienen la palabra `printf`, te sonará porque la usamos en nuestro primer ejemplo, sirve para que el programa escriba cosas en tu terminal. Después te enseñaré a usarla. De momento, simplemente copia el siguiente programa en tu archivo de código fuente y compila como lo hicimos antes. (Copiar desde un PDF suele ser mala idea, teclea a mano, así practicarás)

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int a = 3;
5     int b = 10;
6     printf("a vale: %d\n", a);
7     a = b;
8     printf("b vale: %d y a vale lo mismo, es decir: %d\n", b, a);
9     b = 22;
10    printf("b vale: %d, a sigue valiendo %d\n", b, a);
11 }
```

Programa 4: Ejemplo final de variables

Si tecleas este programa, lo compilas y lo ejecutas debería salirte algo como esto:

```
$ ./main.elf
a vale: 3
b vale: 10 y a vale lo mismo, es decir: 10
b vale: 22, a sigue valiendo 10
```

Como puedes ver, cuando se escribe `a = b`, **los valores de `a` y `b` no quedan ligados**. Pero al asignar cualquier valor (tanto literal como de una expresión) a una variable hay que tener cuidado. Volviendo a la metáfora de las cajas, en una caja caben objetos de determinadas formas y tamaños, si un dato es, por ejemplo, un número decimal (ya sea éste *float* o *double*) al ser asignado a una variable entera, perderá su parte decimal. Pero hay más, si aplicas la lógica, ¿qué pasaría si asignas un número como 1203 a un `char`? Si acudes a la tabla 1: Tipos básicos de C, verás que 1203 está fuera del rango del `char`. Bien, lo que pasa es que no sabes lo que pasa. La manera elegante de decir esto es: «comportamiento no definido» o «*undefined behaviour*». El siguiente programa es un ejemplo de esto:



```
1 #include <stdio.h>
2 int main(void)
3 {
4     char c = 1500;
5     short s = 5555555;
6     float f = 3.8e105;
7
8     printf("c: %d\n", c);
9     printf("s: %hd\n", s);
10    printf("f: %f\n", f);
11 }
```

Programa 5: Asignaciones inválidas

Si lo compilas, el compilador te dará una serie de mensajes llamados *warnings*. Advertencias de que algo que has escrito, si bien es «correcto», no tiene pinta de «estar bien». Si por ejemplo escribieras en español: «¡Mas me duele a mí!», aunque es una frase correcta, es probable que quisieras escribir «¡Más me duele a mí!», y no significan lo mismo. Si compilamos y ejecutamos, en la terminal debería aparecer algo así:

```
$ gcc -o main.exe main.c
main.c: In function 'main':
main.c:86:14: warning: overflow in conversion from 'int' to 'char'
      changes value from '1500' to '-36' [-Woverflow]
   86 |     char c = 1500;
      |             ^~~~
main.c:87:15: warning: overflow in conversion from 'int' to 'short int'
      changes value from '5555555' to '-15005' [-Woverflow]
   87 |     short s = 5555555;
      |             ^~~~~~
$ ./main.exe
c: -36
s: -15005
f: inf
```

Como puedes ver, ni el `char` vale 1.500 (porque no puede), ni el `short` vale 5.555.555, porque tampoco puede. Pero el compilador no dice nada sobre el `float`, al que le hemos asignado un literal fuera de rango. El motivo es que los `float` y `double` en C tienen un valor especial llamado *infinity*, que simboliza el infinito. Esto es porque, como veremos a continuación, no representan los números de manera totalmente correcta, y por eso pueden valer más y menos infinito.

Detengámonos en el problema de los números decimales. Si sabes cómo funciona el código binario, sabrás que en un número binario de n bit se pueden representar 2^n valores. En el caso de los números decimales, usamos un sistema complejo denominado IEEE 754. No voy a entrar en detalles, pero el principal problema de esta manera de representar los números decimales es que no sólo no es exacta (por ejemplo, el número 0,1 no se puede representar exactamente) sino que además su precisión **varía** según el rango en que esté el número representado. ¿Qué quiere decir esto? Que si cerca del cero los `float` pueden distinguir entre 1,10 y 1,11, a lo mejor no pueden distinguir entre 1000000,10 y 1000000,11. Ten esto en cuenta si haces programas que utilicen números decimales.

Como has visto aquí, hay asignaciones permitidas y prohibidas entre tipos, las permitidas (las que el compilador no ve como algo malo), se llaman conversiones implícitas, su nombre viene de que tú no tienes que hacer nada para que ocurran, por ejemplo, como hemos visto, asignando un `char` a un entero. Posteriormente te enseñaré a hacer conversiones entre tipos explícitamente.



4.2. Imprimir cosas

Entre programadores se llama imprimir a escribir cosas en ficheros y, especialmente, por pantalla, como tu primer programa que escribía en pantalla «¡Hola, mundo!». No quiero adelantar más de lo necesario, porque detrás de lo que usamos para imprimir por pantalla hay varios conceptos, pero necesito que sepas imprimir cosas para que puedas probar tus propios programas.

Para imprimir algo en pantalla debes usar la palabra `printf`. Con una sintaxis que es un poco complicada. Debes poner `printf`, un paréntesis y una cosa llamada «formato». El formato es el texto que se va a imprimir, rodeado de comillas dobles ("). Para incluir variables en esa impresión, debes poner los llamados «especificadores», que son textos especiales que indican **el tipo** de las variables que quieres imprimir. Después del formato, pondremos las variables que vayamos a imprimir, separadas por comas, en el orden en que pusimos sus especificadores. Además, hay ciertos caracteres que se deben poner de manera especial, los saltos de línea y los tabuladores. Esto es un poco confuso, así que te dejo dos tablas donde puedes ver los especificadores y los caracteres especiales.

Especificador	Tipo que imprime
%d	Enteros (int)
%f	float
%lf	double
%hd	short
%c	char como letras (no números)
%s	Texto, escrito como "Un texto", comillas incluidas.
%p	Punteros, son una cosa avanzada y la explicaré varias secciones más adelante

Tabla 2: Especificadores de formato en C

Por otro lado, los caracteres especiales son éstos, y se ponen con una barra inclinada invertida (\) delante. En la tabla incluyo ya la barra inclinada invertida correspondiente. Además, como los especificadores empiezan con el símbolo de porcentaje, si quieres imprimir uno, debes ponerlo dos veces.

Secuencia	Carácter impreso
\\	Barra inclinada inversa
\n	Nueva línea
\t	Tabulador (imprime espacios hasta que se alinea a una columna de 4 espacios)
%%	Imprime sólo un símbolo de porcentaje

Tabla 3: Imprimir caracteres especiales en C

Esto es muy pedregoso, así que vamos a ver un ejemplo.



```
1 #include <stdio.h>
2 int main(void)
3 {
4     int entero = 654654;
5     short shorty = 25254;
6     char charty = 'a';
7     double decimal = 2.3;
8     printf("El entero vale:\t%d\nEl short vale:\t%d\nEl char es la
9         letra:\t%c\nEl decimal vale:\t%f\n", entero, shorty, charty,
            decimal);
}
```

Programa 6: Ejemplo de impresión.

La línea es muy larga y en esta página se escribe como varias, pero tú deberías escribirlo todo como una sola línea. (Si te fijas, sólo hay un número, eso indica que es la misma línea pero partida para que quepa) El programa, una vez compilado, debería arrojar este resultado:

```
$ ./main.exe
El entero vale: 654654
El short vale: 25254
El char es la letra: a
El decimal vale: 2.300000
```

Corolario final: `printf` no añade nada que tú no le pongas, incluidos saltos de línea, así que si quieres imprimir lo siguiente en una línea distinta, acuérdate de poner `\n` al final del formato. Además, siempre es recomendable que lo último que imprima tu programa sea un salto de línea, si no, podría dejarse cosas sin imprimir, porque la terminal se fuerza a imprimir las cosas cuando recibe una línea nueva.

4.3. Operadores

Jugar a los trileros con los valores de las variables que declaras en tu programa es... aburrido, lo sé, por ello, vamos a aprender a hacer operaciones con ellas. En C (y en cualquier lenguaje de programación) existen los llamados **operadores**. Son símbolos que nos permiten realizar un cálculo. Los operadores son un concepto matemático, y se aplican a una serie de argumentos, o, mejor dicho, operandos. En matemáticas, los símbolos `+`, `-`, `*`, y `/` son operadores para la suma, la resta, la multiplicación y la división, respectivamente. Del mismo modo que hemos hecho con los tipos básicos, te los voy a presentar en una tabla y después veremos ejemplos de cómo se usan.



Operador	Descripción
+	Suma. Suma tanto números enteros como decimales y entre ellos.
-	Resta. Sustraer del operando de a su izquierda el valor del operando a su derecha.
/	División. Devuelve el resultado de la división del operando izquierdo entre el derecho. Nótese que, si ambos operandos son enteros , el operador realiza la división entera, es decir, sin decimales .
*	Multiplicación. El asterisco hace muchas cosas en C, pero de momento esta la primera de sus funciones que descubrirás. El resultado de una multiplicación es un double .
++	Incremento. Hace que un número (decimal o no) aumente en una unidad. Puede ser prefijo (delante del número) o postfijo (detrás del número).
--	Decremento. Funciona como el incremento, pero disminuye el número en una unidad.
%	Módulo. Es un operador que devuelve el resto (o residuo) de la división entera entre el operando izquierdo y el derecho.

Tabla 4: Operadores matemáticos básicos

En la sección anterior te dije que a una variable le podíamos asignar un valor. También dije que una expresión es un fragmento de código C con un valor, y que el nombre de una variable, sola, es una expresión. Ahora que tenemos operadores podemos realizar expresiones más complejas, por ejemplo, $a+b$ sería una expresión cuyo valor sería la suma de a y b . ¡Ya podemos hacer cálculos!

Hay que tener cuidado, porque, como ya te dije, toda expresión en C tiene **un tipo**, y, como vimos en la sección anterior, asignar un valor de un tipo incorrecto a una variable resulta en errores. Por ejemplo, el operador de la división se comporta distintamente si sus operandos (los números divididos) son enteros o decimales. Veamos un ejemplo:

```
1 #include <stdio.h>
2 int main(void)
3 {
4     double d1 = 1/3;
5     printf("d1: %f\n", d1);
6
7     double d2 = 1.0/3;
8     printf("d2: %f\n", d2);
9 }
```

Programa 7: División entera contra división decimal

Si compilas y ejecutas el programa verás que sale este resultado:

```
$ ./main.exe
d1: 0.000000
d2: 0.333333
```

Lo que parecen operaciones iguales, dan resultados distintos, y esto es porque el **tipo** de los operandos es distinto. En C, un valor literal entero es un `int`, y el operador división, cuando opera dos enteros, tiene tipo entero. Sin embargo, cuando **cualquiera de los dos** operandos es decimal, realiza la operación decimal, y su tipo es `double`.



Los operadores ++ y -- son especiales, porque son operadores unarios. Un operador unario es aquél que sólo se aplica a un operando. Por ejemplo, en matemáticas existe el operador raíz cuadrada, denotado por el símbolo $\sqrt{\quad}$ que, aplicado a un único número, nos da el número (o números) que elevado al cuadrado nos dan el operando. Los operadores incremento y decremento son unarios, y, además, se pueden escribir delante o detrás de su operando. Estos operadores son especiales porque no sólo dan un valor, sino que **cambian el valor del operando al que se aplican**. Simplemente: si a vale tres y realizamos a++, a pasará a valer cuatro, pero si asignamos el valor de la operación a otra variable, esa variable tendrá un valor determinado. Veámoslo en código.

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int a = 3;
5     int b = ++a;
6     printf("a: %d; b: %d\n", a, b);
7     a = 3;
8     int c = a++;
9     printf("a: %d; c: %d\n", a, c);
10 }
```

Programa 8: Operadores de incremento y decremento

Si lo ejecutas saldrá lo siguiente:

```
$ ./main.exe
a: 4; b: 4
a: 4; c: 3
```

Quiero que entiendas bien lo que ocurre aquí: siempre que apliquemos el operador incremento a una variable, ésta aumentará de valor en una unidad. No obstante; dependiendo de si lo hacemos prefijo o postfijo, el valor **de la expresión** cambia. En el caso del operador prefijo, el valor de la expresión es el valor de la variable **con el incremento aplicado** y si lo hacemos postfijo, ese valor es el de la variable **antes de aplicar el operador**. Te preguntarás qué necesidad hay de que haya un operador para esto. Bueno, existe porque contar, tanto hacia delante como hacia atrás, es una operación muy común. Así que para ahorrarnos líneas, en vez de escribir `int b = a; a = a + 1;` podemos escribirlo como ya has visto.

Siguiendo la línea de los operadores incremento y decremento, también hay operadores que aúnan la asignación con alguna operación matemática, es decir, sustituir `a = a * 3;` por `a *= 3;`. Existen los mismos operadores para suma, resta, división y módulo.

Finalmente, decirte que la **prioridad** de las operaciones es la misma que en matemáticas: primero se ejecutan divisiones y multiplicaciones, después restas y sumas. Las operaciones con la misma prioridad se ejecutan de izquierda a derecha. El módulo está al mismo nivel que la multiplicación, y los operadores de incremento y decremento siempre van los primeros. De todos modos, si tienes dudas, pon paréntesis en lo que quieras que se ejecute primero.



4.3.1. Cástring: conversiones explícitas

Es normal que necesites que un tipo se convierta en otro, por ejemplo, en el caso anterior, si los divisores fueran enteros, tendrías que «forzar» a que uno fuera decimal para conseguir la división decimal. También suele pasar que una variable que es un entero deba ser «metida» en un char porque sabes que es del rango correcto. Para esto existe la herramienta del *casting*, que en inglés es «moldear» como quien vierte metal líquido en un molde. Con él, podemos transformar un tipo en otro. Pero no es la panacea, las leyes de la lógica se siguen aplicando, un char no puede valer más de 127, aunque hagas cástrings. Veamos un ejemplo:

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int a = 3;
5     int b = 2;
6     double result = (double)a / b;
7 }
```

Programa 9: Ejemplo de cástring

En la línea 6 necesitamos que al menos unos de los operadores sea decimal (ya sea float o double). Así que realizamos el cástring a ese tipo del primer operando. La sintaxis del cástring es sencilla, pero voy a explicarla detenidamente: debes poner el nombre del tipo al que quieres hacer cástring entre paréntesis, y justo a su lado la expresión a la que quieres hacérselo. Te dejo algunos ejemplos:

1. `(int)` (`decimal_number / other_decimal`): Aquí estamos haciendo cástring a entero del resultado de una división, como puedes ver, necesitamos encerrar en paréntesis la división para que el cástring se haga a esa expresión y no sólo al primer elemento.
2. `(char)` (`number % 128`) aquí estamos viendo uno de los casos en que hacer cástring a un tipo más pequeño es bueno, si `number` es un entero, al hacerle cástring a char podría haber problemas, pero como hemos hecho módulo a 128, el resultado de la expresión siempre será un número entre 0 y 127, que entra en el rango del char.

A continuación te dejo una tabla de conversiones válidas entre tipos básicos:

		Tipo de destino				
		char	short	int	float	double
Tipo de origen	char	OK	OK	OK	OK	OK
	short	Cástring (desbordamiento)	OK	OK	OK	OK
	int	Cástring (desbordamiento)	Cástring (desbordamiento)	OK	OK (precisión)	OK (precisión)
	float	Cástring (redondeo, desbordamiento)	Cástring (redondeo, desbordamiento)	Cástring (redondeo, desbordamiento)	OK	OK
	double	Cástring (redondeo, desbordamiento)	Cástring (redondeo, desbordamiento)	Cástring (redondeo, desbordamiento)	Cástring (redondeo, desbordamiento)	OK

Tabla 5: Conversiones de tipos en C



Donde pongo «OK» quiero decir que se produce una conversión implícita, pero hay que tener cuidado. El rango del entero es muy grande, y puede ocurrir que la precisión de un `float` o un `double` no llegue a la unidad en los extremos del rango del entero. Te seré sincero, en el rango de los miles de millones al que llega el entero, el `float` (y más el `double`) es perfectamente capaz de contener valores enteros con precisión, pero lo incluyo para que no se te olvide el problema de la precisión. Donde pongo `casting` e indico que puede haber desbordamiento es porque estás convirtiendo un tipo más grande (en bytes) a otro más pequeño, quien debe controlar si eso está bien hecho es el programador, es decir, tú.

4.3.2. Ejemplo final de programa con operadores

Habiendo visto todo esto, podemos hacer nuestro primer programa que haga «algo», voy a darte como ejemplo un programa que calcula la solución a un sistema de ecuaciones lineales, es decir:

$$\begin{cases} ax + by = c \\ dx + ey = f \end{cases}$$

Con un sistema así, podemos aplicar el método de sustitución:

$$(1) \quad ax + by = c \rightarrow x = \frac{c - by}{a}$$

$$(2) \quad dx + ey = f \rightarrow x = \frac{f - ey}{d}$$

$$(1) \text{ y } (2) \rightarrow \frac{c - by}{a} = \frac{f - ey}{d} \rightarrow y = \frac{af - dc}{ae - db} \rightarrow x = \frac{f - e \cdot \frac{af - dc}{ae - db}}{d}$$

```

1  #include <stdio.h>
2  int main(void)
3  {
4      int a = 1;
5      int b = 3;
6      int c = 8;
7      int d = 2;
8      int e = 7;
9      int f = 12;
10
11     double y = (a * f - d * c) / (a * e - d * b);
12     double x = (f - e * y) / (d);
13
14     printf(" %dx+%dy=%d\n", a, b, c);
15     printf(" %dx+%dy=%d\n", d, e, f);
16     printf("x = %f; y = %f\n", x, y);
17 }
```

Programa 10: Cálculo de ecuación lineal



Si copias este programa, verás aplicado lo que he explicado sobre variables y expresiones. Además, el programa cuenta con un problema, sólo resuelve **una** ecuación, para que resuelva otra, o para cambiar algunos de los valores, debemos cambiar el programa y recompilarlo. Eso no es práctico, y los programas de verdad no funcionan así. De momento la mayoría de nuestros programas serán así, porque quiero enseñarte primero otras cosas más fundamentales. Hasta ahora, los programas que hemos escrito se comportan de manera muy *aburrida*, sólo ejecutan una serie de instrucciones una detrás de otra. Pero en la vida real los programas ejecutan instrucciones dependiendo de condiciones, o las ejecutan varias veces, etc.

Otro de los problemas es que no podemos cambiar el comportamiento del programa en determinadas condiciones. Si cambias el valor de los números en el programa de ejemplo para que éste sea irresoluble, el programa fallará. Haz la prueba, cambia los valores de *a*, *b* y *c* a 1 y los de *d*, *e* y *f* a 2. Este sistema tiene infinitas soluciones y hará que el programa falle. Si lo compilas y lo ejecutas con los nuevos valores debería salir algo como esto:

```
$ ./main.exe
1x+1y=1
2x+2y=2
x = -nan; y = -nan
```

¿Qué rayos es nan? Es uno de los valores especiales de los números decimales en C (recuerda la IEEE 754), significa *Not a Number*. Es decir, que el resultado de esa operación que hemos hecho no es un número. ¿Por qué? Porque hemos dividido entre cero. Y si sabes un poco de cálculo, sabrás que un número dividido entre cero es una indeterminación, es decir, no sabemos qué es. Así lo expresa C. Y tenemos suerte, si en vez de división flotante fuera división entera, el programa se vería obligado a cerrar inesperadamente. Sería interesante poder comprobar primero si el sistema tiene solución y después calcularla. Eso se hace con estructuras de control, que es el siguiente capítulo.



5. Alterando el flujo normal del programa

Como he introducido en la sección anterior, es conveniente poder hacer que el programa haga una cosa u otra según una condición. Además, es posible (de hecho es imprescindible) que el programa repita instrucciones basándose en condiciones. Esto se llama alterar el flujo del programa, porque en vez de ejecutar una línea después de la anterior, el ordenador podrá saltar de un sitio a otro del programa, tanto hacia delante como hacia atrás.

5.1. Sentencias condicionales

Las sentencias condicionales son las que nos permiten hacer **diverger** el flujo del programa dependiendo de una condición. Lo vas a entender perfectamente con un diagrama que incluiré a continuación:

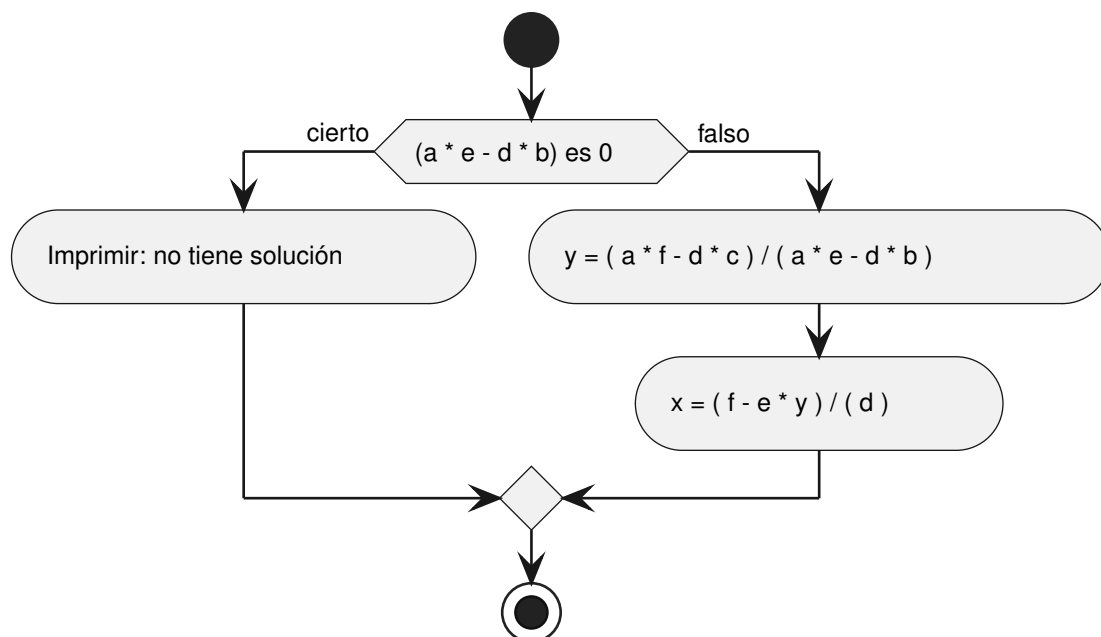


Figura 3: Diagrama de flujo: resolución de sistema de ecuaciones con condicional

Lo que ves se llama diagrama del flujo, pero es muy sencillo de leer, empiezas en el punto negro que está justo arriba, y sigues la flecha, un hexágono significa una **decisión**, según lo que pase en esa decisión, sigue un camino u otro. Si lees el texto dentro de la decisión, verás que precisamente he escrito lo que hace que nuestro sistema sea **irresoluble**. Si lees las etiquetas de las flechas que salen del hexágono, verás que si $ae - db = 0$, imprimiremos que el sistema no tiene solución, y finalizaremos normalmente. Si no es 0, seguiremos haciendo lo que estábamos haciendo hasta ahora.

5.1.1. Operaciones lógicas

Esto del dibujo está estupendo, ahora vamos a ver cómo se hace en C. Pero para eso te tengo que presentar otro conjunto de operadores, llamados operadores booleanos u operadores lógicos. En la sección 1 te hablé de la lógica como ciencia. En concreto vamos a aplicar la lógica proposicional, o de primer orden. En esta lógica tenemos **hechos** que pueden ser únicamente **ciertos** o **falsos**, y que se relacionan con tres operadores. Veamos un ejemplo, y después pasemos a la teoría.



Imagina un sistema antiincendios que funciona de este modo: «Si la temperatura es mayor de 50 °C, saltarán los aspersores, si la temperatura es inferior, pero se detecta humo, también saltarán». Como esto es una ciencia, vamos a escribirlo de manera formal. Cada frase conceptualmente distinta es una **proposición**, y se suelen denotar por letras desde la p en adelante, veamos qué proposiciones tenemos:

1. La temperatura es mayor de 50 °C, la llamaremos p
2. Se detecta humo, la llamaremos q
3. Saltan los aspersores, la llamaremos r

$T > 50\text{ °C}(p)$	Humo(q)	Saltan los aspersores(r)
Cierto	Cierto	Cierto
Cierto	Falso	Cierto
Falso	Cierto	Cierto
Falso	Falso	Falso

Tabla 6: Ejemplo de operación lógica

En lógica hay tres operaciones básicas:

1. La conjunción, coloquialmente llamada «y» o «and». Se denota con el símbolo \wedge .
2. La disyunción u «or». Se denota con el símbolo \vee .
3. La negación. Se denota por varios símbolos, por ejemplo: \sim y \neg , pero también se indica poniendo una barra encima de la expresión negada, por ejemplo: \bar{p} .

Las dos primeras son operaciones binarias y la última es unaria. Se definen por una «tabla de la verdad», que es una tabla que, dados los valores de la o las proposiciones operandos, te da el valor de la operación. Te presento las tablas a continuación:

p	q	$p \wedge q$
Falso	Falso	Falso
Falso	Cierto	Falso
Cierto	Falso	Falso
Cierto	Cierto	Cierto

p	q	$p \vee q$
Falso	Falso	Falso
Falso	Cierto	Cierto
Cierto	Falso	Cierto
Cierto	Cierto	Cierto

p	$\sim p, \neg p, \bar{p}$
Cierto	Falso
Falso	Cierto

Tabla 7: Tablas de la verdad de las operaciones lógicas

El significado conceptual de las operaciones es bastante intuitivo, pero te lo explico aquí para mencionar algunos detalles. La conjunción es cierta sólo cuando **ambos** miembros son ciertos, se corresponde con juntar dos proposiciones con «y» en español, como ya mencioné, por eso a veces se la llama así. «Está lloviendo y hace frío» sólo es cierto cuando llueve al mismo tiempo que hace frío. Por otro lado, la disyunción se corresponde con juntar dos proposiciones con una «o» en español. «Me he roto el tobillo, o me lo he torcido», pero aquí hay un matiz: si miras su tabla de la verdad, verás que cuando **ambos miembros** son ciertos, la disyunción es cierta también. Esto es algo que no se corresponde siempre con el lenguaje hablado, y debes tenerlo en cuenta. Y, finalmente, la negación es cuando antepones «no» a una proposición o a una expresión, «no llueve» es cierto siempre que «llueve» sea falso.

Y después de todo este rodeo lógico digno de Aristóteles, ¿qué? Ya llegamos, en C los operadores lógicos se escriben así:

1. La conjunción se escribe: `&&`



2. La disyunción se escribe `||` (Este símbolo de un símbolo de tubería y en el teclado español se consigue pulsando `alt` derecho y la tecla uno.)
3. La negación se escribe `!`

Si volvemos al ejemplo de los aspersores, vamos a hacer un programa que «simule» ese sistema, te lo escribo a continuación:

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int temperatura_mayor_50 = 1;
5     int humo = 0;
6     int aspersores = temperatura_mayor_50 || humo;
7 }
```

Programa 11: Primer programa con operaciones lógicas

Sí, se parece a los programas aburridos que hacíamos antes, pero no te preocupes, la cuestión es que si vas a la tabla 6: Ejemplo de operación lógica verás que se corresponde con la una disyunción, si hay humo, la temperatura es alta, o las dos a la vez, saltarán los aspersores. Sin embargo; esto sigue siendo algo inútil, no podemos comprobar si la temperatura es mayor o menor que 50, hemos tenido que inventárnoslo. Y esto me lleva a presentarte otro conjunto de operadores, los operadores de comparación. Esto es fácil, en C podemos comprobar si una variable es igual, distinta, mayor o menor que otra. Escribo la tabla y mejoramos el programa que teníamos antes.

Operador	Descripción
<code><</code>	Cierto si el operando izquierdo es menor que el derecho, falso en otro caso.
<code>></code>	Cierto si el operando izquierdo es mayor que el derecho, falso en otro caso.
<code><=</code>	Cierto si el operando izquierdo es menor o igual que el derecho, falso en otro caso.
<code>>=</code>	Cierto si el operando izquierdo es mayor o igual que el derecho, falso en otro caso.
<code>==</code>	Cierto si los valores son iguales, falso en otro caso.
<code>!=</code>	Cierto si los valores son distintos, falso en otro caso.

Tabla 8: Operadores de comparación de C

Hasta ahora, siempre que he hablado de valores lógicos he estado hablando de «cierto» y «falso», en C eso se representa mediante un tipo entero, el que desees. El cero es el valor falso, y **todos** los demás son verdaderos. **En general** el resultado de una operación lógica cierta es 1, como en el código binario, pero a veces no es así, así que asume que siempre que trabajes con tipos enteros como valores lógicos, los valores ciertos pueden ser cualquiera.

Ahora que ya conoces los operadores de comparación y los lógicos, vamos a hacer un programa que compruebe si un número está en el intervalo $(a, b]$, es decir: entre a y b , incluyendo b , pero sin incluir a .



```
1 #include <stdio.h>
2 int main(void)
3 {
4     int number = 5;
5     int a = 0;
6     int b = 10;
7
8     int is_in_interval = number > a && number <= b;
9
10    printf("%d is in (%d, %d] is equal to: %d\n",
11           number, a, b, is_in_interval);
12 }
```

Programa 12: Primer programa con operaciones de comparación

Como puedes ver, no hace falta poner paréntesis, las comparaciones se ejecutan antes que las operaciones lógicas. Aprovecho que ya te he presentado tres conjuntos de operadores (matemáticos, lógicos y de comparación) para decirte que no pasa nada por declarar variables intermedias para recoger valores de parte de la operación si ves que una expresión se vuelve muy grande. Incluso te recomendaría que al principio lo hicieras y después probaras a prescindir de esas variables, como ejercicio inicial, en tus programas.

5.1.2. Bifurcando el programa: el `if`

Ahora que ya sabemos cómo crear condiciones lógicas (proposiciones) que sean ciertas o falsas, podemos crear nuestra primera sentencia condicional. En C, una sentencia condicional se crea con la palabra clave `if`. A continuación te presento la estructura básica de un `if`, pero esto **no** es un programa correcto en C.

```
1 if(/*condición*/)
2 {
3     //Esto se ejecuta sólo si la condición es cierta
4 }
5 else
6 {
7     //Esto se ejecuta sólo si la condición es falsa
8 }
```

Programa 13: Esquema básico de instrucción `if`

¡Ahora podemos mejorar nuestro programa que simula el sistema de aspersores! Vamos a añadir operadores de comparación y lógicos:



```
1 #include <stdio.h>
2 int main(void)
3 {
4     int temperature = 25;
5     int humo = 0;
6
7     if (temperature > 50 || humo)
8     {
9         printf("Aspersores activados.\n");
10    }
11    else
12    {
13        printf("Aspersores apagados.\n");
14    }
15 }
```

Programa 14: Programa del aspersor con operadores lógicos y de comparación

Como nos ha venido pasando, cada vez que cambies algún valor deberás recompilar y ejecutar. Cambia los valores de las variables `humo` y `temperature` para que cambie el resultado del condicional. Finalmente, podemos mejorar nuestro programa de resolución de ecuaciones lineales, antes de hacer cualquiera de las operaciones, debemos comprobar que el divisor no es cero. Para ello simplemente declararemos más variables y después dividiremos por ellas.

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int a = 1;
5     int b = 1;
6     int c = 1;
7     int d = 2;
8     int e = 2;
9     int f = 2;
10
11    double divisor = (a * e - d * b);
12
13    if (divisor == 0 || d == 0)
14    {
15        printf("El sistema es irresoluble.\n");
16    }
17    else
18    {
19        double y = (a * f - d * c) / divisor;
20        double x = (f - e * y) / (d);
21        printf(" %dx+ %dy= %d\n", a, b, c);
22        printf(" %dx+ %dy= %d\n", d, e, f);
23        printf("x = %f; y = %f\n", x, y);
24    }
25 }
```

Programa 15: Programa de resolución de ecuaciones lineales con condicionales



Esto ya se parece algo más a un programa «de verdad», podríamos decir que ya estamos metidos en harina. Pero el condicional aún tiene más desarrollo: se puede encadenar. A veces queremos comprobar una cadena de condiciones una detrás de otra y ejecutar sólo las instrucciones correspondientes a la primera que se cumpla. Imagina un programa que recibe la temperatura ambiente y, en función de ella, da un mensaje sobre el tiempo, por ejemplo:

1. Si hay 40°C o más, imprimir «Hace mucho calor.»
2. Si hay 35°C o más, imprimir «Hace calor.»
3. Si hay 25°C o más, imprimir «Hace un buen día.»
4. Si hay 10°C o más, imprimir «Hace fresquito.»
5. Si hay menos de 10°C imprimir «Hace mucho frío.»

He escrito las frases así a propósito, para que te des cuenta de que, aunque cuarenta grados es más que 10, no queremos imprimir que hace mucho calor, que hace un buen día y que hace fresquito, sólo la condición que llegue antes. Para eso el lenguaje C ofrece la estructura `else-if`, en que «enganchamos» una cláusula `if` al `else` de otro condicional. Así, sólo si no se ha cumplido la condición anterior, se evaluará esta nueva, y si se cumple se ejecutarán las instrucciones que haya dentro, veámoslos con un ejemplo de código que implementa (implementar llevar a cabo algo, se use mucho como sinónimo de «escribir el código de esta idea») lo que he descrito antes.

```
1 #include <stdio.h>
2 int main(void)
3 {
4
5     int temp = 10;
6     if (temp >= 40) {
7         printf("Hace mucho calor.\n");
8     }
9     else if (temp >= 35) {
10        printf("Hace calor\n");
11    }
12    else if (temp >= 25) {
13        printf("Hace un buen día.\n");
14    }
15    else if (temp >= 10) {
16        printf("Hace fresquito\n");
17    }
18    else {
19        printf("Hace mucho frío\n");
20    }
21 }
```

Programa 16: Programa ejemplo de estructura `if-else`

Como puedes ver, hemos terminado la cadena de `if-else` con un `else`, esto quiere decir que si no se cumple ninguna condición de las anteriores, hará eso, en este caso, imprimir «Hace mucho frío». Pero no es obligatorio, puedes terminar la cadena en un `if-else`.



Ahora que ya conocemos bien los condicionales, podemos revisar el programa que resuelve nuestro sistema de ecuaciones. Si recuerdas, en la última versión (Programa de resolución de ecuaciones lineales con condicionales) dábamos el problema como irresoluble si $d = 0$. Sin embargo; si recuerdas un poco del álgebra que aprendiste en el instituto, y repasas la ecuación, verás que si $d = 0$, no es que el sistema sea irresoluble, es que tenemos que despejar la y , no la x , en ambas ecuaciones para aplicar sustitución. Es decir:

$$(1) \quad ax + by = c \rightarrow y = \frac{c - ax}{b}$$

$$(2) \quad dx + ey = f \rightarrow y = \frac{f - dx}{e}$$

$$(1) \text{ y } (2) \rightarrow \frac{c - ax}{b} = \frac{f - dx}{e} \rightarrow ec - eax = bf - bdx \rightarrow x = \frac{bf - ec}{bd - ea}$$

En caso de que esto no fuera posible, quiere decir que ni la x ni la y están en ambas ecuaciones, es decir, tenemos una ecuación que sirve para despejar x y otra para y . O bien tenemos $\begin{cases} ax = c \\ ey = f \end{cases}$ o tenemos $\begin{cases} by = c \\ dx = f \end{cases}$. Simplemente con comprobar si $a = 0$ ya sabremos qué caso tenemos y podremos dar una solución. Finalmente, podría darse el caso de que tuviéramos sólo una ecuación, si los términos que multiplican a x e y en una de ellas fueran cero, pero en ese caso, esto no es un sistema y es imposible dar valores para x e y . Si implementamos estas condiciones en código, veamos qué ocurre:

```

1  #include <stdio.h>
2  int main(void)
3  {
4      int a = 12, b = 2, c = 10, d = 50, e = 11, f = 17;
5      int irresoluble = 0;
6      double divisor, x, y;
7      if (a != 0 && d != 0) {
8          divisor = (a * e - d * b);
9          if (divisor == 0)
10             {
11                 printf("El sistema es irresoluble.\n");
12                 irresoluble = 1;
13             }
14         else
15             {
16                 y = (a * f - d * c) / divisor;
17                 x = (f - e * y) / (d);
18             }
19     }
20     else if (b != 0 && e != 0) {
21         divisor = (b * d - e * a);
22         if (divisor == 0) {
23             printf("El sistema es irresoluble.\n");
24             irresoluble = 1;
25         }
26         else {
27             x = (b * f - e * c) / divisor;
28             y = (c - a * x) / b;
29         }

```



```
30     }
31     else if (a == 0 && b == 0 || d == 0 && e == 0) {
32         printf("Esto no es un sistema\n");
33         irresoluble = 1;
34     }
35     else {
36         if (a != 0) {
37             x = (double)c / a;
38             y = (double)f / e;
39         }
40         else {
41             x = (double)f / d;
42             y = (double)c / b;
43         }
44     }
45     if (!irresoluble) {
46         printf(" %dx+ %dy= %d\n", a, b, c);
47         printf(" %dx+ %dy= %d\n", d, e, f);
48         printf("x = %f; y = %f\n", x, y);
49     }
50 }
```

Programa 17: Programa de resolución de ecuaciones lineales con condicionales

Vamos a repasar las posibles situaciones en las que podemos estar, son estas:

1. x está presente en las dos ecuaciones. Si este es el caso, simplemente lo resolvemos como lo hicimos la primera vez: aplicamos las fórmulas que surgen de despejar x en ambas ecuaciones, si el divisor de la fórmula para y es cero, no tiene solución.
2. y está presente en ambas ecuaciones, este es el caso cuyas fórmulas he escrito hace poco, por lo que simplemente hacemos lo mismo, y si el divisor de esta fórmula es cero, el problema no tendría solución.
3. Podría pasar que los coeficientes que multiplican a x e y en cualquiera de las ecuaciones fuera cero. Esto no sería un sistema, así que simplemente no lo podemos resolver.
4. Podría ser también que no podamos despejar por ninguna variable en las dos ecuaciones porque una de ellas está presente en la primera ecuación y la otra en la segunda.

El principio del programa es sencillo, simplemente declaro las mismas variables que antes y una más, llamada irresoluble, que es una variable lógica para controlar al final del programa si hemos resuelto el sistema. No hemos visto cómo declarar variables de este modo tan compacto aún, pero quería que el programa cupiese en una página. En el primer condicional comprobamos que x esté presente en ambas ecuaciones, si esto es así, podemos utilizar el primer método, aplicamos las fórmulas que surgieron de despejar por x en ambas ecuaciones. Dentro de esta condición, si el divisor de la fórmula es cero, el sistema es irresoluble y lo marcamos como tal. En caso de que esto no se dé, podría ser que despejáramos por y , y entonces podemos aplicar las fórmulas que escribí al inicio de este ejemplo. Esta parte del programa nos enseña que puedes poner condicionales dentro de condicionales.

Seguidamente, si no estamos en ninguno de los dos casos anteriores, podría darse el caso de que esto no sea un sistema, es decir: en una de las ecuaciones no existen ni x ni y . Si se da este caso, esto no es un sistema y no lo podemos resolver. Finalmente, podría pasar que una incógnita esté en la primera ecuación y la otra en la segunda ecuación. Comprobamos si x está presente en la primera ecuación (es decir, que a no es cero). Si no es así, es el caso contrario, y estaría en la primera solución y x en la segunda. Se resuelve de manera sencilla.



5.2. Bloques de código y alcance

Ahora que ya conocemos la primera de las estructuras de control, debo hablarte de una cosa llamada **bloque de código**, un bloque de código es el fragmento del código fuente que está entre dos llaves ({...}). La principal implicación de «encerrar» código entre llaves es que las variables declaradas en él no son visibles fuera, pero las de fuera sí que son visibles dentro. Si recuerdas la estructura básica del condicional, verás que incluye llaves, además, en el primer programa que te presenté te dije que siempre tendrías que poner una serie de líneas, éstas líneas incluyen un único bloque de código, dentro del cual hemos puesto todas nuestras instrucciones. Pero ahora que hay condicionales, tenemos bloques anidados (unos dentro de otros).

Toda variable declarada en C tiene un **alcance**, que es la porción del código fuente donde la variable puede ser vista, el alcance de una variable es el del bloque de código donde fue declarada y todos los definidos en su interior.

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int exterior = 0;
5     if (exterior == 0) {
6         double inner = 1.3;
7         exterior = 120; // OK: variable de un bloque exterior
8     }
9     inner = 10.3; // Error: variable no definida
10 }
```

Programa 18: Ejemplo de alcance de variables declaradas

Si intentas compilar el código que te acabo de incluir, verás que el compilador dice que la variable `inner` no está declarada, aunque la hayas declarado «antes». Otro de los efectos secundarios es que puedes definir variables que ya existieran en bloques superiores, veamos otro ejemplo:

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int number = 0;
5     int exterior = 3;
6     if (number == 0) {
7         int exterior = 10;
8         printf("%d\n", exterior);
9     }
10    printf("%d\n", exterior);
11 }
```

Programa 19: Ejemplo de redifinición de variable

Si compilas y ejecutas este programa, verás que imprime primero 10 y después 3. Esto es porque hay dos variables con el mismo nombre. ¿Y cómo sabe el programa a cuál te refieres? Simplemente las variables locales (aquellas definidas en el bloque en que estás) tienen prioridad sobre las más exteriores, así que no puedes acceder a la variable definida fuera desde el bloque interno si en este has definido una con el mismo nombre.



Y ahora que ya sabes lo que es el alcance, vamos a explicar un poco de esas cosas que no te había podido explicar antes. Lo primero es decirte que fuera de las llaves de nuestro primer programa puedes escribir cosas. En concreto, de momento, te voy a decir que se pueden declarar variables, que se llaman **variables globales**. Se llaman así porque, al estar declaradas fuera de todos los alcances, tienen un alcance «global». Es decir, cualquier instrucción de tu programa tiene acceso a ellas (salvo que hayas declarado otra con el mismo nombre como hemos visto antes). No me voy a detener mucho más en ellas porque, de momento, no nos son útiles, pero te voy a poner un ejemplo rápidamente de cómo luciría un programa con una variable global.

```
1 #include <stdio.h>
2
3 int globalVariable = 20;
4
5 int main(void)
6 {
7     printf("%d\n", globalVariable);
8 }
```

Programa 20: Ejemplo de programa con una variable global

La regla para la declaración de variables es que deben declararse al inicio de su bloque. Esto no es obligatorio, es una buena práctica que se suele utilizar en el lenguaje C, aunque hay quien dice que no es necesario. Lo dejo a tu discreción. Además, debes declararlas en el bloque más interno que te sea posible. Por ejemplo, en el programa 15 hemos declarado las variables `x` y `y` en el único bloque en que nos hacían falta, podríamos haberlas declarado al inicio del programa, o como globales, pero como no hacía falta, no lo hemos hecho.

5.3. Otras instrucciones de salto: `switch` y `goto`

Ya sabes la principal manera de bifurcar el flujo del programa, pero quedan dos que tienen cierta utilidad, ya las he nombrado en el epígrafe: `switch` and `goto`. La primera es una instrucción que se comporta como un distribuidor del flujo del programa, dada una variable, examina su valor y lo compara con una serie de casos, según el caso, salta a esa línea de código. Podríamos imaginar el `switch` como un operario de fábrica que clasifica productos que salen de una línea de montaje, según el estado del producto, hace una cosa u otra. Veamos un ejemplo.



```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int day = 3;
6
7     switch (day) {
8         case 0:
9             printf("Hoy es: lunes\n");
10            break;
11         case 1:
12             printf("Hoy es: martes\n");
13            break;
14         case 2:
15             printf("Hoy es: miércoles\n");
16            break;
17         case 3:
18             printf("Hoy es: jueves\n");
19            break;
20         case 4:
21             printf("Hoy es: viernes\n");
22            break;
23         case 5:
24             printf("Hoy es: sábado\n");
25            break;
26         case 6:
27             printf("Hoy es: domingo\n");
28            break;
29         default:
30             printf(";Ese número no es ningún día!\n");
31            break;
32     }
33 }
```

Programa 21: Ejemplo de programa con un switch

Si miras el programa de ejemplo, un switch empieza de manera similar a un `if`, pero dentro de los paréntesis no va una condición, sino una variable. Esa variable tiene que ser de un tipo entero. Después, en el cuerpo del `switch` hay una serie de líneas que empiezan con `case`, después va un valor (no puede ser una variable) y después las instrucciones correspondientes a ese caso. Esas instrucciones acaban o no con la instrucción especial `break`. Ésta impide que las instrucciones siguientes se ejecuten. Eso es un poco confuso, pero lo vas a entender en seguida, cuando se evalúa un `switch`, se **salta** al caso que cumpla la condición, pero después **sigue**. Usamos el `break` para decirle que salga del cuerpo del `switch`. Si compilas y ejecutas el programa como está escrito, imprimirá sólo «Hoy es: jueves», pero si quitas todas las líneas donde pone `break`, imprimirá todos los días después del jueves.



Verás que existe un caso que no es un `case`, sino que se ha creado con la palabra `default`. Esto es porque esta palabra clave nos permite indicar qué pasará si el valor de la variable comprobada en el `switch` no concuerde con ningún caso anterior. En este ejemplo imprimiremos que el número no es un día. Quizás hayas notado, o quizás no, que un `switch` con `break` en todos sus casos es básicamente un `if-else`. Es cierto, un `switch` siempre puede ser sustituido por una cadena de condicionales (si el `switch` tiene `breaks` en todos sus casos), y tanto es así que hay lenguajes donde esta estructura no existe, pero veremos más adelante un patrón de código donde se usa intensivamente. Cuando se produce esta equivalencia entre cadena de `else-if` y un `switch`, el caso `default` es el equivalente a terminar la cadena de condicionales en un `else`.

El `switch` es un ejemplo de salto a una etiqueta. Una etiqueta es algo especial, porque establece un punto en el código al que puedes saltar con o sin una condición, la instrucción para saltar allí es el `goto`, veamos un ejemplo muy simple y, después, explicaré su uso más común y pondré otro código de ejemplo de cómo se utiliza, pero deberá esperar a otra sección.

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hola, este es el inicio del programa.\n");
6
7     goto final;
8
9     printf("Yo soy una línea que no debería salir.\n");
10
11     final:
12     printf("Hemos terminado el programa.\n");
13
14 }
```

Programa 22: Ejemplo de programa con un `goto`

Si compilas y ejecutas esto, verás que imprimir la primera y la última línea, pero no la segunda. Esto es así porque hemos usado la instrucción `goto` para saltar a un punto posterior del programa. El `goto` es una instrucción muy peligrosa, en el sentido de que si en tu programa hay un número ligeramente elevado de ellos, es señal de que o no has pensado bien tu proceso, o te faltan herramientas para programarlo de una manera más apropiada. Así que, hasta que no veamos otras secciones del este manual, te recomiendo que reserves los `goto` como una curiosidad más que como algo que debas usar.

5.4. Repetir instrucciones: bucles

Al inicio de esta sección te expliqué que hacer que el ordenador repita operaciones es algo imprescindible para los programas funcionen, y aquí vamos a aprender cómo: con los bucles, o en inglés: *loops*. Hay dos tipos principales de bucles en C: el `while` y el `for`. Del mismo modo que hicimos con el condicional, vamos a ver primero la estructura del flujo del programa, después cómo se escribe en C y finalmente veremos ejemplos de cómo se utilizaría en algunos programas.



5.4.1. El bucle while

Este es el tipo de bucle más sencillo y por eso lo vamos a explicar primero, básicamente declara una serie de instrucciones que se repetirán mientras se cumpla una condición. Imagínate que quisieras imprimir todos los números de 1 al cien. Podrías escribir 100 líneas que imprimiesen cada número o podrías decirle al ordenador que imprima una variable, le sume uno, y vuelva a imprimirla durante cien veces.

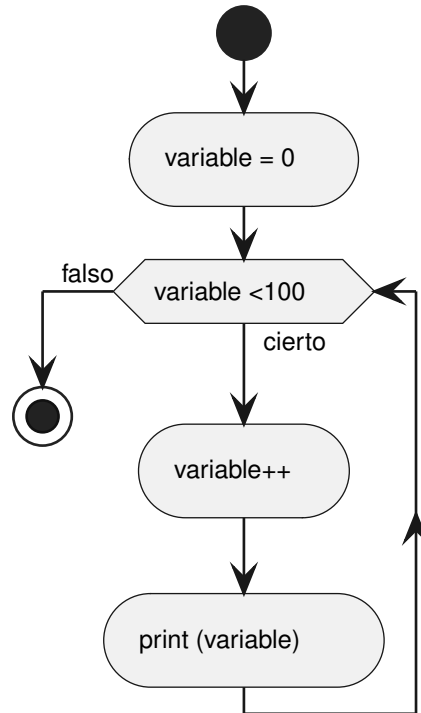


Figura 4: Diagrama de flujo: programa que imprime los números del 1 al 100

Como siempre, a continuación te indico cómo se escribiría ese programa.

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int variable = 0;
5
6     while (variable < 100) {
7         variable++;
8         printf("%d\n", variable);
9     }
10 }
```

Programa 23: Ejemplo de programa con un bucle while

El bucle while es el más simple, como puedes ver, las instrucciones que hay en él se ejecutan sólo si la condición es cierta. Esto tiene dos implicaciones: si la condición **no es cierta** cuando se llega a la instrucción while, el bucle no se ejecutará nunca. La otra implicación es que si no hay nada dentro del bucle while que cambie la condición evaluada, ese bucle se ejecutará infinitamente, por ejemplo, en nuestro programa, dentro del bucle, incrementamos la variable para que se llegue a cien y se salga del bucle.



Hay un tipo especial de bucle `while` que ejecutará las instrucciones siempre **al menos una vez**, porque la primera ejecución la hace antes de evaluar la condición. Veamos un ejemplo:

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int variable;
5
6     do{
7         variable = rand();
8         printf("Variable aleatoria = %d\n", variable);
9     }while(variable != 10);
10 }
```

Programa 24: Ejemplo de programa con un bucle `do-while`

Este programa genera un número aleatorio (tendrás que creerme en esto hasta que te explique otra cosa), lo imprime, y después si ese número no es diez, vuelve a imprimirlo después de cambiar la variable por otro aleatorio. ¿Por qué necesitamos usar el `do-while`? Porque así no necesitamos inicializar la variable fuera del bucle, y todas las iteraciones de las llamadas a `rand` se harán dentro de él. Este bucle es mucho menos usado que los demás, comparte con el `switch` esta cualidad de ser un artefacto del lenguaje usado en patrones de código muy concretos, pero que facilita bastante.

5.4.2. El bucle `for`

El bucle `for` es un bucle que funciona como el bucle `while`, pero que hace dos cosas más: ejecuta una instrucción **antes de empezar el bucle** y otra **al final de cada vuelta**. Veamos un diagrama de flujo y pasemos a ver cómo se escribe:

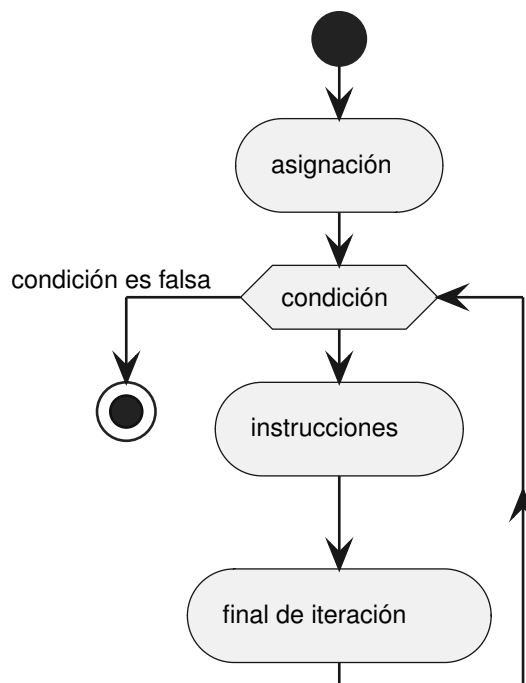


Figura 5: Diagrama de flujo de un bucle `for`



```
1 #include <stdio.h>
2 int main(void)
3 {
4     for (/*Asignación*/; /*Condición*/; /*Final de iteración*/) {
5         /*Instrucciones*/
6     }
7 }
```

Programa 25: Estructura de un bucle for

Donde pongo asignación es porque en esa parte del bucle debe ir una asignación a una variable (o una declaración con asignación). En donde pone condición debe ponerse una condición que se comporta como la condición del bucle `while`, se seguirán ejecutando las operaciones dentro del bucle mientras esa condición sea cierta. Al final de cada «vuelta» (formalmente una vuelta a un bucle se llama iteración, del verbo iterar, que significa, simplemente: repetir) se ejecutará la instrucción que pongas en donde he puesto «final de la iteración». Se lee mucho mejor en un ejemplo.

```
1 #include <stdio.h>
2 int main(void)
3 {
4     for (int ii = 1; ii <= 100; ++ii) {
5         printf("%d\n", ii);
6     }
7 }
```

Programa 26: Ejemplo de programa con un bucle for

Este programa realiza la misma tarea que el ejemplo que pusimos del `while`. Como puedes ver, creamos e inicializamos una variable llamada `ii`, le asignamos el valor uno, ejecutamos el cuerpo del bucle (lo que va entre llaves) y al final de cada vuelta incrementamos la variable en una unidad. Este bucle suele usarse cuando se usa una cosa llamada arrays, que veremos posteriormente.

5.5. Interrupción de bucles

Es relativamente común querer terminar un bucle sin completar una iteración, por ejemplo, escribamos un programa que calcula la potencia de dos que sea mayor o igual que un número objetivo. Además, establecemos una potencia máxima para parar de calcular si la superamos.

```
1 #include <stdio.h>
2 int main(void) {
3     int steps      = 0;
4     int max_steps  = 20;
5     int number     = 1;
6     int target     = 1024;
7
8     while (steps < max_steps && number < target) {
9         steps++;
10        number *= 2;
11    }
12    printf("2 a la %d es mayor o igual que %d\n", steps, target);
13 }
```

Programa 27: Ejemplo de interrupción de un bucle con una variable auxiliar



En este programa utilizamos la variable `steps` como contador de la potencia que estamos calculando. `max_steps` es una variable que simboliza la potencia máxima que queremos calcular, `number` es la potencia calculada en la iteración concreta y `target` el número al que queremos llegar.

```
1 #include <stdio.h>
2 int main(void) {
3     int steps      = 0;
4     int max_steps  = 20;
5     int number     = 1;
6     int target     = 1025;
7
8     while (steps < max_steps) {
9         steps++;
10        number *= 2;
11        if (number >= target) {
12            break;
13        }
14    }
15    printf("2 a la %d es mayor o igual que %d\n", steps, target);
16 }
```

Programa 28: Interrupción de un bucle con la instrucción `break`

En este caso hemos movido parte de la condición a un condicional dentro del cuerpo del bucle. En él, comprobamos si hemos llegado a la condición que nos haría terminar de ejecutar. Nótese que esta condición es la **inversa** a la mitad que hemos retirado antes, si bien antes comprobábamos que el número fuera menor estrictamente que el objetivo (`target`), ahora en este condicional tenemos que `number` sea mayor o igual que `target`, esto es porque la anterior era la condición para **seguir** computando, y este es la condición **para dejar** de hacerlo, de ahí que sea lo contrario. Si se cumple esta condición de parada, ejecutamos `break` y salimos.

Esta manera de escribir programas, es decir: utilizar un `break` dentro de un bucle para terminarlo es algo que muchos programadores desaconsejan, pero como estamos estudiando el lenguaje específicamente y no buenas prácticas de programación, he de ponerte un ejemplo a fin de enseñártelo y, que si lo ves escrito en algún otro código, lo entiendas.

Además: `break` tiene una contraparte, que es `continue`. Esta instrucción se salta **lo que quede de esta iteración**, yendo directamente a la siguiente. Imagina, por ejemplo, que queremos imprimir si los años en un determinado rango de números son bisiestos. Un año es bisiesto si:

1. Es divisible entre cuatro.
2. Pero no entre cien.
3. Pero sí lo es si es divisible entre 400.

Si una de las condiciones no se cumple, no tiene sentido que comprobemos las siguientes, veamos cómo se implementaría sin `continue`.



```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     for(int ii = 0; ii < 2021; ++ii){
6         int bisiestro = ii % 400 == 0 ||
7             (ii % 4 == 0 && !(ii % 100 == 0));
8         if(bisiestro){
9             printf("El año %d es bisiestro.\n", ii);
10        }
11    }
12 }
```

Programa 29: Ejemplo de algoritmo de año bisiestro

Se crea una condición un tanto compleja, pero si lo hacemos con `continue`:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     for(int ii = 0; ii < 2021; ++ii){
6         if(ii % 4 != 0){
7             continue;
8         }
9         if(ii % 100 == 0 && ii % 400 != 0){
10            continue;
11        }
12        printf("El año %d es bisiestro\n", ii);
13    }
14
15
16 }
```

Programa 30: Ejemplo de algoritmo con `continue`

Utiliza más líneas, pero se lee mejor, dado que se podría leer como «si no es divisible entre cuatro, pasa al siguiente número si es divisible entre 100, pero no entre 400, pasa al siguiente también». Como puedes ver, podemos poner la impresión en el bucle sin un condicional porque, si se llega a esa línea, es porque se han cumplido las condiciones para que sea bisiestro.

Este programa, siendo sinceros, está peor escrito que su variante sin `continue`, pero te lo he explicado para que lo conozcas y lo entiendas si lo ves escrito, del mismo modo que pasaba con `break`,



6. Estructuras de datos

Las estructuras de datos son una de las cosas más importantes en programación y en informática, hasta ahora la única manera que tenías almacenar datos eran variables, cada una de un tipo y con un nombre. Esta es la estructura de datos más simple, pero a veces necesitamos estructuras más complejas, en esta sección veremos dos estructuras de datos simples que te proporciona el lenguaje C, la primera es el array y la segunda el *struct*, o estructura, en español.

6.1. El array

A veces queremos hacer paquetes de datos, estos paquetes se llaman *arrays*. Un array es una estructura en que declaramos un espacio para varias variables, a las que referenciaremos por el nombre del array y su posición en él. Es decir, referenciamos los datos en un array por «el quinto elemento del array a». Veamos cómo se declaran y se usan.

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int list_of_numbers[10];
5 }
```

Programa 31: Ejemplo de declaración de array

En el ejemplo hemos declarado un array de diez posiciones de tipo entero. Y aquí viene la primera cosa importante: los array no se empiezan a contar desde el uno, sino desde el cero. Eso quiere decir que un array de diez posiciones como este no tiene una posición diez, sino posiciones del cero al nueve. Para acceder a un elemento del array debemos poner el nombre del mismo y, entre corchetes ([]), la posición a la que queremos acceder. Dentro de los corchetes puede haber cualquier expresión con un valor, incluidas variables, que es lo más común.

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int list_of_numbers[10];
5     list_of_numbers[0] = 30;
6     int a = list_of_numbers[0];
7
8 }
```

Programa 32: Ejemplo de uso de array

En este programa estamos declarando el array, asignándole un valor a su primera posición, y después usando ese valor para asignárselo a una variable. Recuerda que, cuando hacemos esto, del mismo modo que cuando asignamos un valor a una variable, los valores de la primera posición del array y la variable no han quedado ligados. Si recuerdas lo que dije en la última sección, utilizaremos mucho los bucles *for* junto con estas estructuras de datos.



```
1 #include <stdio.h>
2 int main(void)
3 {
4     int list_of_numbers[10];
5     for (int ii = 0; ii < 10; ++ii) {
6         list_of_numbers[ii] = ii;
7     }
8 }
```

Programa 33: Ejemplo de uso de array

Varias consideraciones: primero, la variable que se declara en los bucles `for` cuando se accede a un array suele llamarse `i`. Es una manía personal mía llamarla `ii`, queda a tu discreción, pero si quieres que otros programadores lean tu código cómodamente, recomendaría usar uno de los dos nombres; segundo: mira bien el bucle `for`, `ii` toma valores desde 0 hasta 9, que son las posiciones del array, y les asigna el valor que hay en `ii`, así que tendríamos un array de números con los valores 0, 1... hasta 9.

Pero los arrays pueden tener más de una dimensión, es decir, si ahora tenemos arrays de datos, podemos tener matrices, o cubos, o incluso estructuras de dimensiones ilimitadas. En general, el ser humano tiene dificultad manejando más de dos o tres dimensiones, así que te pondré el ejemplo de cómo usar un array de dos dimensiones.

```
1 #include <stdio.h>
2 int main(void)
3 {
4     int matrix[5][10];
5     for (int i = 0; i < 5; ++i) {
6         for (int j = 0; j < 10; ++j) {
7             matrix[i][j] = 1 + (i * 5 + j);
8         }
9     }
10    for (int i = 0; i < 5; ++i) {
11        for (int j = 0; j < 10; ++j) {
12            printf("%2d ", matrix[i][j]);
13        }
14        printf("\n");
15    }
16 }
```

Programa 34: Ejemplo de uso de array bidimensional

La primera dimensión del array (en este caso 5) indica las filas que éste tendrá, la segunda, las columnas. Puedes pensar en esto como un «array de arrays». Esto es extrapolable a todas las dimensiones, como ya hemos dicho. Debes tener cuidado, si intentas acceder a una posición del array que no existe, lo más probable que es tu programa termine de manera abrupta. Y, si utilizas bucles anidados con arrays de varias dimensiones, la convención es que las variables después de `i` suelen llamarse con letras sucesivas. Esto es una reminiscencia matemática, donde operadores grandes como sumatorios o productorios se indexan por estas letras.



6.2. La estructura

En C un *struct* -o estructura, en español- es un «paquete» de datos de distintos tipos que tienen un nombre. Sería como agrupar una serie de variables y poder referirnos a ellas como un conjunto. ¿Qué ventaja tiene esto? Que podemos crear nuevos tipos juntando los tipos básicos. Si lo piensas, es algo bastante natural, una estructura nos permite utilizar los ladrillos básicos del lenguaje, los tipos básicos, para crear nuevos conceptos. Cuando declaras una estructura estás creando un tipo nuevo. En general, queremos que todas las partes de nuestro programa puedan ver esos tipos, así que la declaración de la estructura se hace fuera de todos los bloques de código, como cuando te expliqué qué eran las variables globales.

Cuando creamos una variable de este tipo, estamos **instanciando** un *struct*. En informática instanciar es crear una variable de un tipo determinado. Por ejemplo, podríamos haber dicho «instancias un entero», pero se suele aplicar a tipos compuestos, no básicos. Así que primero debemos crear el tipo y después variables de ese tipo.

A continuación te presento la sintaxis para declarar un *struct*, para crear una variable de su tipo y para utilizarlo.

```
1 #include <stdio.h>
2 struct my_struct {
3     int field1;
4     char field2;
5     double field3;
6 };
7
8 int main(void)
9 {
10     struct my_struct my_variable;
11
12     my_variable.field1 = 100;
13     my_variable.field2 = 'b';
14     my_variable.field3 = 3.3;
15 }
```

Programa 35: Declaración, instanciación y uso de un *struct*

Los datos que van dentro de una estructura se llaman **campos**, y se accede a ellos con un punto. Si estás pensando que eso hace que el punto sea un operador, estás en lo cierto, el punto es un operador que accede a los **miembros** -o campos- de los *structs*. Nota que para declarar una instancia -una variable de tipo estructura-, no sólo debes poner su nombre sino la palabra reservada *struct* delante.

Veamos un ejemplo de la utilidad de esto. Imagínate que queremos hacer un programa que calcule la distancia entre dos puntos, podríamos hacerlo así:



```
1 #include <stdio.h>
2 #include <math.h>
3 int main(void)
4 {
5     double punto1_x = 1.1;
6     double punto1_y = 3.2;
7     double punto2_x = 2.3;
8     double punto2_y = 5.4;
9
10    double diff_x = punto1_x - punto2_x;
11    double diff_y = punto1_y - punto2_y;
12    double distancia = sqrt(diff_x*diff_x + diff_y*diff_y);
13
14    printf("P1 : [%f, %f]\n", punto1_x, punto1_y);
15    printf("P2 : [%f, %f]\n", punto2_x, punto2_y);
16    printf("Distance: %f\n", distancia);
17 }
```

Programa 36: Ejemplo de resolución de distancia entre puntos

Como puedes ver, el programa funciona y hace lo que tiene que hacer... pero es muy confuso, lo único que «une» las dos coordenadas de un punto es el nombre de la variable, y además hay que declarar muchas variables. Creo que puedes ver que, en cuanto utilicemos algunos puntos más, esto podría convertirse en algo indescifrable. Veamos qué pasa si usamos nuestro flamante conocimiento de las estructuras.



```
1 #include <stdio.h>
2 #include <math.h>
3
4 struct point_s {
5     double x;
6     double y;
7 };
8
9 int main(void)
10 {
11     struct point_s A;
12     struct point_s B;
13
14     A.x = 1.1;
15     A.y = 3.2;
16     B.x = 2.3;
17     B.y = 5.4;
18
19     double diff_x = A.x - B.x;
20     double diff_y = A.y - B.y;
21     double distance = sqrt(diff_x * diff_x + diff_y * diff_y);
22
23     printf("P1 : [%f, %f]\n", A.x, A.y);
24     printf("P1 : [%f, %f]\n", B.x, B.y);
25     printf("Distance: %f\n", distance);
26 }
```

Programa 37: Ejemplo de resolución de distancia entre puntos usando estructuras

Como puedes ver, el programa queda mucho más limpio. Además de que no tengo que confiar en el nombre de la variable para indicar que algo es un punto, sino que puedo usar una restricción del lenguaje.

Nota: Estos dos programas usan una biblioteca -más adelante hablaremos de ellas- y necesitan ser compilados con una opción especial, simplemente agrega `-lm` a la línea que has venido usando para compilar hasta ahora, quedaría así: `gcc -o main.exe main.c -lm` (sin las comillas).

6.3. Listas de inicialización

Tanto los arrays como las estructuras tienen una manera abreviada de inicializarse. Si recuerdas el capítulo sobre variables, inicializar es darle valor por primera vez a una variable. Esta manera abreviada es la lista de inicialización o, simplemente, inicializadores. Veamos un ejemplo:



```
1 #include <stdio.h>
2 struct point_s {
3     double x;
4     double y;
5 };
6 int main(void)
7 {
8     struct point_s punto1 = {1.1, 2.3};
9     int my_array[10] = {1,2,3,4,5,6,7,8,9,10};
10 }
```

Programa 38: Inicialización con llaves

En el caso del *struct*, los miembros del mismo se inicializan por orden, así que el campo *x* de nuestro *struct* sería 1,1 y el campo *y* 2,3. En el caso del array, hemos rellenado todas las posiciones, pero podrías elegir rellenar sólo algunas, el resto se rellenarían a cero. Ten cuidado, porque los inicializadores con llaves como estos sólo valen para inicializar, si a nuestro programa agregaras `punto1 = {2.2, 4.6}` en la línea siguiente el compilador no te dejaría compilar. Lo mismo para el array. Pero con los *struct* podemos hacer una cosa, inicializar sólo los miembros que nos interese, saltándonos algunos, etc. Esto se hace con una sintaxis un poco confusa del lenguaje. Vamos a usar de ejemplo un programa con un *struct* que indica la fecha. Una de sus variables es lógica, indica si es un año bisiesto. Es algo que podemos calcular, pero no es extraño guardar datos calculados de un *struct* en él mismo para no tener que calcularlo todo el rato.

```
1 #include <stdio.h>
2 struct date_s {
3     int isLapYear;
4     char day;
5     char month;
6     short year;
7 };
8 int main(void)
9 {
10     struct date_s moonLanding = {.month=7, .year=1969, .day = 20};
11     moonLanding.isLapYear = moonLanding.year % 4 == 0;
12 }
```

Programa 39: Inicialización con llaves de struct con selección de campos

Como puedes ver, primero hemos inicializado los campos que queríamos y después hemos calculado lo que nos interesaba con ellos. Nota, además, que como hemos indicado en el inicializador el campo al que corresponde cada valor, ni siquiera hemos de preocuparnos del orden. (La regla para saber si un año es bisiesto es más compleja, pero esto es sólo un ejemplo).

Quiero añadir como colorario final que es posible (y común) declarar arrays de *structs* y estructuras con arrays como miembros. Ejemplo:



```
1 #include <stdio.h>
2 struct point_s {
3     double x;
4     double y;
5 };
6 struct triangle_s{
7     struct point_s points[3];
8 };
9 int main(void)
10 {
11     struct point_s points[2] = {{1.1, 2.3}, {4.5, 6.6}};
12     struct triangle_s triangly = {{{1.1, 2.2},{3.3, 4.4}, {5.5, 6.6}}};
13     triangly.points[0].x = 1.6;
14     triangly.points[0].y = 3.4;
15 }
```

Programa 40: Combinación *struct* con array

Presta atención a la lista de inicialización de la línea 12, en ella las llaves más externas sirven para indicar que estás inicializando la estructura *triangly*, después, hay otro nivel de llaves que abarca todo lo que hay dentro de la lista, que simboliza que estás inicializando el campo *points* y, finalmente, para cada elemento de este array de tres puntos, usamos una lista de inicialización de un *struct* punto. Una manera más clara, aunque más larga, de escribirla sería:

```
1 struct triangle_s triangly = { .points = { {.x = 1.1, .y = 2.2},
2                                           {.x = 3.3, .y = 4.4},
3                                           {.x = 5.5, .y = 6.6}} };
```

He de advertirte una cosa, cuando se inicializa un array con una lista de inicialización no puedes indicar su dimensión con una variable, es decir, debes hacerlo con un valor literal. Esto tiene un motivo muy entendible, si yo inicializo un array con una dimensión variable, por ejemplo `int array[var] = {1,2}`; la lista indica que el array tiene, al menos, dos elementos, pero no sabemos cuánto vale *var*, si vale más, no pasa nada, el resto serán cero, pero si vale uno, o incluso cero... ¿a quién le hace caso el lenguaje, a la lista o a la variable? Ten en cuenta que el compilador no puede predecir el valor que ninguna variable tendrá mientras el programa se ejecuta. Pensarás que en programas simples podría, y de hecho es así, pero sería horrible tener una regla como esta que fuera válida sólo cuando el compilador pudiera predecir lo que vale una variable, cosa muy difícil en programas serios, imposible, de hecho. Así que ten esto en cuenta, si quieres un array que tenga un tamaño determinado por una variable, no lo inicialices con una lista. De todos modos, el compilador te advertiría.

6.4. Ejercicios de la sección

Ahora que ya tienes una serie de conocimientos, te propondré un conjunto de ejercicios que requerirán haberlos asimilado.

Ej. 1: Escribe un programa y declara en él una estructura que defina un círculo en dos dimensiones (su centro y su radio). Y haz que el programa declare una variable de ese tipo y calcule su área.

Ej. 2: Haz un programa que, basándose en el *struct* punto presentado en el ejemplo, declare e inicialice un array de ellos y vaya diciendo las direcciones que hay que seguir para ir de uno a otro, por ejemplo, si los puntos fueran: (1, 2), (3, 4), (2, 5), (2, 1) , se imprimiría:

Derecha, Arriba
Izquierda, Arriba
Quieto, Abajo



Ej. 3: Haz un programa que declare un array bidimensional y calcule la suma de sus filas y sus columnas y lo imprima de este modo:

```
1 2 3 = 6
1 4 2 = 7
5 3 4 = 12
-----
7 9 9
```

Ten en cuenta que si pones números de cifras distintas quedarán desalineadas las columnas, esto no tienes que arreglarlo. (Pista: puedes declarar más arrays)

Ej. 4: Haz un programa que haga lo siguiente para los números del uno al cien ambos incluidos: si el número es divisible entre dos, debe imprimirse por pantalla «fizz», si es divisible entre cinco, «buzz», y si es divisible entre los dos, «fizzbuzz», no imprimir nada en otro caso. Te presento un ejemplo de los primeros diez números:

```
fizz
fizz
buzz
fizz
fizz
fizzbuzz
```




7. Funciones

Llegamos a una de las partes más importantes del manual. Como habrás leído en el título de la sección, vamos a ver lo que es una función. En programación, una función es la traducción casi directa de lo que es una función en matemáticas. Empecemos por ahí, pues. En matemáticas una función es un objeto que recibe una serie de argumentos de determinados dominios y da un resultado que pertenece a un codominio (es decir, a un tipo de objetos matemáticos determinados). Formalmente:

$$f(x) : \mathbb{R} \longrightarrow \mathbb{R}$$

$$x \rightarrow x^2$$

Eso de arriba es básicamente una manera muy formal de escribir $f(x) = x^2$ y x es un número real. Hemos definido el dominio (todos los números reales), su codominio (lo mismo) y la transformación que se hace de la entrada. Bien, pues en C una función es lo mismo, es un fragmento de código que recibe una serie de argumentos y «devuelve» un resultado. Pero aún voy a explicártelo de otro modo, una función es una «caja negra» que recibe ingredientes y nos entrega lo que queremos, sin tener por qué saber lo que pasa por dentro.

Después de esta introducción, veamos cómo se declara, define y usa una función en C. La declaración de una función es como sigue:

1. Tipo de retorno de la función, es decir, el codominio, el tipo de dato que la función nos va a devolver. Por ejemplo, `double`.
2. Nombre de la función, al igual que las variables, las funciones han de tener nombre. Por ejemplo, `power`.
3. Paréntesis de apertura.
4. Lista de argumentos con su tipo separados por comas. Por ejemplo: dos enteros llamados `base` y `exponent`.
5. Cierre de paréntesis.
6. Como siempre, final de la línea punto y coma.

Siguiendo los ejemplos que hemos puesto, quedaría tal que:

```
1 double power(int base, int exponent);
```

Programa 41: Declaración de una función en C

Pero esta función aún **no se puede usar**. Porque no la hemos definido. Definir una función es como darle un valor a una variable (aunque sólo se puede hacer una vez). Y es decir **qué** hace la función. Para esto copiamos de nuevo la declaración (sin el punto y coma), y ponemos un bloque de código (abrimos y cerramos llaves) justo después. Es decir, con el ejemplo anterior quedaría:

```
1 double power(int base, int exponent)
2 {
3     //Aquí pondremos instrucciones
4 }
```

Programa 42: Definición de una función en C



Si has estado avisado, habrás visto una similitud entre este último fragmento de código y el programa básico que te presenté al principio del manual. Esto no es casual, cuando ponías `int main(void)`... lo que estabas haciendo era **declarar y definir** una función llamada `main`. Y tú te preguntarás por qué esto es así, pues porque en el sistema operativo Linux (y en la mayoría de sistemas) los programas en C se empiezan llamando a la función `main`. Por eso tuve que hacerte escribir su definición antes de explicarte qué era una función. Finalmente, en cuanto a terminología, la «forma» de una función (el trío de tipo de retorno y argumentos) se llama habitualmente con la palabra en inglés *signature*.

Volvamos a la función `power`, ahora que ya tenemos la función lista para ser escrita, podemos definir su comportamiento. En una función, el bloque de código que va después de la lista de argumentos se llama **cuerpo** de la función. Dentro del cuerpo de la función los argumentos de la lista de argumentos se pueden usar como variables locales. Con ello puedes calcular el valor que quieres que la función **devuelva** y ordenar a C que lo devuelva con la palabra clave `return`. Cuando se usa esta palabra, la función termina (se sale de ella). Veamos cómo podemos implementar la función de ejemplo que, si no lo has adivinado por nombre, calculará la potencia de base elevado a `exponent`. Quedaría de este modo:

```
1 double power(int base, int exponent)
2 {
3     double res = 1;
4     int ii = 0;
5     while (ii != exponent) {
6         if (exponent < 0) {
7             res /= base;
8             ii--;
9         }
10        else {
11            res *= base;
12            ii++;
13        }
14    }
15    return res;
16 }
```

Programa 43: Ejemplo de función en C

La función hace lo siguiente: declara una variable llamada `res` que será donde guardemos la potencia calculada. Después, utilizamos un bucle `while` para multiplicar o dividir (según la potencia sea positiva o negativa) tantas veces como haga falta. Al final, devolvemos (también se dice retornar, por la palabra `return`) el resultado.

Bien, ahora que ya hemos definido la función veamos cómo se usa. Para usar una función ésta se «invoca». Para invocarla simplemente pones su nombre y los argumentos que necesita. Veamos un ejemplo, suponiendo que más atrás en el programa tenemos definida nuestra función `power`, este fragmento de código imprimirá varias potencias. Si lo piensas, utilizar una función en programación es como escribir un valor de la misma en lenguaje matemático, si $f(x) = x^2$ entonces sabes que escribir $f(3)$ es lo mismo que escribir nueve.



```
1 #include <stdio.h>
2 // Pega aquí la definición de power
3 int main(void)
4 {
5     double powers[3];
6     powers[0] = power(2, 10);
7     powers[1] = power(2, 0);
8     powers[2] = power(10, -3);
9     for(int ii = 0; ii < 3; ++ii){
10         printf("%f\n", powers[ii]);
11     }
12 }
```

Programa 44: Invocación de función en C

Como puedes ver, se usan como valores que crean los operadores, se pueden guardar en arrays o en otras variables. Y ya que estamos, terminemos de explicar otra cosa, `printf`, como ya habrás podido adivinar, es una función. Lo que pasa con ella es que es una función especial, su primer argumento (el formato) es de un tipo que aún no te he explicado llamado «puntero». Y además `printf` es un tipo de función especial que recibe **un número variable de argumentos**. De momento, todas las funciones que hagamos nosotros tendrán un número de argumentos fijo, que es lo más habitual.

Y, por último, la palabra clave `void`. *Void* en inglés significa vacío. Es la palabra que usamos cuando queremos indicar que una función no recibe ningún argumento (como `main`) o que no devuelve ningún valor. Un momento, ¿y para qué vale una función que no devuelve ningún valor? Bien, aunque las funciones en C se parecen mucho a las funciones en matemáticas, no son exactamente iguales, porque las funciones en C pueden manipular otras cosas además de sus argumentos y el valor que devuelven: las variables globales. El ejemplo más claro es la propia `printf`, que hace una cosa que no es devolver un valor, sino que manipula la terminal, que está simbolizada como una variable global de un tipo concreto. Te dije cuando las mencioné por primera vez que éstas no nos eran útiles «aún». Cuando uno tiene varias funciones, a veces necesita utilizar variables globales porque estarán disponibles en todas las funciones, aunque, como ya te dije, no es la mejor de las prácticas.

Finalmente, una precisión: te he dicho que dentro del cuerpo de la función los argumentos se comportan como variables locales, y esto puede haberte llevado a hacerte la pregunta de: si cambio el valor de un argumento de la función, ¿queda esa variable cambiada siempre? No, los argumentos que las funciones reciben son **copias** de los que se les proporcionó cuando se las invocó. Veámoslo con un ejemplo:

```
1 #include <stdio.h>
2
3 void doble(int a){
4     a = a * 2;
5 }
6 int main(void)
7 {
8     int number = 3;
9     printf("%d\n", number);
10    doble(number);
11    printf("%d\n", number);
12 }
```

Programa 45: Demostración de que una función recibe copias de sus argumentos



Si compilas y ejecutas este programa verás ambas veces el programa imprime el número 3. Esto es porque, como te he dicho, lo que la función manipula dentro de sí misma es una copia de `number`.

7.1. Separación entre declaración y definición

Te he explicado cómo declarar y cómo definir una función por separado, pero en los ejemplos siempre he incluido únicamente la definición. Esto es porque en la definición incluimos la declaración. Hay dos motivos por los que puedes separar ambas cosas, el primero es que quieras separar tu código en varios archivos, cosa que te explicaré más adelante, pero el segundo es que necesites definir todas las funciones porque se usan unas a otras, veamos un ejemplo de dos funciones, cuyo objetivo es imprimir siempre «Temporada de patos» y «Temporada de conejos», dependiendo de a cuál llames primero, el orden será uno u otro. El código sería el siguiente:

```
1 #include <stdio.h>
2
3 void patos(void){
4     printf("Temporada de patos\n");
5     conejos();
6 }
7
8 void conejos(void){
9     printf("Temporada de conejos\n");
10    patos();
11 }
12
13 int main(void)
14 {
15     patos();
16 }
```

Programa 46: Declaración no separada de definición

Pero si intentas compilar esto, el compilador dirá que la función `conejos` no está definida cuando la quieres usar dentro de la función `patos`, y, como puedes ver, no lo está, porque está definida más abajo. El programa, sin embargo, funcionaría, pero no es, de nuevo, recomendable, utilizar funciones sin definir las. ¿Cómo solucionamos esto? Poniendo la declaración de ambas funciones **antes** de sus definiciones. Es decir:



```
1 #include <stdio.h>
2
3 void patos(void);
4 void conejos(void);
5
6 void patos(void){
7     printf("Temporada de patos\n");
8     conejos();
9 }
10
11 void conejos(void){
12     printf("Temporada de conejos\n");
13     patos();
14 }
15
16 int main(void)
17 {
18     patos();
19 }
```

Programa 47: Declaración separada de definición

Si ejecutas esto, tu programa se ejecutará para siempre, así que pulsa la tecla control y la letra C a la vez para terminarlo.

7.2. Las funciones y los arrays

Los arrays son una cuestión especial cuando se trata de funciones por dos motivos: cuando a una función le pasas un array, **sí que puedes modificar sus elementos**. Esto es porque los arrays cuando se pasan a una función se convierten en **punteros**. Como ves, ya los he mencionado varias veces. Son uno de los conceptos más centrales de este lenguaje y donde reside su potencia, por lo que lo explicaré más adelante, pero su presencia impregna de manera invisible todo lo que te he contado hasta ahora, como verás cuando lleguemos a ese punto. De momento, quédate con que un array que pases a una función te permite modificar sus elementos, veamos un ejemplo de dos funciones que reciben un array, una modifica los elementos y otra no.



```
1 #include <stdio.h>
2
3 void print_array(int array[], int array_size) {
4     for(int ii = 0; ii < array_size; ++ii){
5         printf("%d ", array[ii]);
6     }
7     printf("\n");
8 }
9
10 void add_one_to_each(int array[], int array_size){
11     for(int ii = 0; ii < array_size; ++ii){
12         array[ii]++;
13     }
14 }
15
16 int main(void)
17 {
18     int my_array[] = {1,2,3,4,5,6,7,8};
19     print_array(my_array, 8);
20     add_one_to_each(my_array, 8);
21     print_array(my_array, 8);
22 }
```

Programa 48: Uso de arrays en funciones

Como puedes ver, si ejecutas esto, primero imprimirás el array original y después el array con todos sus elementos incrementados en uno. Esto te demuestra que los elementos de un array se pueden modificar en funciones. Quizás te preguntes si las estructuras también se pueden modificar. **No**, no puedes modificar los campos de una estructura cuando se la pasas a una función. Quiero que veas también que para pasarle a una función un array, en la lista de argumentos debes escribir tipo nombre[], por ejemplo, en ambas funciones del último fragmento `int array[]`. Como otra precisión, una función **no puede devolver un array**, esto es por causas que te explicaré cuando entremos en profundidad en el tema de los punteros. Finalmente, a una función no se le pueden pasar arrays de dos dimensiones, por motivos similares.

7.3. Ejercicios de la sección

En esta sección te pediré que escribas varias funciones, cabe mencionar que aunque no se pide, es recomendable, para comprobar que lo has hecho bien, que **pruebes** estas funciones llamándolas en tu función `main` con valores imprimiendo los resultados si fuera necesario.

Ej. 5: Escribe una función que calcule si un número es primo o no. Nota: un número es primo si sólo es divisible por sí mismo y por la unidad; el uno no es primo ni compuesto (no primo).

Ej. 6: Escribe una función que calcule la distancia entre dos estructuras punto de las usadas en la sección anterior. Para calcular la raíz cuadrada debes usar la función `sqrt`, para poder usarla debes incluir al principio de tu programa (justo debajo de `#include <stdio.h>`) la línea `#include <math.h>` y agregar `-lm` al comando de compilación, quedando
`gcc -o main.elf main.c -lm`.

Ej. 7: Escribe una función que reciba un array de enteros y un caracter separador que imprima los elementos del array separados por ese caracter. Por ejemplo, para el array `{1,2,3,4}` y el carácter `'\n'`, imprimiría:



1
2
3
4

Ej. 8: Escribe una función que encapsule el programa 17: Programa de resolución de ecuaciones lineales con condicionales. La función debe recibir los coeficientes de las ecuaciones (a, b, c, d, e y f). Puede recibirlos por separado o en un array. Para devolver el resultado puedes crear una estructura que simplemente tenga dos `double`.

Ej. 9: Escribe una función que normalice los elementos de un array de `double`. Normalizar es poner todos los elementos en términos de la unidad, así que para normalizarlo deberás dividir todos los elementos por el elemento más grande.



8. La memoria

Llega una de las secciones más importantes del manual. Aunque éste es sobre programación en C, es muy difícil, si no imposible, programar en C de manera solvente sin entender cómo funciona, al menos en parte, la memoria de un ordenador. Hasta ahora te he dicho que las variables que declares, tanto de tipos básicos como de arrays o estructuras se almacenan en «memoria», pero, ¿qué es exactamente la memoria de un ordenador? Bueno, empecemos por lo más prosaico, cómo se ve dicha memoria:

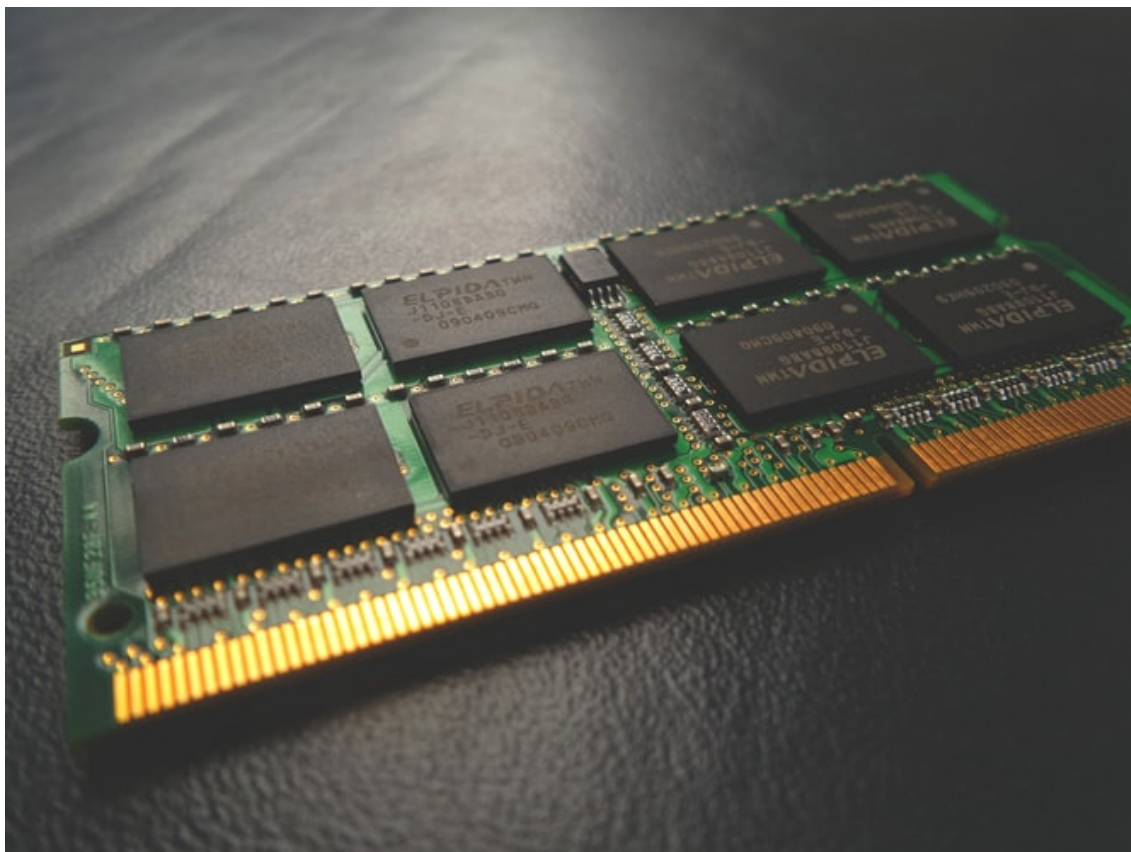


Figura 6: Módulo de memoria RAM

Aquí puedes ver un módulo de memoria de un ordenador. Este módulo es una placa de circuitos con chips soldados encima. Los chips son los rectángulos negros. En ellos se guarda la información en código binario. Una memoria es un sistema al que le pides una porción de información y ésta deposita el valor de esos datos en una serie de «cables» que hay en el ordenador, desde los cuales esos valores viajan a la CPU, al procesador, donde éste los utiliza en los cálculos oportunos. Para poder pedir los datos, la memoria está **direccionada**. Esto quiere decir que cada porción de la memoria está referenciada por un número.

Utilizando una analogía, imagina que la memoria fuera un cuaderno con una cuadrícula, en cada cuadrado digamos que cabe una cifra o una letra, para poder rellenar o leer el cuaderno, lo que haremos es numerar las cuadrículas. Empezaremos con la cuadrícula cero, después la uno, y así sucesivamente.



f	a	3	7	J	n	c	C	H	r	y	D	s	b	z
a	y	y	4	c	B	W	D	x	S	2	2	J	j	z
h	M	a	i	R	r	V	4	1	m	t	z	x	l	Y
v	K	W	r	O	7	2	t	K	0	L	K	0	e	1
z	L	O	Z	2	n	O	X	p	P	l	h	M	F	S
v	8	k	P	0	7	U	2	0	o	0	J	9	0	x
A	0	G	W	X	l	l	w	o	7	J	4	o	g	H
F	Z	Q	x	w	Q	2	R	Q	0	D	R	J	K	R
E	T	P	V	z	x	l	F	r	X	L	8	b	7	m
t	K	L	H	l	G	h	l	h	5	J	u	W	c	F

Tabla 9: Ejemplo de una cuadrícula con valores

En la tabla anterior puedes ver una simulación de este cuaderno, imagina que te digo: «dime qué está escrito en la casilla 24». Teniendo en cuenta que es una tabla de 15 columnas y que **empezamos a contar desde el 0**, tendrías que decirme «S», porque la **dirección 24** se corresponde con la novena posición de la segunda fila. En general, las memorias de los ordenadores están **direccionadas por bytes**, esto quiere decir que a cada byte le corresponde un número, una dirección. Quizás te hayas percatado de que es absurdo pintar esto en una tabla si estamos asignando simplemente números, debería ser una lista continua, una tabla de una columna. Si has pensado eso, tienes razón, porque es así como se suele representar la memoria.

En resumen: la memoria de los ordenadores es una sucesión continua de bytes numerados desde el 0 en adelante, a los que podemos referenciar por ese número tanto para leer como para escribir en ellos.

8.1. Sistemas numéricos posicionales: decimal, binario y hexadecimal

Como ya dijimos en la introducción, el ordenador sólo entiende código binario, y esto es aplicable tanto a las direcciones de la memoria como a su contenido. Debido a esto, me veo compelido, pues, a darte una pequeña clase sobre cómo funciona el código binario. El código binario es un sistema numérico posicional. Un sistema posicional es aquél en que los números se componen de cifras, cada una de las cuales tiene un valor dependiendo de su **posición** dentro del número. Cuanto más a la izquierda están las cifras en un número, más valen. Recuerda cuando aprendiste cómo funcionaban los números: había unidades, decenas, centenas, millares... y así sucesivamente, el sistema que usamos es posicional, y es decimal, porque tenemos 10 cifras para representar números, desde el 0 al 9. Pues en binario funciona igual, pero sólo disponemos de dos cifras.

Un sistema numérico posicional funciona de este modo: si la base numérica (el número de cifras disponibles, en el caso de nuestro sistema, diez) es n , cada cifra de un número vale el valor de la misma cifra multiplicado por n elevado al número de posiciones que queden a la derecha del número. Como siempre, veámoslo con un ejemplo: si escribimos el número 34.789, el cálculo que usamos para saber cuánto vale es el siguiente: (Recuerda que un número elevado a 0 es 1)

$$3 \cdot 10^4 + 4 \cdot 10^3 + 7 \cdot 10^2 + 8 \cdot 10^1 + 9 \cdot 10^0$$

$$3 \cdot 10000 + 4 \cdot 1000 + 7 \cdot 100 + 8 \cdot 10 + 9 \cdot 1$$

De nuevo, podemos expresarlo en decenas, centenas, unidades... que son los nombres que le dimos a esas potencias de 10.

Así que si tuvieras un número binario, harías el mismo cálculo, pero en vez de unidades, decenas, centenas, millares, etc. tendrías unidades, pares, cuartetos, octetos y grupos de 16 (lo siento, no



existe una palabra para esto). Sea el número binario 1110101, el cálculo sería:

$$1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$$

$$1 \cdot 64 + 1 \cdot 32 + 1 \cdot 16 + 0 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 117$$

Ahora que ya sabes leer un número binario, hay que ver cómo se convierte un número decimal a binario. Para eso, se hace lo siguiente:

1. Dividir el número entre 2, calculando el resto.
2. Si la división es exacta, apuntas un cero a la izquierda del número, si tiene resto (que sólo puede ser 1), pones un uno.
3. Repetir hasta que el **resultado** de la división sea 0.

Veamos un ejemplo de esto, sea el número 253, si aplicas el procedimiento, debería quedarte algo como lo de abajo. Si apuntas los restos de las divisiones en la dirección de la flecha inferior podrás componer el número en binario final, en este caso: 11111101.

253	2							
5	126	2						
13	0	63	2					
1	6	0	31	2				
	0	3	1	15	2			
		1	11	1	7	2		
			1		1	3	2	
					1	1	1	2
						1	1	0
1	0	1	1	1	1	1	1	1

←

Si eres usuario de ordenadores, quizás hayas oído expresiones como «este ordenador tiene un procesador de 32 bits» o «este ordenador es compatible con *software* de 64 bits». Ese número de bits es el tamaño de, entre otras cosas, las direcciones de memoria. Un ordenador de 32 bits tiene direcciones de 32 bits, por lo que puede direccionar 2^{32} bytes. Hoy en día la mayoría de ordenadores son de 64 bits, por lo que la mayoría de ordenadores pueden direccionar 2^{64} bytes de información en su memoria.

El problema es que 64 cifras binarias con muchas para leerse de manera sencilla, mira este número binario: 1101001001010101001010100101101010101010110100100111010010101. Es muy largo. Por ello, cuando se expresan direcciones de memoria se utiliza otro sistema numérico. Este sistema numérico es el **hexadecimal**. Es un sistema de base 16, con cifras del 0 a la F. Sí, a la F, has leído bien. Como los números que usamos son de base 10, no tenemos símbolos para representar una cifra que valga 10, 11, 12... y así hasta 15, así que utilizamos letras del alfabeto latino. Por lo demás, funciona igual que el binario o el decimal. Un número hexadecimal se suele representar con «0x» delante para indicar que es ese tipo de número. Veamos un ejemplo, sea el número hexadecimal 0xF2A:

$$0xF2A = 15 \cdot 16^2 + 2 \cdot 16^1 + 10 \cdot 16^0 = 3882$$

Para convertir un número en decimal a hexadecimal hay que hacer lo siguiente:

1. Convertir el número a binario
2. Dividir el número en grupos de cuatro bits, **empezando por la derecha**.
3. Convertir cada grupo de 4 bits a decimal y escribir la cifra hexadecimal correspondiente.



Volvamos al ejemplo del número que convertimos a binario, el 253, en binario es 11111101, si quisiéramos convertirlo a hexadecimal debemos dividirlo en grupos de 4 bits: 1111 1101, el número 1111 es un 15, así que sería F, y el número 1101 es un 13, es decir, D, así que 253 sería igual a 0xFD. Si el grupo más a la izquierda no es de 4 bits, debes añadir ceros a la izquierda. Por cierto, para convertir de hexadecimal a binario, simplemente coge cada dígito hexadecimal y conviértelo de nuevo a su valor en binario, pero, recuerda, cada dígito tiene cuatro bits, así que por ejemplo 0x33 sería 0011 0011, no 1111, que sería 0xF. Ya estás preparado para seguir entendiendo cómo funciona la memoria de un ordenador.

8.2. El mapa de memoria

Una de las maneras más habituales de representar la memoria de un ordenador es con un **mapa de memoria**, que es un dibujo en forma de columna en que se explican los contenidos de la misma, indicando al lado las direcciones relevantes. Mira la siguiente figura:

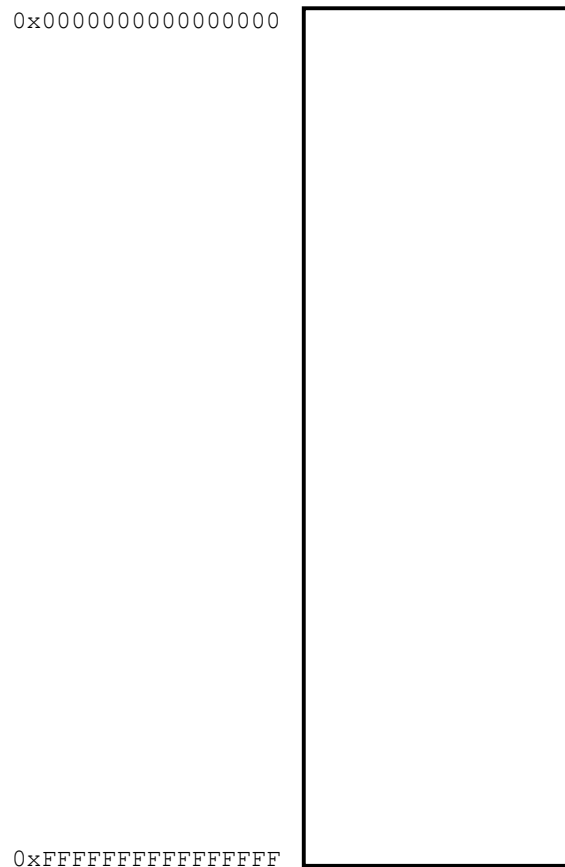


Figura 7: Mapa de memoria vacío

En ella hemos representado toda la memoria del ordenador en una columna, con las direcciones más bajas (cercanas a cero) arriba y las más altas (cercanas a 0xFF...) abajo. En general yo prefiero esta representación, pero en algunas fuentes verás el mapa invertido.



Si tus programas fueran el único *software* que se ejecutara en el ordenador, tendrías disponible el mapa de memoria para ti solo y no tendrías que hacer nada para escribir en memoria, simplemente... hacerlo. Pero éste no es el caso, porque tus programas se ejecutan sobre el sistema operativo. El sistema operativo tiene muchas funciones: se encarga de coordinar los procesos que se ejecutan en la máquina, de manejar el sistema de ficheros, de permitir que el ordenador (la CPU) entienda a los dispositivos... pero una de sus funciones básicas es **la gestión de memoria**. Primeramente: cuando un programa se ejecuta, un proceso es la encarnación real de un programa, cuando se ejecuta un programa, el contenido del programa se carga desde donde está almacenado (tu disco duro), a la memoria RAM y así se crea un **proceso**. Puedes verlo de este modo: un programa es un plano de un coche, y el proceso es un ejemplar real de ese coche que funciona y existe. El SO otorga a los procesos porciones de memoria en la que éstos pueden escribir o no, y **controla** que no se salgan de sus parcelas asignadas.

En la sección en la que hablaba de los arrays te dije que si accedías a una parte de un array que no existía, tu programa terminaría abruptamente, Pruébalo. Haz un programa que tan solo declare un array de, por ejemplo, 10 posiciones, y después escriba algo en la posición 2.500. Verás como el programa lanza un mensaje como éste al ser ejecutado:

```
$ ./main.exe
Segmentation fault (core dumped)
```

Es posible que no lo lance, por cómo maneja el ordenador algo llamado memoria virtual. De todos modos, para un programador en C, la gestión de la memoria (y sobre todo saber que no escribe ni lee de regiones de memoria de las que no debería) es una de las tareas más importantes, si no la que más. En esta tarea, el sistema operativo engaña a nuestros programas. Para tu programa, tienes una memoria utilizable que equivale al mapa completo (los 2^{64} bytes), aunque el ordenador tenga, por ejemplo, 8 GB which is several thousand times less. Por cierto, en términos de informática, cada nivel en los prefijos de multiplicación no es 1.000, sino 2^{10} , 1.024, si fuéramos exhaustivos, deberíamos escribir GiB, MiB y demás, que es como se indican esos múltiplos de 1.024. Lo que hace el sistema operativo es alojar lo que le pidamos en partes de la memoria **física** disponible y darnos direcciones de ese mapa que suponemos que tenemos, y las traduce. Este proceso es la **virtualización de memoria**, y es una de las funciones más importantes de un sistema operativo. Gracias a él, el programador no debe gestionar dónde se pone físicamente lo que él escribe. Además, esto aísla a los procesos, un proceso no tiene ni idea de qué está pasando en el mapa de memoria de otros procesos y ningún otro sabe qué pasa en este.

En este mapa de memoria virtualizado, que es efectivamente el que vamos a usar, el sistema operativo crea una serie de **regiones de memoria**, las cuales están destinadas a alojar distintos tipos de información de cada programa ejecutándose en un ordenador. A continuación te presento un mapa de memoria con las regiones más importantes, y seguidamente te explicaré qué son y por qué nos importan.

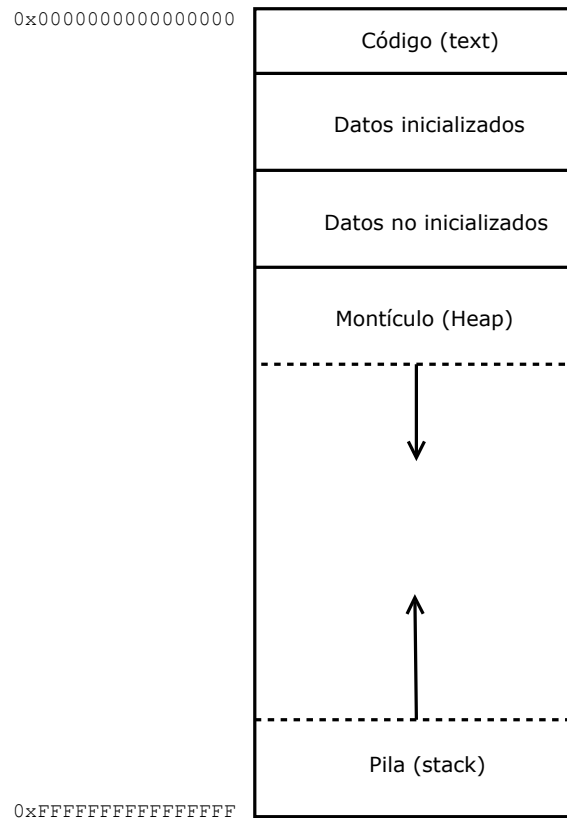


Figura 8: Regiones del mapa de memoria de un proceso

La región de código o texto es sencilla, es donde se almacenan las instrucciones que tu programa va a ejecutar. Cuando te expliqué lo que era la compilación te dije que convertiríamos nuestras instrucciones en un binario que el ordenador podría ejecutar. Pues es en esta sección donde se almacena. Al generar tu programa, esas instrucciones binarias se almacenan en el disco en el ejecutable. Cuando el programa se ejecuta, éstas se cargan en memoria en esta sección. Esta sección no se puede cambiar ni leer dentro del programa.

La siguiente empieza a ser interesante. En esta sección se guarda el valor de variables globales, ya sean de tipos básicos, arrays o estructuras que inicialices (debes declararlas e inicializarlas, la inicialización que hagas en otro momento no cuenta a estos efectos). Esto es así porque esos valores que almacenas en ellas existirán antes de que tu programa empiece a ejecutarse, nada más crear el proceso. En la siguiente sección se guardan las variables, arrays, estructuras, etc. **globales** que no hayas inicializado. ¿Por qué sólo las globales? Porque la función `main` es una función, y las variables que se declaran en las funciones (o en cualquier bloque de código) no se guardan aquí, sino en la siguiente sección: la pila.

Mira el mapa, esta sección está abajo (en las direcciones más altas), pero te la voy a explicar ahora porque es una de las más importantes. En la pila se guardan las variables que se declaran en bloques de código: funciones, bucles, condicionales, etc. ¿Por qué? Porque de esta sección de memoria los datos pueden salir y entrar, o mejor dicho, se puede reutilizar la memoria de datos antiguos para escribir datos nuevos. Para explicarte cómo ocurre esto, primero he de explicarte cómo funciona una pila.



Una pila es una estructura de datos, como los arrays o las estructuras, que funciona de este modo: cuando introduces algo en una pila, este elemento se queda arriba, y cuando sacas algo de la pila, sólo podrás sacar lo que esté en la parte superior. Te voy a poner un ejemplo físico: una lata de las famosas patatas Pringles®. La única patata que puedes sacar es la superior, si quieres sacar más, puedes, pero siempre en el orden inverso al que fueron metidas en la lata. Pues con cualquier pila pasa lo mismo. El modo en que esta pila se aplica a la programación en C es que, cuando se entra en cualquier **bloque de código**, las variables declaradas en él son **añadidas** a la pila. Esto incluye los arrays y estructuras. Cuando se sale de ese bloque de código, esas variables son **extraídas** de la pila, es decir, se pierden porque se entiende que ya fueron usadas. Éste es el motivo por el que las funciones **no pueden devolver arrays**. Porque esos arrays dejan de existir cuando la función deja de ejecutarse.

Quizás te estés preguntando por qué puedes declarar variables de tipos básicos o estructuras en tus funciones y devolverlos. Esto es porque, del mismo modo que los argumentos, el valor devuelto se **copia**. El valor devuelto por una función se deja en la cima de la pila para que se recoja en el punto en que **la función fue llamada**. Ahí es donde tú lo copias a una variable, con el operador de asignación. Con los arrays **no podemos usar el operador de asignación**, no es como se copian arrays. De hecho el compilador lanzaría un error si intentaras asignar un array a otro.

Veamos un ejemplo de cómo se ve la pila de un programa en el caso de ejecutar alguno de los programas de ejemplo que hemos escrito. Vamos a utilizar el programa 48: Uso de arrays en funciones. En este ejemplo vas a ver que en la pila hay dos nombres (`my_array` y `array`), pero recuerda que **apuntan al mismo array**. Sólo copian la dirección donde el array empieza, no sus elementos.



La pila empieza vacía	Entramos en main my_array	Entr. en print_array array array_size my_array
Entr. en el for ii array array_size my_array	Salimos del for array array_size my_array	Sal. de print_array my_array
Entr. en add_one_to_each array array_size my_array	Entr. en el for ii array array_size my_array	Sal. del for array array_size my_array
Sal. de add_one_to_each my_array	Entr. en print_array array array_size my_array	Entr. en el for ii array array_size my_array
Sal. del for array array_size my_array	Sal. de print_array my_array	Sal. de main

Tabla 10: Ejemplo del estado de la pila en una ejecución

Ahora que ya sabes cómo funciona la pila y las implicaciones que tiene que los arrays se guarden en ella, podemos ver la última región y quizás la más importante: el montículo (o *heap*). En ésta región se guarda la memoria que tú le pides al sistema operativo mediante una serie de funciones que vamos a ver. Y pensarás, ¿para qué voy a hacer eso si puedo hacer un array? Sencillo: esta memoria que pides está permanentemente a tu disposición **hasta que la liberas**. Es decir, al contrario que los arrays, debes preocuparte tú de saber cuándo liberar esa memoria. Es una de las tareas más importantes de un programador en C, pero para poder hacer eso, tenemos que aprender primero qué son los punteros.

8.3. Punteros

Ahora que ya sabes que la memoria está direccionada por bytes, debes saber cuál es la manera en que usamos direcciones en el lenguaje C. Lo hacemos con un nuevo (para nosotros) tipo de variable llamado puntero. Un puntero es una variable que guarda una dirección de memoria, y nos permite comunicarle a funciones u otras partes del programa **una región de memoria**. Ya has usado punteros, pero no lo sabías porque he preferido explicar primero el resto de cosas que has visto en secciones anteriores, aunque te he ido avanzando su uso.



Te dije en su momento que cuando a una función se le pasa un array éste decae a tipo puntero. Es decir, como no hacemos una copia de los arrays que le pasamos a las funciones, lo que hacemos es decirle a la función dónde están los elementos del array. Así, si es necesario que haga una copia, podemos hacerla, y si no, los leerá o manipulará directamente.

Si un puntero simboliza una dirección de memoria, puedes pensar que un único tipo que simbolice una dirección sería suficiente, pero no. Cada tipo de dato (tanto básico como estructura) que haya en tu código tiene asociado su tipo puntero, es decir, no hay únicamente punteros, sino punteros a `int`, a `double`, a `char`, a tal o cual *struct*... ¿Por qué es eso así? Porque al usar punteros con tipos asociados, sabemos **qué hay** en la dirección apuntada. Por ejemplo, si tenemos un puntero a `int` que vale `0xFB455DE`, sabemos que debemos coger ese byte y los tres siguientes y decodificarlos como un entero. Veámoslo en código:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int a = 2;
6     int *ptr_to_a;
7     ptr_to_a = &a;
8     printf("a es un entero alojado en %p que vale %d\n", ptr_to_a, a);
9 }
```

Programa 49: Ejemplo de uso de punteros

En la línea 7 declaramos por primera vez una variable puntero, esto se hace con ese asterisco que ves entre el tipo de la variable y su nombre. Es aquí donde se establece que este puntero está asociado a un `int`. En la siguiente línea asignamos a este puntero el valor de la dirección donde está alojada la variable `a`. Para eso usamos el operador `&`. El nombre del operador que más se usa es *ampersand*, en español se llama et, pero este nombre es poco conocido. Dicho operador, delante de una expresión, nos da el puntero de su tipo a donde su valor esté alojado. Ten en cuenta que, para que esto funcione, esa expresión debe estar alojada en algún sitio. Es decir, los valores temporales otorgarían un error, por ejemplo `&(a*2)` lanzaría un error, porque `a*2` no ha sido guardado en ningún sitio aún. Si ejecutas ese programa, la salida será parecida a esto:

```
$ ./main.exe
a es un entero alojado en 0x7ffff5c58e7c que vale 2
```

Entonces, veamos un caso práctico de para qué valen los punteros, por ejemplo: ya hemos dicho muchas veces que una función que recibe un entero lo que hace es recibir una copia. ¿Y si quisiéramos ahorrarnos eso? Si, por ejemplo, quieres una función que multiplique un número por otro, quizás no quieres que se haga una copia, que devuelva el resultado y copiar ese resultado en la variable, simplemente **deja de que la función haga el trabajo**. Si le pasamos el puntero a nuestra variable, será la función quien cambie el valor, vamos a verlo:



```
1 #include <stdio.h>
2 void multiply(int* ptr, int b) {
3     (*ptr) *= b;
4 }
5 int main(void)
6 {
7     int a = 2;
8     int* pointer_to_a;
9     pointer_to_a = &a;
10    printf("a es un entero alojado en %p que vale %d\n", pointer_to_a,
11           a);
12    multiply(pointer_to_a, 4);
13    printf("a es un entero alojado en %p que vale %d\n", pointer_to_a,
14           a);
15 }
```

Programa 50: Declaración de punteros

Aquí puedes ver cómo declaramos una función que recibe un puntero a entero (la variable que queremos multiplicar) y un entero (el número por el que la multiplicaremos). En esta función verás un uso nuevo del operador asterisco (*), que es el de **desreferenciar** un puntero. ¿Qué es eso? Es acceder al valor que ese puntero referencia (de ahí el nombre). Recuerda que, como puntero, `ptr` contiene la dirección de memoria, necesitamos una manera de decirle a C que meta en esa dirección el número multiplicado. Para eso usamos el asterisco delante del puntero. Simplemente: el asterisco convierte el puntero a entero (verblint!) en el entero (verblint!) al que aquél apuntaba. Es decir, «sigue el puntero». Después, usamos el operador `*=` para multiplicar y asignar por el otro argumento. En la línea 11 puedes ver como simplemente llamamos a la función, sin tener que guardar lo que devuelve (porque de hecho la hemos definido como `void` así que no devuelve nada) y nos ahorramos la copia del entero que queríamos multiplicar.

Además del operador asterisco, hay otro operador que se utiliza con punteros que debes conocer, y éste es operador flecha `->`. Sirve para acceder a los campos de un puntero a una estructura. Esto puede parecer un poco confuso, pero lo voy a explicar detenidamente. Imagina que tenemos la estructura punto que escribimos en la sección sobre estructuras. Si, por el motivo que fuera, estuviéramos manejando un puntero a una estructura de este tipo y quisiéramos acceder a sus campos, tendríamos que utilizar el operador asterisco para desreferenciar el puntero y después el punto para acceder al campo. Por ejemplo: sea `point_ptr` un puntero a `struct point_s`, para acceder a su valor `x`, tendríamos que escribir `(*point_ptr).x`. No parece un problema, pero te informo de que es común tener estructuras con punteros a otras que apunten a su vez a otras... puede ser bastante ilegible hacer esto al cabo de tres o cuatro punteros. Por eso existe el operador flecha, para que esto se convierta simplemente en `point_ptr->x`. Quiero que quede claro que este operador accede al campo, no otorga un puntero al mismo. Es decir, siguiendo con el ejemplo, `point_ptr->x` será un `double`, no un `double*`. Algo más adelante veremos este operador en uso.

8.3.1. Aritmética de punteros

Los arrays son en ciertos aspectos (pero **no** todos) equivalentes a punteros, debido a esto, los punteros pueden ser referenciados con el operador corchete. Y de hecho, puedes convertir un array a puntero explícitamente en tus programas. Veamos cómo:



```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int my_array[10] = {0,1,2,3,4,5,6,7,8,9};
6     int* pointer_like_array = my_array;
7     for (int ii = 0; ii < 10; ++ii) {
8         printf("array[%d] = %d\n",ii, my_array[ii]);
9     }
10    puts("=====");
11    for (int ii = 0; ii < 10; ++ii) {
12        printf("array[%d] = %d\n",ii, pointer_like_array[ii]);
13    }
14 }
```

Programa 51: Arrays como punteros

Esto que ves en el programa 51 es lo que pasa sin que te des cuenta cuando una función recibe un array, éste se convierte en un puntero y puedes manejarlo con los mismos operadores de un array. Esto, sin embargo; sólo es válido para arrays de una dimensión, por un motivo que veremos más tarde. En puridad, el operador corchete es un *atajo*. En realidad lo que hace es sumar al puntero y utilizar el asterisco para desreferenciar. Cuando sumas un entero a un puntero entra en juego la aritmética de punteros, veamos un ejemplo y te explico lo que es exactamente.

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int my_array[10] = { 0,1,2,3,4,5,6,7,8,9 };
6     int* pointer_like_array = my_array;
7     for (int ii = 0; ii < 10; ++ii) {
8         printf("En la dirección %p hay un %d\n",
9             pointer_like_array + ii,
10             *(pointer_like_array + ii));
11     }
12
13 }
```

Programa 52: Aritmética de punteros

Si ejecutas el programa verás que las direcciones que imprimen están separadas 4 unidades. Esto es porque cuando a un puntero le sumas un entero, aunque dicho puntero es una dirección de memoria de una memoria direccionada por bytes, al ser un puntero a **entero**, esa expresión de sumarle un número se traduce a sumarle ese número multiplicado por el tamaño de un `int` (cuatro bytes). Después de esto, utilizamos el operador asterisco para que ese puntero al que hemos sumado un entero sea desreferenciado. Utilizar la aritmética de punteros es útil cuando quieres pasar a una función el puntero a una posición de un array. Veamos un ejemplo.



```
1 #include <stdio.h>
2 void multiply(int* number, int other) {
3     *number *= other;
4 }
5
6 void multiply_array(int* array, int array_length, int other) {
7     for (int ii = 0; ii < array_length; ++ii) {
8         multiply(array + ii, other);
9     }
10 }
11
12 void print_array(int array[], int array_size) {
13     for (int ii = 0; ii < array_size; ++ii) {
14         printf(" %d ", array[ii]);
15     }
16     printf("\n");
17 }
18
19 int main(void)
20 {
21     int array[] = { 1,2,3,4,5,6,7,8,9,10 };
22     print_array(array, 10);
23     multiply_array(array, 10, 10);
24     print_array(array, 10);
25 }
```

Programa 53: Ejemplo práctico de aritmética de punteros

Por ejemplo, si utilizamos la función que multiplica un entero sin tener que devolverlo, podemos hacer la versión que haga lo propio con un array (aquí puedes apreciar como las funciones sirven para no duplicar código). Y podemos ver como, usando la aritmética de punteros, no necesitas tener en cuenta tú mismo el tamaño de cada tipo de dato, el lenguaje lo hace por ti. Hay una sintaxis alternativa para esto que a veces verás porque es más compacta, y es utilizar el operador corchete para acceder al elemento y después usar el operador & para conseguir la dirección, haciendo eso, la línea 8 quedaría como sigue `multiply(&array[ii], other);`. En estos casos usar u otra alternativa queda a discreción del programador.

8.3.2. El puntero a char

Por fin voy a desvelar uno de los misterios que más tiempo he estado ocultándote (muy a mi pesar, que conste en acta) sobre los programas que hemos escrito hasta ahora. Este misterio es: qué son los textos que escribimos entre comillas dobles, por ejemplo: `"¡Hola, mundo!"`. Me he saboteado un poco a mí mismo porque lo he puesto en este epígrafe, pero son una manera abreviada de escribir **arrays de char**. Ya sabes que un char es una letra, y que un array es un es una sucesión de datos. La conclusión lógica es que, en C, los textos son arrays a char. Bueno, y si son arrays a char, dónde está su declaración y por qué van por ahí sueltos entre comillas. En resumen: debido a que escribir textos en un programa es muy común, los creadores de C decidieron añadir una **expresión constante** para poder declarar arrays de char temporales donde fuera necesario, esta expresión es, efectivamente, poner el texto entre comillas. Hay expresiones para declarar arrays de otros tipos de datos del mismo modo, pero no son tan importantes así que las veremos en secciones posteriores.



Sin embargo; hay una diferencia entre un array de char (o puntero cuando pasa a una función) y un array de otro tipo de dato. La función `printf`, por ejemplo, recibe un puntero a char, pero en ningún sitio indicamos el tamaño de éste. Alguna manera debe tener cualquier función que reciba un puntero a char que contenga texto de saber cuánto mide dicho array. Esa manera es que todas las cadenas de texto (*strings* en inglés) en C terminan con un char con el valor 0. Es decir, al escribir "`¡Hola, mundo!`" estamos creando rápidamente un array de char que contiene **catorce** char, las letras que ves y un char con valor 0 al final. Voy a demostrarte que esto es así con un pequeño programa:

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     char correct_string[] = "Me gusta el batido de chocolate.\n";
6     char incorrect_string[] = { 'M','e',' ','g','u','s','t','a',' ','
7                                'l','a',' ','p','i','z','z','a',' ','
8                                'c','o','n',' ','b','a','c','o','n','
9                                '.' };
10    printf(correct_string);
11    printf(incorrect_string);
12 }
```

Programa 54: Arrays de char

La primera cadena se imprimirá correctamente, pero la segunda se imprimirá y, con toda probabilidad, después de ella saldrán cosas, probablemente sin sentido. Esto es porque como la segunda cadena no acaba en un char con valor 0, `printf` no sabe cuándo dejar de imprimir. Aparte de esta manera de trabajar con ellos, los arrays a char funcionan como cualquier otro array y, cuando se convierten en punteros, como cualquier puntero. En secciones posteriores te explicaré cómo manipular cadenas de texto de maneras más avanzadas.

8.3.3. El puntero nulo

Hay un valor especial para todos los tipos de puntero, que es el puntero nulo, o, como se escribe en el lenguaje: `NULL`. Es un puntero a `void` que vale 0. Si recuerdas el mapa de memoria, esa dirección correspondería al código, nuestro programa no puede modificarse a sí mismo ni leerse, entre otras cosas porque parte de esa sección no es nuestro código, sino código del sistema operativo que incrusta en nuestro binario para permitirnos realizar ciertas funciones. Así que los diseñadores del lenguaje utilizaron este valor para simbolizar un puntero que está en un estado especial.

Uno de los usos más importante de este puntero es que sirve para explicar si una operación ha ido bien o no. Por ejemplo, cuando abrimos un archivo con una función llamada `fopen`, éste devuelve un puntero a una estructura, si el archivo no existe, o no tienes permisos para abrirlo, ese puntero será `NULL`. Muchas funciones que reciben un puntero usan `NULL` para indicar un comportamiento especial. Por ejemplo, vamos a escribir una función que encapsule la funcionalidad del programa 37, esto ha sido un ejercicio, así que si no lo has hecho, hazlo ahora; pero le vamos a dar un giro: en vez de recibir en la función las estructuras, vamos a recibir punteros. Primero, porque como ya te dije, ahorramos la copia de la estructura y además podemos usar `NULL` para indicar valores especiales.



```
1 struct point_s {
2     double x; double y;
3 };
4
5 double distance(struct point_s *a, struct point_s *b) {
6     double res = 0.0;
7     struct point_s origin = { .x = 0.0, .y = 0.0 };
8     if(NULL == a){
9         a = &origin;
10    }
11    if(NULL == b){
12        b = &origin;
13    }
14    double diff_x = a->x - b->x;
15    double diff_y = a->y - b->y;
16    res = sqrt(diff_x * diff_x + diff_y * diff_y);
17    return res;
18 }
19
20 int main(void)
21 {
22     struct point_s a = {.x = 3, .y = 4};
23     double d = distance(&a, NULL);
24     printf("Distance: %f\n", d);
25 }
```

Programa 55: Uso de punteros a NULL

Si observas la función que hemos escrito, dentro de ella declaramos un punto que simboliza el **origen de coordenadas**. Lo que hacemos es que si uno de los punteros a estructuras es NULL, entendemos que ese punto es el origen de coordenadas. Lo que hacemos es que asignamos a nuestros argumentos (que son punteros) la dirección a esta variable local que hemos declarado. Recuerda: los argumentos que una función recibe son **copias** de los valores. En este caso, nuestro argumento es un puntero a una estructura. Al asignar a nuestro argumento otro valor **no estamos alterando la estructura original**, porque no hemos cambiado el valor al que nuestro argumento apunta, sino el mismo argumento. Una vez hecho esto, podemos calcular la distancia del mismo modo que haríamos si no fueran punteros. ¿Qué utilidad tiene escribir esta función así? Encontrar la distancia de un punto al origen de coordenadas es una operación común, al hacer esto así, permitimos que quien llame a la función lo haga sin declarar una estructura extra que simbolice el origen.

8.4. Alojamiento y liberación de memoria

Ahora que ya hemos aprendido lo que es un puntero, y a manejarlos, podemos pedirle al sistema operativo memoria de esa que se almacena en el montículo y podemos gestionar de manera más flexible que los arrays. Para eso utilizamos dos funciones: `malloc` y `free`. Los nombres son bastante descriptivos: la primera significa *memory alloc* y la segunda liberar. Cuando necesitas reservar memoria en el montículo, llamas a `malloc` y le pides una región contigua de memoria de n bytes. La función `malloc` devuelve un puntero a `void`. Ese mismo puntero deberá, en algún momento, liberarse pasándoselo a `free` más tarde.



Hablando de punteros a `void`, te dije que `void` significaba que las funciones o no recibían nada o no devolvían nada, según donde lo pusieras. El significado del puntero a `void` está relacionado: es un puntero a algo que no sabes qué es, `malloc` no tiene modo de saber qué vas a hacer con esa memoria, así que es el tipo de puntero que devuelve. Un puntero a `void` también es útil cuando queremos hacer funciones o estructuras que valgan para distintos tipos de datos. Veamos primero un ejemplo sencillo de cómo usar `malloc` y `free`.

Una de las utilidades principales de la reserva dinámica de memoria es cuando otra parte del programa hace un cálculo que devuelve un resultado cuyo tamaño no sabemos. Por ejemplo, imagínate una función que, dado un array de números, devuelve un vector con un ejemplar de cada número distinto, es decir, elimina las repeticiones del array, llamémosla `erase_reps`. Las funciones no pueden devolver arrays, eso ya lo sabemos, pero quien llame a `erase_reps` podría declarar un array y pasárselo a la función, pero en estos casos existe un problema: no sabemos el tamaño que tendría ese array. Es cierto que en este caso tenemos un **umbral superior**, si le damos a esta función un array de n posiciones, sólo puede devolverse, como máximo, un vector de n posiciones. Así que podrías declarar un array de n posiciones y pasárselo a la función para que ésta escribiera allí el resultado. No obstante, hay un problema, ¿cómo sabemos qué parte de ese array es solución y cuánto es sobrante? Lo único que puedes hacer es devolver desde `erase_reps` el tamaño de la solución. En este caso tendrías un array de n posiciones de las cuales sólo usarías una indeterminada cantidad menor. Estás desperdiciando memoria. Y si bien no parece un problema, puede convertirse en uno.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int *erase_reps(int* array, int array_length, int* final_length) {
5     *final_length = 0;
6     int preliminary_array[array_length];
7     for (int ii = 0; ii < array_length; ++ii) {
8         int unique = 1;
9         for (int jj = 0; jj < ii; ++jj) {
10             if (array[ii] == array[jj]) {
11                 unique = 0;
12                 break;
13             }
14         }
15         if (unique) {
16             preliminary_array[*final_length] = array[ii];
17             ++(*final_length);
18         }
19     }
20
21     int* result = malloc(*final_length * sizeof(int));
22
23     for (int ii = 0; ii < *final_length; ++ii) {
24         result[ii] = preliminary_array[ii];
25     }
26     return result;
27 }
28
29
30 int main(void)
31 {
32     int array[] = { 20,1,2,3,4,5,8,7,8,9,6,6,5,4,1,2,3,8,5,4,4,5,6};
33     int length;
34     int* result = erase_reps(array, 23, &length);
35     for (int ii = 0; ii < length; ++ii) {
36         printf("%d\n", result[ii]);
37     }
38     free(result);
39 }
```

Programa 56: Ejemplo de reserva dinámica



En la línea dos del código vemos como aparece la línea `#include <stdlib.h>`, es una línea necesaria para usar `malloc` y `free`. Después llega la declaración de la función, hemos hecho que devuelva el puntero al resultado y reciba tres argumentos: el array del que vamos a eliminar repeticiones, la longitud del array y un puntero a entero que servirá para **indicar la longitud del vector resultado**. Este patrón de diseño es muy común en programas en C, cuando necesitas que la función calcule muchas cosas, recibes punteros a sitios donde dejar la información, esto es lo que hemos hecho con el tamaño del resultado. En el cuerpo de la función asignamos el valor 0 a `final_length` para empezar. Después declaramos un array preliminar en que guardar los números únicos, ¿por qué un array? Porque este no es el resultado, sino un array que usaremos para guardar los números hasta que sepamos cuántos y cuáles son. No sabremos cuántos números únicos hay hasta que los contemos, así que le asignamos el tamaño del array de entrada para poder usar ese umbral superior que mencioné antes, si el array de entrada mide n , nuestro resultado no puede tener más que n posiciones. Ese patrón también es muy común: utilizar una estructura de datos auxiliar que luego copiaremos a una mejor dimensionada y definitiva.

El siguiente bucle simplemente comprueba, para cada número del array, si ese número aparece antes. Presta atención al bucle interno, para cada elemento i del array, examinamos los elementos desde el primero (el 0) al anterior al i -ésimo y comprobamos si alguno es igual. Si lo es, utilizamos la variable `unique` para indicar si el número ha sido encontrado antes y, por lo tanto, no es único, así que asignamos a esta variable lógica el valor 0. Después, una vez hemos comprobado todos los elementos anteriores al actual, aumentamos la longitud final y copiamos este número al array preliminar. Cuando ya tenemos calculada la longitud del array final y todos los números en el array preliminar, podemos usar `malloc` para crear la solución final.

La función `malloc` devuelve, como ya hemos dicho, un puntero que indica el inicio de la zona de memoria que nos ha reservado. Para ello, recibe **el tamaño en bytes** que queremos. Y aquí verás el operador `sizeof`. Sí, he dicho operador, no función. De hecho, verás que ninguna función sale en azul en los ejemplos, y `sizeof` sí. Esto es por es un operador **unario** que nos da el tamaño en bytes de un tipo de dato que escribamos después, entre paréntesis. También sirve para obtener el tamaño de expresiones en general, pero eso lo veremos más adelante. De momento quédate con que `sizeof(int)` es igual al tamaño en bytes de un entero. Como puedes ver, simplemente multiplico el tamaño del entero por el de posiciones que sé que son únicas. El siguiente bucle, simplemente, copia de la solución parcial a la definitiva. Finalmente, devolvemos el puntero reservado con `malloc`.

En la función `main` simplemente declaramos un array con varios números aleatorios con repeticiones, llamamos a la función sobre ellos y mostramos el resultado por pantalla. Verás que funciona como se espera. Nota el uso que hacemos del operador `&` para pasarle a la función `erase_reps` el puntero a una variable normal.

Como apunte, he estado usando y mezclando las palabras array y vector en esta sección, y no lo he hecho casualmente: un vector es un conjunto de memoria contigua que aloja datos, pero que es susceptible de crecer y encoger, y que, en consecuencia, ha sido reservado dinámicamente. Un array es ese tipo de expresión que te expliqué en su propia sección. Se parecen, y de hecho comparten muchas características, pero no son exactamente lo mismo.

Y ya que hablamos de vectores y arrays, hay una diferencia fundamental entre los vectores o punteros y los arrays. De un array **podemos conocer su tamaño** y de los punteros que apuntan a vectores no. Y si de los arrays podemos conocer su tamaño, ¿por qué hemos estado usando siempre un valor literal? Porque no quería contarte eso hasta que pudiera comparar punteros y arrays. Antes te dije que el operador `sizeof` nos da el tamaño en bytes de un tipo de dato o **de una expresión**. El tamaño en bytes de un array es lo que esperaríamos, si contiene 10 enteros de 4 bytes, 40. Pero el tamaño de un puntero **es siempre el mismo**. Es más, el tamaño de un puntero a cualquier tipo de dato es siempre el mismo. Veamos cómo utilizar `sizeof` para calcular el tamaño de un array.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int array[] = { 12,42,53,85,45,54,11,26,21,13 };
7     int array_size = sizeof array / sizeof array[0];
8
9     for(int ii = 0; ii < array_size; ++ii){
10         printf("%d\n", array[ii]);
11     }
12
13 }
```

Programa 57: Diferencia entre sizeof con punteros y arrays

Aquí es donde entra en juego eso que te dije de que `sizeof` es un operador y no una función. Si ves la línea 7, utilizamos `sizeof` para hallar el tamaño del array y lo dividimos por el tamaño del primer elemento. Se puede ver que `sizeof` no es una función porque, cuando calculamos el tamaño de una expresión, lo podemos utilizar sin paréntesis. Debes usar paréntesis, sin embargo, cuando calcules el tamaño de un tipo de dato por nombre. Cabe aclarar que `sizeof` no necesita saber qué hay **dentro** de la primera posición, sólo el tipo de la expresión, en este caso `array[0]`, que sería un `int` y por tanto cuatro bytes. Podrías utilizar `sizeof` de manera similar en memoria que no fuera accesible y no pasaría nada porque `sizeof` no lee la memoria sino que evalúa el tipo de la expresión sobre el que aplicar el operador. De este modo, podemos saber fácilmente el número de posiciones. Aquí lo he asignado a una variable para que lo veas mejor, pero podríamos haber puesto esta expresión directamente en el bucle `for`. Bien, y sabiendo esto, quizás te estés preguntando por qué siempre que le pasamos un array a una función pasamos también un argumento que indica el tamaño del array, si podríamos usar `sizeof` para ello. Pues es que no podemos, porque cuando a una función se le pasa un array, recuerda que éste se convierte en un puntero. Lo bueno de los vectores, sin embargo, es que como los reservamos basándonos en su tamaño, podemos asignar ese tamaño a una variable antes de llamar a `malloc` y así tenerlo accesible.

Y aún tengo otro truco que enseñarte sobre este operador, y es que nos permite escribir las llamadas a `malloc` en un modo que favorezca algunos cambios en el código. Imagínate esta llamada a `malloc`: `int *vector = malloc(length * sizeof(int));` Es como la que hemos hecho antes, pero presenta un pequeño problema: si cambiamos el tipo del array, debemos estar muy atentos de cambiar también el tipo que hay dentro de `malloc`, porque si no estaríamos reservando menos espacio del que pensamos. No obstante; recordemos que `sizeof` permite calcular el tamaño de expresiones, así que podemos cambiar la llamada a una como esta: `int *vector = malloc(length * sizeof *vector);`. Fíjate, si `vector` es un puntero a `int`, `*vector` sería el primer elemento, un entero, y `sizeof` nos dará en consecuencia el tamaño de un entero (4). La ventaja de este estilo de llamada es que así, si cambiamos de opinión y queremos que `vector` sea un `double*`, no tenemos que acordarnos de cambiar nada dentro del argumento de `malloc`.

8.5. Composición de punteros

Ahora que ya conoces los mecanismos básicos de los punteros, podemos ver algunos ejemplos de estructuras más complejas con ellos. Por ejemplo, uno de los casos más interesantes con los que te encontrarás con relativa frecuencia es el puntero a puntero de determinado tipo de dato. Esto es el equivalente a un array de dos dimensiones, pero con reserva dinámica de memoria. Veamos un programa en C que cree esta estructura, te invito a que lo compares con el programa 34: Ejemplo de uso de array bidimensional, en el que veíamos la creación y uso de un array bidimensional.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int rows = 10;
7     int columns = 5;
8     int** matrix = malloc(rows * sizeof(*matrix));
9
10    for (int ii = 0; ii < rows; ++ii) {
11        matrix[ii] = malloc(columns * sizeof(*matrix[ii]));
12        for (int jj = 0; jj < columns; ++jj) {
13            matrix[ii][jj] = ii * columns + jj + 1;
14        }
15    }
16
17    for (int ii = 0; ii < rows; ++ii) {
18        for (int jj = 0; jj < columns; ++jj) {
19            printf("%2d\t", matrix[ii][jj]);
20        }
21        printf("\n");
22    }
23
24    for (int ii = 0; ii < rows; ++ii) {
25        free(matrix[ii]);
26    }
27    free(matrix);
28 }
```

Programa 58: Reserva, uso y liberación de un vector de vectores

En la línea 8 del programa de ejemplo podrás ver que declaramos un puntero con dos asteriscos, esto es porque es un puntero a puntero a entero. Los punteros pueden encadenarse infinitamente, y, en el fondo, no tendría por qué no ser así, si lo piensas detenidamente. Si un puntero es una dirección de memoria, nada me impide hacer que ésta apunte a un lugar de la memoria donde hay otra dirección, y así sucesivamente. Además, aquí puedes ver que reservamos memoria para `rows` punteros a entero. La regla para entender los punteros es que el número de asteriscos en la declaración se anula con los asteriscos usados para desreferencias, de este modo, si hemos declarado `matrix` como un `int**`, al hacer `*matrix` estamos obteniendo un `int*`. Observa la simetría, o, simplemente, cuenta los asteriscos, esta simple regla es sencilla, pero potente a la hora de aclarar nuestras ideas sobre punteros.

En la línea 10 empezamos un `for` que reserva memoria para cada fila de `matrix`, después, una vez reservada, rellenamos esos elementos con un valor con el bucle de la línea 12. Aquí simplemente estamos haciendo que cada casilla valga su posición desde el principio (empezando desde uno, que queda más bonito). Como puedes ver, estamos usando corchetes para indizar este doble puntero, con los corchetes pasa lo mismo que con los asteriscos, así que si `matrix` es, de nuevo, un `int**`, al hacer `matrix[ii][jj]` estamos obteniendo un `int`. Los siguientes dos bucles anidados simplemente imprimen la matriz.

Finalmente, debemos liberar la matriz, observa también la simetría de este proceso con el proceso de reservar la memoria, si primero hicimos un `malloc` para `rows` punteros y después cada puntero lo reservamos con un `malloc` de `columns` enteros, aquí liberaremos los punteros en orden inverso, primero liberaremos cada fila de la matriz y después la matriz en sí misma.



Como esto puede ser confuso la primera vez que se ve, voy a dibujar el mapa de memoria de esta situación, para que tengas una imagen visual de qué está ocurriendo. Los punteros son un concepto muy abstracto, así que no te preocupes si no acabas de entenderlos a la primera.

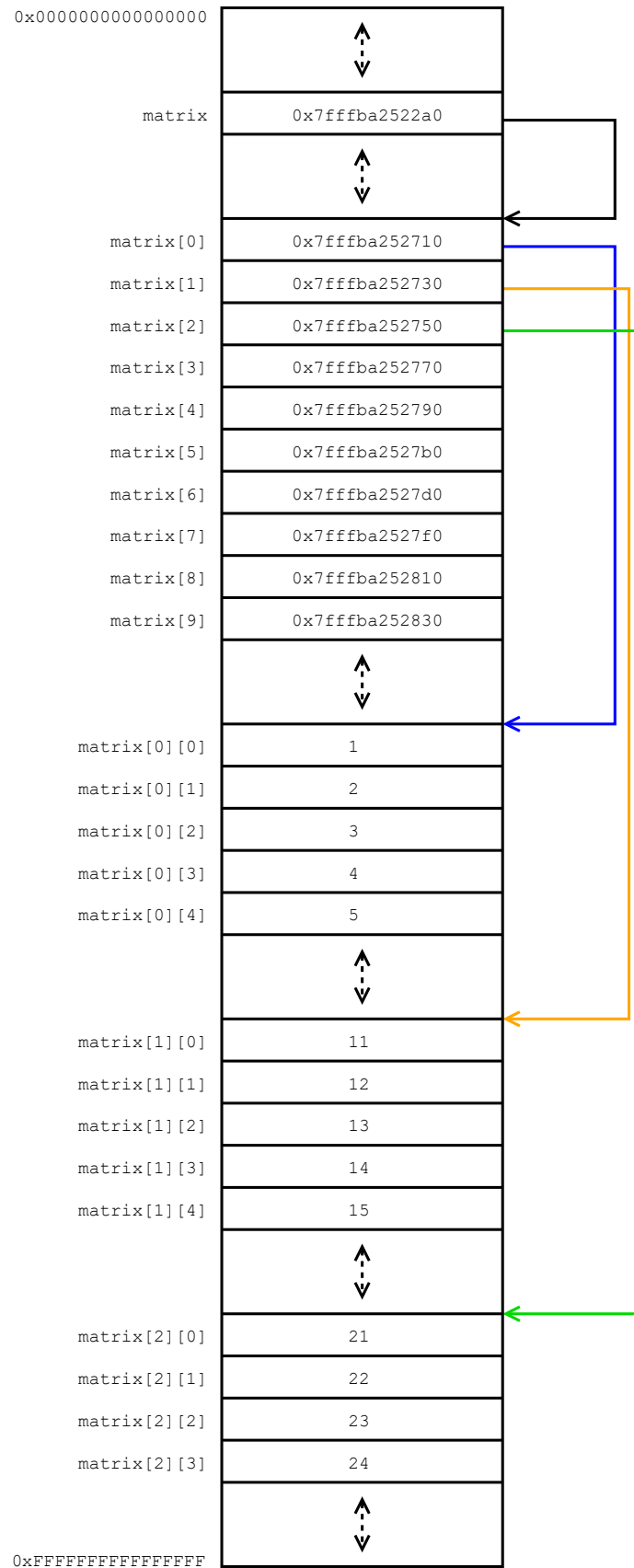


Figura 9: Mapa de memoria de un vector de vectores (doble puntero)



En la figura represento el mapa de memoria, a la izquierda están los nombres que tienen esas ubicaciones de memoria en el programa, y en el rectángulo expreso su contenido. Las flechas del lado derecho de la imagen representan las referencias de los punteros a dichas direcciones de memoria. Los colores simplemente sirven para que puedas seguirlas más fácilmente. Bien, si miras donde pongo la etiqueta `matrix`, verás que contiene una dirección de memoria, esta dirección referencia a un **vector de punteros**, es decir, rows punteros juntos. Cada uno de estos punteros apunta, a su vez, a una región contigua de memoria, en la cual se guarda una fila entera de la matriz. Sólo he representado las primeras tres filas, porque si no la figura sería excesivamente grande.

Ahora voy a poner la figura de cómo sería el mapa de memoria en el caso de un array doble, no incluyo el código porque su declaración sería simplemente `int matrix[10][5]`.

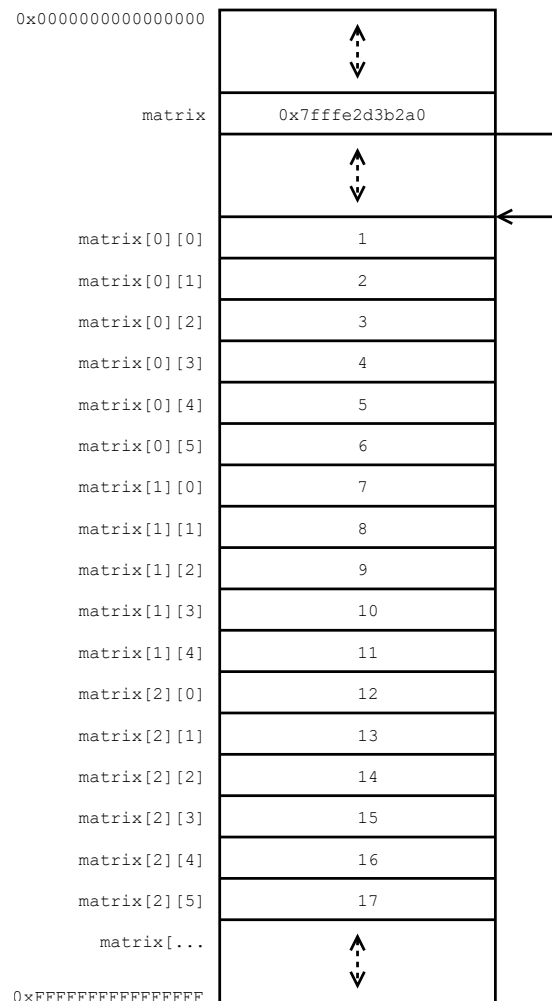


Figura 10: Mapa de memoria de un doble array



Como puedes ver, aunque el array es de dos dimensiones, no hay una segunda desreferenciación. Es decir, en ningún momento se sigue un segundo puntero. ¿Cómo es esto posible? Si miras el mapa, verás que el array se almacena en una región de memoria continua. Esto quiere decir que C sólo tiene que adquirir la dirección de inicio del array y, después, sumarle lo que indiquen los operadores corchete. Es aquí donde se presenta un problema cuando intentamos pasar un array bidimensional a una función. Llegados a la función, C no sabe si ese puntero que tiene es un vector de vectores o un array de arrays, por ello, si le pasaras este array doble a una función que recibe un puntero doble, al hacer, por ejemplo, `matrix[1][2]` lo que intentaría hacer es acceder como si fuera un puntero, y haría: `*(*(matrix + 1))+2`. Es decir, primero sumaría (recuerda la aritmética de punteros) uno a la dirección base, después, **interpretaría su contenido como un puntero** a otro vector, y a ese intentaría sumarle 2 para desreferenciarlo. El problema es que `*(matrix+1)` **no es** un puntero, es directamente el número.

C puede hacer esto porque, del mismo modo que te indiqué cuando te expliqué cómo funciona el operador `sizeof`, podemos ver que C sabe el tamaño de un array siempre que éste no decaiga a puntero, es decir, puedes saber el tamaño de un array en cualquier alcance interior a aquél en que fue declarado.

La conclusión lógica de lo que acabamos de ver es que el número máximo de dimensiones de array que puede recibir cualquier función es uno, porque es el que se comporta como un puntero sin problemas. El hecho de que el array, por ser declarado en una única orden, sea una estructura continua en memoria hace que pueda ser interpretado como una estructura de una sola dimensión. Por ejemplo, en el siguiente código declaramos y rellenamos un array de dos dimensiones y, sin embargo, podemos acceder a él como si sólo tuviera una dimensión.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int rows = 10;
7     int columns = 5;
8     int matrix[10][5];
9
10    printf("matrix = %p\n", matrix);
11
12    for (int ii = 0; ii < rows; ++ii) {
13        for (int jj = 0; jj < columns; ++jj) {
14            matrix[ii][jj] = ii * columns + jj + 1;
15        }
16    }
17
18    for (int ii = 0; ii < rows * columns; ++ii) {
19        printf("%d\n", (*matrix)[ii]);
20    }
21
22 }
```

Programa 59: Uso de un array como una estructura unidimensional

Como puedes ver, una vez hemos llegado a la región contigua de memoria, es decir `(*matrix)`, sólo tenemos que recorrerla como si fuera un array unidimensional.



8.6. Ejercicios de la sección

Ej. 10: Completa esta tabla de números en diferentes bases numéricas:

Decimal	Binario	Hexadecimal
73		
	00100110	
		0x12F
128		

Ej. 11: Vuelve al ejercicio noveno y reproduce los contenidos de la pila en cada bloque de código del programa. Utiliza de referencia la solución que propongo yo.

Ej. 12: Haz un programa que cree un puntero de tres niveles de tipo `int`, lo reserve correctamente, lo rellene con el valores correlativos **empezando en uno** y después lo imprima de una manera comprensible. Finalmente, libéralo también de tal modo que no quede memoria sin liberar al final del programa.

Ej. 13: Basándote en el programa anterior, crea dos funciones, una para crear una matriz tridimensional con memoria dinámica dadas sus tres dimensiones y otra para liberarla.



9. Modificadores de tipos: constancia y signo

Hasta ahora sólo hemos manejado los tipos básicos, pero a esos tipos se les pueden añadir **modificadores**, que son cualidades que crean un tipo ligeramente distinto basado en el anterior. El primero y más importante es el modificador `const`. Éste nos permite indicar que una variable es **de sólo lectura**, es decir, que una vez le demos valor cuando la declaramos, no podremos modificarla. Esto es muy útil para asegurarnos de que no introducimos errores en el código cuando manejamos datos que no deben ser cambiados.

Por ejemplo, imagínate un programa que utilice el número π . Por ejemplo para calcular el área de un círculo, necesitaríamos escribir `surface = PI * pow(r, 2)`. Pero, imagina que nos equivocamos y escribimos `surface = PI *= pow(r, 2)`, este error sería muy difícil de ver porque la primera vez la superficie estaría bien calculada, pero `PI` habría quedado sobreescrita por el uso del operador `*=`, y todos los cálculos posteriores serían incorrectos. Si declaramos `PI` como una constante, el compilador nos notificará de estos errores.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 const double PI = 3.141592;
5
6 double surface_circle(double radius) {
7     return PI * radius * radius;
8 }
9
10 double perimeter(double radius) {
11     return 2 * PI * radius;
12 }
13
14 int main(void)
15 {
16     double r = 3.5;
17     printf("Un círculo con un radio de %.3f tiene un perímetro de %.2f
18           y un área de %.2f\n", r, perimeter(r), surface_circle(r));
19 }
```

Programa 60: Uso de una constante numérica

En la línea cuatro del código puedes ver cómo declaramos `PI` como una constante de tipo `double`. Puedes ver que, además, estamos declarándola como variable global. Si recuerdas cuando te expliqué el alcance las variables y lo que era una variable global, te dije que era útil cuando había funciones. Aquí puedes ver uno de los usos comunes de las variables globales, cuando se definen constantes universales, como podrían ser π , e o la constante gravitatoria G . Pero volviendo al hecho de la constante, si intentas escribir una instrucción que modifique la constante, el compilador lanzará un error como este:

```
$ gcc -o main.exe main.c
main.c: In function 'main':
main.c:17:8: error: assignment of read-only variable 'PI'
   17 |     PI = 1.1;
      |     ^
```




Pero el modificador `const` se aplica en otro sitio muy importante, se utiliza en las declaraciones de los argumentos de las funciones para indicar que los argumentos no se pueden modificar. De nuevo, los argumentos de las funciones son copias de sus valores, así que, sabiendo esto, no tendría utilidad una manera de decir que no los podemos modificar, el asunto nuclear de esto es que el modificador `const` se aplica a punteros, indicando que no se puede modificar su contenido, y es ahí donde se vuelve tremendamente útil en conjunción con las funciones. Por ejemplo, volvamos a la función que utiliza punteros para calcular la distancia entre dos estructuras `point_s`, como no queremos que estas estructuras sean modificadas por la función, we can do this:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4
5 struct point_s {
6     double x; double y;
7 };
8
9 double distance(const struct point_s* a, const struct point_s* b) {
10     double res = 0.0;
11     struct point_s origin = { .x = 0.0 , .y = 0.0 };
12     if (NULL == a) {
13         a = &origin;
14     }
15     if (NULL == b) {
16         b = &origin;
17     }
18     double diff_x = a->x - b->x;
19     double diff_y = a->y - b->y;
20     res = sqrt(diff_x * diff_x + diff_y * diff_y);
21     return res;
22 }
23
24 int main(void)
25 {
26     struct point_s a = { .x = 3 , .y = 4 };
27     double d = distance(&a, NULL);
28     printf(" Distance : %f\n", d);
29 }
30 }
```

Programa 61: Uso de punteros constantes como argumentos de función

Si miras el programa, la única diferencia es que en la declaración antepone el modificador `const` delante del tipo del dato. Esto impide que modifiquemos el contenido de este puntero dentro de la función. Si intentas hacer, por ejemplo: `a->x++`; el compilador lanzará un error. Esto es de suma utilidad para el programador que use la función sin haberla escrito, porque así entiende si una función va a modificar los datos que éste le pase, sin necesidad de saber qué hace la función por dentro. Cuando veamos cómo dividir el programa en varios archivos, verás la importancia de esto en su totalidad.



Además, el modificador `const` añade el concepto de *const correctness*, este concepto es aquél por el cual hay que respetar la cualidad de constancia de las variables y argumentos. Esto quiere decir que debes definir tus funciones con cuidado de indicar todo lo que sea posible como constante. Por ejemplo, en el caso de la función que calcula la distancia entre dos puntos los dos argumentos deben ser constantes. Pero es más, una función **puede devolver una constante**. Esto sirve cuando se crean estructuras de datos que no quieres que el usuario sea capaz de modificar directamente, sino sólo mediante las funciones que tú quieras.

Hablaremos más de esto en un futuro, pero te pongo un ejemplo básico: imagínate una estructura que guarda datos de una persona. En esta estructura tendremos varios punteros a `char`: el nombre, un primer apellido y un segundo apellido. Lo que haremos es crear una función que cree esta estructura recibiendo los tres textos, reservará la memoria necesaria y después copiará los textos a esa memoria. Después, para manejar esa estructuras vamos a crear funciones que reemplacen estos arrays cuando el usuario quiera. Vamos a por ello:



```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  struct person_s {
6      char* name;
7      char* last_name_1;
8      char* last_name_2;
9  };
10
11 struct person_s person_create(const char* name,
12                               const char* last_name_1,
13                               const char* last_name_2) {
14     struct person_s res;
15
16     res.name = malloc(strlen(name) + 1);
17     res.last_name_1 = malloc(strlen(last_name_1) + 1);
18
19     for (int ii = 0; ii < strlen(name) + 1; ++ii) {
20         res.name[ii] = name[ii];
21     }
22
23     for (int ii = 0; ii < strlen(last_name_1) + 1; ++ii) {
24         res.last_name_1[ii] = last_name_1[ii];
25     }
26
27     if (NULL != last_name_2) {
28         res.last_name_2 = malloc(strlen(last_name_2) + 1);
29         for (int ii = 0; ii < strlen(last_name_2) + 1; ++ii) {
30             res.last_name_2[ii] = last_name_2[ii];
31         }
32     }
33     else {
34         res.last_name_2 = NULL;
35     }
36     return res;
37 }
38
39 void person_set_name(struct person_s* person, const char* name){
40     free(person->name);
41     person->name = malloc(strlen(name)+1);
42     for(int ii = 0; ii < strlen(name) + 1; ++ii){
43         person->name[ii] = name[ii];
44     }
45 }
46
47
48 void destroy_person(struct person_s *person){
49     free(person->name);
50     free(person->last_name_1);
51     free(person->last_name_2);
52 }
```

Programa 62: Estructura con punteros constantes – Funciones de manipulación



```
1 const char* person_get_name(const struct person_s* person) {
2     return person->name;
3 }
4
5 const char* person_get_last_name_1(const struct person_s* person) {
6     return person->last_name_1;
7 }
8
9 const char* person_get_last_name_2(const struct person_s* person) {
10     if (NULL == person->last_name_2) {
11         return "";
12     }
13     else {
14         return person->last_name_2;
15     }
16 }
```

Programa 63: Estructura con punteros constantes – Funciones de recuperación de información

```
1 int main(void)
2 {
3     struct person_s myself = person_create("Francisco", "Rodríguez", "
4         Melgar");
5     printf("Esta persona es: %s %s %s\n", person_get_name(&myself),
6         person_get_last_name_1(&myself),
7         person_get_last_name_2(&myself));
8
9     person_set_name(&myself, "José");
10
11     printf("Esta persona es: %s %s %s\n", person_get_name(&myself),
12         person_get_last_name_1(&myself),
13         person_get_last_name_2(&myself));
14
15     destroy_person(&myself);
16 }
```

Programa 64: Estructura con punteros constantes – función main

En este código puedes ver como «ocultamos» al usuario cómo manejamos estos punteros. Para impedir que él cambie su contenido sin utilizar nuestras funciones, las funciones que devuelven los punteros para poder usarlos, por ejemplo para imprimirlos, devuelven punteros constantes. Si intentaras hacer, por ejemplo: `person_get_name(&myself)[3] = 'a'` el compilador lanzaría un error. Esto es una herramienta para impedir que el usuario de la estructura se olvide de liberar la memoria a la hora de reemplazar el texto por otro.

Si estás atento, te habrás dado cuenta de que todo esto es algo inútil cuando el usuario de la estructura podría simplemente escribir `myself.name[0] = 'a'`, y tendrías razón. En secciones posteriores veremos maneras de evitar esto. Pero incluso con este problema, hacer esto es una buena manera de ahorrarle trabajo al usuario de la estructura.



El otro modificador que quiero presentarte es el signo, o mejor dicho, la ausencia de signo. La palabra reservada `unsigned` nos permite declarar variables de los mismos tipos que los tipos básicos, pero que sólo contengan valores positivos. Si no ves a primera vista la utilidad de esto, ésta reside en que al hacer el tipo sin signo, aumentas el rango que puede almacenar en los números positivos, si un `char` tiene un rango en $[-127, 128]$, al hacerlo sin signo, éste tiene un rango de $[0, 255]$. Además, esto permite añadirle significado a tus variables.

Por ejemplo, una variable que guarde el tamaño de algo, por ejemplo de un array o de un vector, no debería tener signo, porque nunca puede ser negativa. Una variable que guarde un mes, por ejemplo, tampoco. En el caso de tamaños de arrays o vectores es importante porque hacer las variables que guardan su tamaño no tengan signo, porque así nos permite hacer arrays el doble de grandes. No existen tipos sin signo para números de coma flotante (`float` y `double`). Veamos un ejemplo de cómo utilizar el modificador `unsigned` para declarar variables, argumentos y tipos de retorno.

```
1 #include <stdio.h>
2
3 unsigned int factorial(unsigned int n) {
4     unsigned int res = 1;
5     for (unsigned int ii = 0; ii < n; ++ii) {
6         res *= n - ii;
7     }
8     return res;
9 }
10
11 int main(void)
12 {
13     unsigned int number = 10;
14     printf("%d! = %d\n", number, factorial(number));
15 }
```

Programa 65: Uso de tipos sin signo

Sé que parece un poco engorroso escribir `unsigned` cada vez, más adelante veremos cómo solventar esto.

9.1. Ejercicios de la sección

El ejercicio más apropiado es que revises todos los ejercicios que hemos hecho hasta ahora y los reescribas teniendo en cuenta la constancia de las funciones y el signo.



10. Comunicar tu programa con el exterior

Por fin vamos a llegar al punto en que tu programa pueda comunicarse con el exterior, hasta ahora, todos los datos que hemos introducido en el programa están escritos en el código como literales. Eso es muy impráctico, en términos generales, un programa leerá, ya sea por consola o desde un archivo concreto, los datos que deba utilizar. Hay tres fuentes de información externa básicas que vamos a manejar (hay muchas más, que no explicaré en este manual):

1. Argumentos de la línea de comandos.
2. La entrada por consola.
3. Ficheros.

Lo primero es algo que aún no sabes qué es, pero que será bastante útil. Hasta ahora, la función `main` no recibía ningún argumento, pero puede recibirlos. Si `main` es una función que sólo sirve como punto de entrada a nuestro programa, ¿quién la puede llamar con argumentos? Básicamente estos argumentos vienen de la línea de comandos con la que llamemos a nuestro programa. Para poder acceder a ellos dentro del programa, debemos declarar la función `main` de este modo:

```
1 int main(int argc, const char** argv) {  
2 // ...
```

Programa 66: Declaración de una función `main` que reciba argumentos

De estos dos argumentos, el primero es el número de argumentos que ha recibido tu programa, y el segundo es un vector de vectores a `char` que nos llega en forma de puntero a puntero. Los argumentos que recibe un programa vienen en formato de cadenas de texto, por lo que, si son números, debemos utilizar funciones auxiliares para convertirlos a esos tipos de dato. Vamos a hacer un programa que reciba un número indeterminado de argumentos y los imprima, cada uno en una línea nueva.

```
1 #include <stdio.h>  
2  
3 int main(int argc, char const *argv[])  
4 {  
5     for (int ii = 0; ii < argc; ++ii) {  
6         printf("%s\n", argv[ii]);  
7     }  
8 }
```

Programa 67: Utilización de los argumentos de un programa

Y quizás te preguntes que cómo le paso esos argumentos al programa, bien, es sencillo: al llamar el programa, después de la ruta al ejecutable, pones todos los argumentos separados por espacios, por ejemplo:

```
$./main.exe argumento1 argumento2 argumento3
```

Si compilas este programa y lo ejecutas con esos argumentos, imprimirá esto:

```
./main.exe  
argumento1  
argumento2  
argumento3
```



Y sí, como ves, el primer argumento es el comando con el que se llamó el programa, esto es importante porque, como deducirás, quiere decir que tu programa siempre recibirá al menos uno. Este primer argumento cambia según la orden con la que llamemos a nuestro programa, por ejemplo, si en vez de con esa ruta relativa lo llamamos con una ruta absoluta, obtendríamos los siguiente:

```
$ /home/usuario/test/project/main.exe arg1 arg2 arg3
/home/usuario/test/project/main.exe
arg1
arg2
arg3
```

Como última nota, si el espacio es el carácter que se usa para separar argumentos, ¿pueden tener espacios los argumentos? Sí, simplemente rodea el argumento de comillas dobles. Y, entonces, ¿pueden tener comillas dobles? Sí, simplemente pon una barra inversa delante de la comilla, veamos un ejemplo rápido,

```
$ ./main.exe "Esto es una comilla: \" \"\" \"\" \"\" \"Frase con espacios"
./main.exe
Esto es una comilla: "
""
Frase con espacios
```

Si necesitamos leer un número, debemos utilizar, como ya hemos dicho, una función que nos convierta el argumento en un número. La función básica para hacer esto es `atoi` (del inglés: *ASCII to integer*). Por ejemplo, vamos a crear un programa que reciba una serie de argumentos y los sume:

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char const* argv[])
5 {
6
7     int total = 0;
8
9     for (int ii = 1; ii < argc; ++ii) {
10         total += atoi(argv[ii]);
11     }
12
13     printf("La suma de los argumentos es: %d\n", total);
14 }
```

Programa 68: Programa que suma argumentos

Nota que empezamos a sumar desde el segundo argumento (la posición uno del vector) porque el primero no es un número. Esto me lleva a advertirte de que `atoi` es una función muy básica y si introduces en ella una cadena que no es un número, devolverá valores sin sentido, en muchas ocasiones 0. Así que ten cuidado sobre esto. De todos modos, en otras secciones veremos maneras de manipular cadenas de texto más complejas que nos permitirán comprobarlo. Sería un ejercicio interesante que hicieras un programa que comprobara si una cadena de texto es un número o no. Finalmente, también existe `atof` que hace lo mismo, pero con decimales.



Otra opción que existe es dejar que, una vez iniciado el programa, el usuario escriba por consola lo que deba éste recibir. Por ejemplo, podríamos hacer un programa que sea una versión 2.0 de nuestro primer *Hola Mundo*. Esta versión podría primero preguntar el nombre del usuario y después saludarle por él. Para conseguir esto se utiliza (o se puede utilizar, hay otras alternativas) la función `scanf`. Esta función es la hermana gemela de `printf` porque se comporta igual, simplemente especificas un formato y le das las variables donde quieres guardar los datos y la función leerá de la terminal con ese formato. Por ejemplo, hagamos este *Hola Mundo 2.0*.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define NAME_LENGTH ((size_t)1024)
5
6 int main(int argc, char const* argv[])
7 {
8
9     char name[NAME_LENGTH] = "";
10    printf("Hola, ¿cuál es tu nombre?\n");
11    scanf("%s", name);
12    printf("Encantado de conocerte, %s.\n", name);
13 }
```

Programa 69: Ejemplo básico de `scanf`

Como puedes ver, es sencillo, pero debes tener en cuenta que si lees tipos de datos básicos con `scanf` debes pasarle punteros a esas variables. En el caso de un puntero a `char` es menos evidente. Además, salvo que llames a otras funciones para cambiar cómo se comporta la terminal, `scanf` sólo leerá hasta el primer carácter en blanco que se encuentre, es decir, hasta el primer espacio, por ejemplo. Eso quiere decir que si quieres leer varias palabras debes invocar a `scanf` varias veces o invocarlo con varios especificadores de formato. Vamos a ver un ejemplo un poco más complicado para que quede todo esto claro.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define NAME_LENGTH ((size_t)1024)
5
6 int main(int argc, char const* argv[])
7 {
8
9     char name[NAME_LENGTH] = "";
10    int age = 0;
11    double height = 0.0;
12    printf("Hola, quiero conocerte, dime tu nombre, tu edad y cuánto\n");
13    scanf("%s %d %lf", name, &age, &height);
14    printf("Encantado de conocerte, %s. Así que tienes %d años y mides\n");
15    printf("%f m\n", name, age, height);
16 }
```

Programa 70: Ejemplo avanzado de `scanf`



Como hemos dicho, `scanf` deja de leer en los espacios, así que puedes escribir cada dato y pulsar enter o escribir las tres respuestas separadas por espacios y darle a enter una sola vez. Además, nota que, como queremos leer en un `double`, tenemos que usar el especificador `%lf`, el compilador lanza un *warning* si lo dejamos con simplemente `%f`. Esta función **bloquea** el programa hasta que recibe **todos** los argumentos pedidos por los especificadores. Además, lo que se escriba en la terminal y no se consuma (se lea por `scanf`) queda pendiente para llamadas posteriores, es decir, si en vez de escribir en nuestro ejemplo una sola llamada, escribieras tres, una con cada especificador y variable, el efecto sería el mismo, podrías seguir escribiendo todo separado por espacios. Del mismo modo que `atoi`, si especificas que recibirá números y la entrada es distinta, fallará dando valores con poco o ningún sentido.

Y finalmente: los ficheros, o archivos. Los programas pueden eliminar, crear, escribir y leer de archivos. Para esto hay varias maneras, algunas más pedestres que otras, porque algunas son más estándar y otras dependen del sistema operativo. Al contrario que con los otros métodos, los ficheros tienen un ciclo de vida más complejo. El ciclo de vida es, en este contexto, la descripción de cuándo empieza a existir, existe y deja de existir algo. Por ejemplo, el ciclo de vida de una región de memoria dinámica es desde que la reservas hasta que la liberas, el de un array, desde que se entra en su bloque de código hasta que se sale, etc. Con los archivos pasa algo parecido.

Los archivos son un concepto que, de nuevo, gestiona el sistema operativo, de tal modo que tenemos que utilizar funciones concretas para abrirlos, escribir y leer en ellos y, finalmente, cerrarlos. A la mínima que estés un poco atento, verás el paralelismo entre esto y la reserva de memoria. La función para abrir un archivo es `fopen`, veamos su declaración:

```
1 FILE *fopen(const char *pathname, const char *mode);
```

Programa 71: Declaración de la función `fopen`

Devuelve un puntero a un tipo llamado `FILE`, este tipo es **opaco**, esta es una palabra que se utiliza en informática para decir que no puedes saber lo que hay dentro, es decir, es el sistema quien lo gestiona y tú sólo interactúas con este tipo mediante llamadas a función. Los argumentos de esta función son dos punteros a `char`. El primero es la ruta al archivo. Esta ruta puede ser relativa o absoluta, pero ten cuidado, si es relativa, ésta toma como origen **el directorio de trabajo** de la terminal donde lo ejecutes. Es decir, si le dices a la función que abra un archivo en la ruta `./file.txt`, lo buscará (o lo creará, según proceda), en tu directorio de trabajo, no en el que se encuentre el ejecutable, salvo que sean el mismo.

El segundo argumento es interesante, es, como indica su nombre, el modo de apertura del archivo. En éste argumento se suele escribir un literal que contiene una serie de letras, estas letras son atributos a la manera en que abrimos el archivo, veamos qué opciones tenemos.

1. `r`: Abre el archivo para lectura (sólo para lectura) la cabeza de lectura se sitúa al inicio del archivo. Si el archivo no existe, se produce un error y la función devuelve `NULL`.
2. `r+`: Igual que la anterior, pero permite escribir también.
3. `w`: **Vacía** el archivo o lo crea si no existe y permite escribir en él, situando la cabeza de lectura, lógicamente, al inicio.
4. `w+`: Igual que la anterior, pero permite leer también.
5. `a`: Abre el archivo para escribir, pero **no lo vacía**, sí que lo crea si no existe. La cabeza se sitúa al final del archivo, para **añadir** a al contenido que hubiera.
6. `a+`: El archivo es creado si no existe, se escribirá al final del mismo y permite también leer.



En la descripción de las opciones hablo de una cosa llamada cabeza de lectura. Es un concepto propio de los ficheros, en un fichero el programa guarda la posición del mismo en el que está esta cabeza. Cuando escribes o lees, ésta se mueve hacia delante tanto como el tamaño de los datos que hayas leído o escrito. Una analogía que se solía utilizar es que un fichero es como una cinta de vídeo, y el cabezal de lectura como el del reproductor. El cabezal empieza al inicio de una cinta, supongamos que ves 10 minutos, la pausas y la vuelves a reaudar, empezarás desde donde lo dejaste. Puedes rebobinar, avanzar el vídeo y, además, ver partes del vídeo hace que la cabeza avance. Es decir, leer avanza el cabezal, y si quieres ver algo que ya viste debes rebobinar. Todo sea dicho, el cabezal de lectura es único por proceso y archivo, así que dos programas pueden leer o escribir en el mismo archivo en puntos distintos. Por otro lado, cuando la cabeza de lectura llega al final, se para allí (lógicamente), pero si estás realizando una operación de escritura, el archivo se agrandará, como es lógico, porque de no ser así, no podríamos escribiríamos en archivos nuevos o vacíos.

Ahora que ya sabes cómo abrir un archivo, veamos cómo se lee o escribe de él. Las funciones que se usan para esto son dos: `fwrite` y `fread`.

```
1 size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Programa 72: Declaración de la función `fread`

```
1 size_t fwrite(const void *pt, size_t size, size_t nmemb, FILE *stream);
```

Programa 73: Declaración de la función `fwrite`

Las dos funciones se comportan más o menos igual, reciben un puntero, un tamaño, otro número y finalmente el puntero a la estructura que simboliza el fichero. El primer puntero es de entrada para escribir y de salida para leer. Es donde pondremos nuestros datos, o la memoria para que la función lo escriba cuando leemos. Estas funciones están hechas para escribir o leer objetos de un determinado tamaño, de ahí que existan los argumentos `size` y `nmemb`; el primero es el tamaño de cada uno de los objetos y el segundo el número de ellos que queremos leer o escribir. La función devuelve el número de **objetos** escritos o leídos, no de bytes. Más sobre esto en los ejemplos.

Finalmente, después de hacer lo que deseas con los archivos que has abierto, debes cerrarlo. Cerrar el archivo provoca que todos los cambios que se han escrito se sincronicen con el disco duro o sistema de almacenamiento subyacente. Esto es importante, si te olvidas de cerrar un archivo puedes ver que cambios que has escrito no se reflejan en el archivo. La función para cerrar archivos se llama `fclose`, su *signature* es la siguiente.

```
1 int fclose(FILE *stream);
```

Programa 74: Declaración de la función `fclose`

Como puedes ver, sólo recibe el archivo que deseamos cerrar o liberar.

Ahora que ya hemos explicado las funciones que hay que utilizar, vamos a ver un ejemplo simple de programa que utilice ficheros. Un programa muy típico podría ser uno que copie un fichero a otro, como el comando `cp`, que es el que se usa en Linux para hacer eso. Vamos a hacer un programa que reciba dos argumentos, el primero será el archivo que deseamos copiar y el segundo dónde. El comando en Linux admite que la segunda localización sea una carpeta, es decir, podrías hacer `cp ~/novel.txt /home/joe/novels` pero para hacer más sencillo nuestro ejemplo, obligaremos a que el usuario siempre especifique dos rutas completas. Veamos cómo se haría.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char const* argv[])
5 {
6
7     FILE *origin_file = NULL;
8     FILE *destiny_file = NULL;
9     char byte          = 0;
10
11     if (argc < 3) {
12         printf("Uso: main.exe <origen> <destino>\n");
13         return EXIT_FAILURE;
14     }
15
16     origin_file = fopen(argv[1], "r");
17
18     if (NULL == origin_file) {
19         printf("ERROR: El archivo de origen no existe.\n");
20         return EXIT_FAILURE;
21     }
22
23     destiny_file = fopen(argv[2], "w");
24
25     if (NULL == destiny_file) {
26         printf("ERROR: El archivo de destino no existe.\n");
27         fclose(origin_file);
28         return EXIT_FAILURE;
29     }
30
31     while (0 != fread(&byte, sizeof(char), 1, origin_file)) {
32         fwrite(&byte, sizeof(char), 1, destiny_file);
33     }
34
35     fclose(origin_file);
36     fclose(destiny_file);
37     return EXIT_SUCCESS;
38 }
```

Programa 75: Ejemplo de manejos de ficheros



Como puedes ver, hemos declarado dos variables de tipo puntero a `FILE`, las inicializamos a `NULL` y empezamos el programa. Cuando recibes argumentos es recomendable que los interpretes primero, porque si éstos son incorrectos no tiene sentido seguir procesando. Después, intentamos abrir el archivo de origen, nota que lo abrimos para solo lectura y sin eliminarlo, lógicamente. Si esa llamada falla, el valor devuelto será nulo, comprobamos que esto no es así antes de continuar. Hacemos lo mismo, después, con el archivo de destino, pero este archivo debemos abrirlo con permisos de escritura, creándolo si no existe y vaciando su contenido si ya existía, es decir, con la opción `r`. Date cuenta de que, en caso de error, además de salir del programa (terminar la función `main` con la palabra `return` termina el programa, lógicamente), debemos cerrar el archivo anterior que ya abrimos. Finalmente, utilizamos un bucle para leer byte a byte el archivo de origen y escribirlo en el de destino. Como puedes ver, la condición del bucle es que continuará hasta que el `fread` devuelva cero. Esto es así porque tanto `fread` como `fwrite` devuelven el número de estructuras del tamaño indicado por el segundo argumentos leídas o escritas, cuando se devuelva cero, es que se ha terminado de leer el archivo.

Voy a concentrarme en el valor de retorno, como ya vimos hace poco, a `fread` o `fwrite` le indicamos un tamaño y un multiplicador, los argumentos `size` y `nmemb` respectivamente. Esto nos permitiría indicar, por ejemplo, que se escribieran `nmemb` estructuras de su tamaño hallado con `sizeof`. El tamaño devuelto por las funciones es el número de estructuras escritas, no el de bytes. Si quieres escribir, como en este caso, un número de bytes, simplemente indica que `size` es uno y `nmemb` es el tamaño en bytes de lo que quieres escribir.

Ahora bien, quizás pienses que escribir un programa como este que lee byte a byte es un poco ineficiente, y tienes razón. Cada llamada a las funciones que tratan con cosas que gestiona el sistema operativo es relativamente costosa, por lo que es inteligente minimizarlas. En este caso, podríamos leer todo el primer archivo en memoria, cerrarlo y escribir en el segundo, en una sola orden, todo lo que ya leímos. El problema es que si por ejemplo intentarás copiar un archivo de 12 GB es probable que llenaras toda la memoria de un ordenador normal. Cuando eso pasa, o bien el sistema operativo termina tu proceso, o simplemente el ordenador se bloquea. Para evitar ambos extremos, lo que se suele hacer es utilizar un tamaño moderado lógico que guardar en memoria cada vez. Por ejemplo, digamos que queremos copiar los archivos mediante bloques de 100 MB, (o mejor dicho, 100 MiB). Veamos cómo quedaría el programa.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char const* argv[])
5 {
6
7     const size_t BLOCK_SIZE = (size_t)(1024 * 1024);
8     FILE *origin_file = NULL;
9     FILE *destiny_file = NULL;
10    char *buffer = malloc(BLOCK_SIZE);
11    size_t bytes_read = 0;
12
13    if (argc < 3) {
14        printf("Uso: main.exe <origen> <destino>\n");
15        return EXIT_FAILURE;
16    }
17
18    origin_file = fopen(argv[1], "r");
19
20    if (NULL == origin_file) {
21        printf("ERROR: El archivo de origen no existe.\n");
22        return EXIT_FAILURE;
23    }
24
25    destiny_file = fopen(argv[2], "w");
26
27    if (NULL == destiny_file) {
28        printf("ERROR: El archivo de destino no existe.\n");
29        fclose(origin_file);
30        return EXIT_FAILURE;
31    }
32
33    while (0 != (bytes_read = fread(buffer,
34                                    sizeof(char),
35                                    BLOCK_SIZE,
36                                    origin_file)))
37    {
38        fwrite(buffer, sizeof(char), bytes_read, destiny_file);
39    }
40
41    fclose(origin_file);
42    fclose(destiny_file);
43    free(buffer);
44    return EXIT_SUCCESS;
45 }
```

Programa 76: Ejemplo de lectura de fichero con *buffer*



El único cambio es que en vez de un único `char` declaramos un vector de ellos con un tamaño definido en la variable correspondiente y que, después, en el bucle de copia, en vez de copiar siempre un `byte`, intentamos leer el tamaño del bloque, y al escribirlo, utilizamos el valor de retorno de `fread`. Quiero detenerme aquí porque quizás esta línea te sorprende. Cualquier asignación es una expresión con un valor. Esto es lo que nos permite hacer algo como `a = b = c`. Gracias a esta propiedad, puedes comparar el valor de una asignación con otra variable, que es lo que estamos haciendo. En este caso, podríamos incluso prescindir de la operación de comparación, porque comprobar si un número es distinto de cero es lo mismo que pasa si pones ese número en el `if` directamente, sin embargo, me gusta hacerlo explícito.

Esto es un uso básico de lectura y escritura en archivos, pero otra cosa que se hace a veces es **mover la cabeza de lectura** sin leer o escribir. Por ejemplo, imagínate que queremos imprimir un archivo en orden inverso. Nos vemos en el mismo problema que el caso de uso anterior: lo más sencillo sería leer el archivo entero y después invertirlo, pero es un gran problema si el archivo es muy grande. Podríamos hacer lo mismo que hicimos antes, leer el archivo a trozos, invertir los trozos y después ponerlos juntos, también en orden inverso. Eso es complicado, gracias a que podemos mover la cabeza de lectura, podemos leer directamente los bloques en el orden inverso, gracias a esta operación.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define BLOCK_SIZE ((long) 100)
6
7 void invert_bytes(char* stream, int length)
8 {
9     for (int ii = 0; ii < length / 2; ++ii) {
10         char temp = stream[ii];
11         stream[ii] = stream[length - 1 - ii];
12         stream[length - 1 - ii] = temp;
13     }
14 }
15 int main(int argc, char** argv)
16 {
17     long current_pos = 0;
18     FILE* file = NULL;
19     if (argc != 2) {
20         printf("Uso del comando: main.exe <archivo>\n");
21         return EXIT_FAILURE;
22     }
23     file = fopen(argv[1], "r");
24
25     if (NULL == file) {
26         return EXIT_FAILURE;
27     }
28
29     fseek(file, 0, SEEK_END);
30     current_pos = ftell(file);
31     while (current_pos != 0) {
32         char block[BLOCK_SIZE + 1] = {};
33         long next_pos = 0;
34         long block_size = 0;
35         if (current_pos - BLOCK_SIZE < 0) {
36             next_pos = 0;
37             block_size = current_pos;
38         }
39         else {
40             next_pos = current_pos - BLOCK_SIZE;
41             block_size = BLOCK_SIZE;
42         }
43         fseek(file, next_pos, SEEK_SET);
44         fread(block, 1, block_size, file);
45         fseek(file, -block_size, SEEK_CUR);
46         invert_bytes(block, block_size);
47         printf("%s", block);
48         current_pos = ftell(file);
49     }
50     fclose(file);
51     printf("\n");
52 }
```

Programa 77: Ejemplo de uso de funciones para mover la cabeza de lectura



El programa es un poco complicado, pero, como siempre, iremos por partes. Lo primero que hacemos es definir una función que pueda invertir los bytes de un array a `char`. No es lo principal de este ejemplo, pero quédate con el algoritmo porque es una tarea usual y un algoritmo clásico. Después empieza lo que queremos hacer, es un poco complicado, pero me interesa más que veas lo que hacen las funciones de las que estamos hablando más que el proceso general. Hasta la línea 27 lo único que hacemos es algo con lo que ya estás familiarizado: procesamos los argumentos y abrimos el archivo, comprobando que todo ha ido bien. Después usamos la función `fseek` que nos permite **mover la cabeza de lectura y escritura**, concretamente al final. Veamos cómo funciona.

Esta función nos permite poner la cabeza de lectura en un punto añadiendo un desplazamiento, que puede ser positivo o negativo. Hay tres puntos de referencia que puedes usar con esta función:

1. `SEEK_SET`: El inicio del archivo, por ejemplo: `fseek(100, SEEK_SET)`; pondría la cabeza en el centésimo byte del archivo. Si usas este punto de referencia no puedes usar desplazamientos negativos, lógicamente.
2. `SEEK_CUR`: Es la posición actual, puedes usar desplazamientos positivos o negativos. Por ejemplo: `fseek(100, SEEK_CUR)`; podría usarse en un bucle para leer el byte 100, el 200, el 300...
3. `SEEK_END`: Es el final del archivo, por ejemplo: si quisieras ir al tercer byte desde el final podrías escribir: `fseek(-3, SEEK_END)`;

En la línea 29 lo que hacemos es irnos directamente al final del archivo. Después usamos otra función interesante llamada `ftell`. Esta función nos da la posición actual de la cabeza lectora en el archivo que recibe como argumento. Así sabremos la posición absoluta en que estamos. Después comprobamos si nos queda un tamaño del bloque que hemos elegido entero hasta el inicio del archivo. Si podemos, utilizamos ese tamaño del bloque, si no, utilizamos lo que podamos. Después, desplazamos a la siguiente posición que hemos calculado, como puedes ver, usamos un posicionamiento absoluto. Leemos y **volvemos a desplazarlo hacia atrás** tantos bytes como hemos leído. Después llamamos a la función que invierte los bytes e imprimimos la cadena de texto. Después, actualizamos la variable que nos dice en qué posición estamos. Cuando hayamos llegado a la primera posición del archivo, sabremos que hemos terminado. Debes tener cuidado, `fseek` sólo funciona cuando es posible moverse a donde le indicas. Cuando no es posible, no mueve la cabeza y da error (devuelve -1). Finalmente, cerramos el archivo e imprimimos un salto de línea para que el *prompt* salga en la siguiente línea.

No lo hemos usado en el ejemplo, pero podemos **eliminar** archivos. Para ello utilizamos la función `remove`, cuya *signature* es ésta:

```
1 int remove(const char *pathname);
```

Programa 78: Declaración de la función `remove`

Como puedes ver, simplemente recibe un nombre de archivo. Veamos un ejemplo básico de programa que haga uso de ella: uno que reciba por argumento una ruta y la elimine.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char const *argv[]) {
5     if (argc != 2) {
6         printf("Usage: ./main <path to the file>");
7     }
8
9     int error = remove(argv[1]);
10
11     if (error == 0) {
12         return EXIT_SUCCESS;
13     } else {
14         return EXIT_FAILURE;
15     }
16 }
```

Programa 79: Ejemplo de programa que usa la función `remove`

Como puedes ver, el programa es muy sencillo, llamo a `remove` con el primer argumento (después del nombre del programa) y compruebo si ha funcionado o no.

10.1. Ejercicios de la sección

Ej. 14: Escribe un programa que reciba un número variable de números como argumentos e imprima la descomposición en factores primos de todos ellos de este modo: si los argumentos fueran: 10, 8, 55 y 103:

```
Factors of
10: 2, 5
8: 2, 2, 2
55: 11, 5
103: 103
```

Se recomienda hacer control de errores comprobando que los argumentos son números antes de utilizarlos, etc.

Ej. 15: Escribe un programa que lea **por consola** una serie de palabras y que sólo deje de leer cuando se introduzca «!!» como palabra. Después, debe imprimir dichas palabras en orden aleatorio. La función `rand` devuelve un número aleatorio entre cero y el máximo entero positivo. Si quieres que devuelva números aleatorios **distintos** cada vez debes ejecutar `srand(time(NULL))`; al inicio de la función `main`. Debes añadir la línea `#include<time.h>` justo después de la línea `#include <stdio.h>`

Ej. 16: Haz una función que lea dos archivos e **intercambie** su contenido, escribe dicho programa de tal modo que no sea necesario alojar ninguno de los dos archivos en memoria completamente. Para hacer esto puedes usar este proceso:

1. Copia los contenidos del primer archivo en un archivo auxiliar en el directorio `/tmp/`
2. Cierra el archivo que has copiado.
3. Ábrelo de nuevo con un modo que elimine su contenido.
4. Copia los contenidos del segundo archivo en el primero.
5. Cierra el segundo archivo.
6. Ábrelo de nuevo en un modo que elimine su contenido.



7. Copia los contenidos del archivo auxiliar en el primero.

8. Elimina el archivo auxiliar en `/tmp/`.

Ej. 17: Escribe una función que reciba una palabra como argumento e indique en qué posición (en bytes) dentro del archivo se encuentra la palabra. Sólo tienes que dar la primera ocurrencia, si la palabra no se encuentra, devuelve un número negativo. Haz un programa que, con esa función, reciba una ruta a un archivo y una palabra e imprima el resultado de buscar la palabra en el archivo.



11. Cómo escribir programas legibles y claros

Todos los lenguajes de programación tienen un grado mayor o menor de **legibilidad**. Se entiende por esto lo sencillo que es para programadores que lean código en este lenguaje entender rápidamente qué está haciendo ese código fuente. No es el lenguaje que se usa lo único que afecta a la legibilidad, cómo se escriba en él afecta mucho a la legibilidad. Algunos consejos para aumentar la legibilidad son:

1. Utiliza nombres de variables significativos, es decir, en vez de `a`, llama a las variables cosas como `length`, `days` o algo que tenga que ver con su significado.
2. Utiliza nombres de función que expliquen lo que hace la función, del mismo modo, utiliza nombres de argumentos que indiquen qué son.
3. Escribir el código tabulando cada bloque de código nuevo, etc. Más sobre esto más adelante.

Además, en C hay dos herramientas de vital importancia para aumentar la legibilidad del código que ahora te voy a presentar. Una es la palabra reservada `typedef`. Esta palabra reservada nos permite **darle nombre a un tipo de dato**. Es decir, nos permite poner otros nombres a tipos que ya existieran. Uno de los usos más prácticos de él es que nos permite definir un tipo con una sola palabra para referirnos a una estructura, así nos ahorramos el hecho de escribir `struct mystruct_s`, no me dirás que no es un alivio después de tantos programas llenos de la palabra `struct`. Veamos un programa ejemplo de esto:

```
1 #include <stdio.h>
2
3 struct point_s {
4     double x;
5     double y;
6 }
7 };
8
9 typedef struct point_s point_t;
10
11 int main(void)
12 {
13     point_t p = {.x = 3.3, .y = 1.1};
14     struct point_s q = { 1, 3 };
15 }
```

Programa 80: Definición de un tipo a partir de una estructura

Esta sería la sintaxis más explícita para hacer esto, hay otra alternativa que veremos ahora. En las líneas 1 a 4 no hay nada nuevo, simplemente declaramos la estructura, es la línea 6 la que es clave, como puedes ver, usamos la palabra `typedef` para definir que el tipo `struct point_s` se va a llamar ahora también `point_t`. Nota que, como se ve en la línea 14, podemos seguir usando el nombre antiguo de los tipos. Todo sea dicho, salvo que quieras expresar un significado distinto, es lo mejor usar siempre el mismo nombre para los tipos. Hay una manera más abreviada para hacer esto, de hecho, dos, puedes combinar la sentencia `typedef` en una única línea con la creación del `struct`. Al hacer esto, puedes elegir ponerle nombre a la estructura o no, porque su nombre «real» será el que defines en el `typedef`. Veámoslo:



```
1 #include <stdio.h>
2
3 typedef struct point_s {
4     double x;
5     double y;
6 } point_t;
7
8
9 typedef struct {
10     point_t center;
11     double radius;
12 } circle_t;
13
14 int main(void)
15 {
16     struct point_s p = { 1.1, 2.3 };
17     point_t q = { 1.2, 3.4 };
18     circle_t origin = { .center = {.x = 0, .y = 0}, .radius = 1 };
19 }
```

Programa 81: Diferentes combinaciones de struct con typedef

Las dos declaraciones de estas dos estructuras son equivalentes en el hecho de que les asignan un nombre simple: `point_t` y `circle_t`, pero con un matiz: la estructura punto conserva su nombre de struct, por lo que se podría utilizar para declarar variables de su tipo como se ve luego, al struct círculo no le hemos puesto nombre de estructura, sólo lo hemos usado para definir un tipo. Esto es más simple, pero tiene un problema, cuando haces esto y el compilador necesita decirte que ha habido un error, será distinto dependiendo de la técnica que utilices. Con la estructura `circle_t`, si declaramos una función que recibe un argumento de este tipo y la llamamos con un tipo distinto, provocando un error, el mensaje del compilador será:

```
main.c:14:18: note: expected 'circle_t' {aka 'struct <anonymous>'} but
      argument is of type 'int'
14 | int foo(circle_t c){
```

Como puedes ver, te dice el nombre que tiene el tipo, y después intenta explicarte qué tipo era originariamente, el problema es que como hemos definido la estructura sin nombre, no tiene nada que decirnos, es un struct anónimo. Y te estarás preguntando que todo esto cómo puede ser, bueno, es porque en C puedes declarar estructuras sin nombre con una variable para usarlas y tirarlas. Te lo voy a enseñar, pero es algo que no he visto en código profesional, así que, como el goto, interioriza que existe, pero sería mejor que no lo uses.

```
1 #include <stdio.h>
2
3
4 int main(void)
5 {
6     struct { double x; double y; } temporal_point = { .x = 1, .y = 2 };
7     printf("Esto es un punto temporal que está en [%1.2f, %1.2f]\n",
8           temporal_point.x, temporal_point.y);
9 }
```

Programa 82: Estructura anónima y efímera



Es por eso que podemos definir struct que no tienen nombre y usar ese struct anónimo para definir un tipo. Mi consejo es que siempre le pongas nombre a los struct para evitar que el error del compilador sea más difícil de leer. Compara ese error del compilador que te he presentado antes con éste para la estructura punto que sí tenía un nombre independiente del typedef.

```
main.c:13:18: note: expected 'point_t' {aka 'struct point_s'} but
      argument is of type 'int'
13 | void foo(point_t p){
    |           ~~~~~^
```

Terminando ya con este tema: si te fijas, siempre que le he puesto nombre a una estructura, lo he terminado con `_s`, y siempre que he hecho un typedef, lo he terminado con `_t`. Esto es una convención, es decir, es algo que los programadores hacemos por tradición, pero no es obligatorio, ni el compilador ni ningún analizador de código te dirán que esto está mal (salvo que, claro, los hayas configurado para seguir la convención). El subfijo `_s` es menos importante, pero sí te recomiendo encarecidamente que termines todos los tipos que defines con typedef en `_t`, primero: porque es muy común, lo hace casi todo el mundo y, segundo: porque los editores entienden que cualquier identificador (nombre, vaya), que termina en `_t` es un tipo, y lo utilizan para darte pistas sobre qué es cada cosa, por ejemplo, los editores de texto colorearán todos los identificadores terminados en `_t` del color de los tipos en ese editor.

Aparte de esta utilidad tan cómoda para nosotros, hay otra, sirve para renombrar tipos básicos, por ejemplo, podemos renombrar el `char` sin signo como `byte_t`. Así si utilizamos nuestro programa para manejar listas de bytes podemos usar este tipo y el lector de nuestro código sabrá de qué estamos hablando.

```
1 #include <stdio.h>
2
3 typedef unsigned char byte_t;
4
5 void print_byte(byte_t b) {
6     byte_t current_byte = 128;
7     for (int ii = 0; ii < 8; ++ii) {
8         printf("%d", (b & current_byte) != 0);
9         current_byte /= 2;
10    }
11    printf("\n");
12 }
13
14 int main(void)
15 {
16     print_byte(110);
17 }
```

Programa 83: Redefinición de un tipo básico

Quizá no entiendas todo este código, la línea 8, concretamente, pero simplemente observa cómo al utilizar un tipo concreto, se lee todo mucho mejor. Además, nos ahorramos escribir `unsigned` múltiples veces. Del mismo modo que con el modificador de signo, podemos definir tipos con el modificador constante, por ejemplo: `typedef const char letter_t`.



```
1 #include <stdio.h>
2
3 typedef const char letter_t;
4
5 letter_t dictionary[] = { 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i',
6                           'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w',
7                           'x', 'y', 'z' };
8
9 int pos_of_letter(letter_t l){
10     for(unsigned int ii = 0; ii < sizeof(dictionary); ++ii){
11         if(l == dictionary[ii]){
12             return ii + 1;
13         }
14     }
15     return -1;
16 }
17
18 int main(void)
19 {
20     letter_t l = 'f';
21     printf("%c es la letra número %d del diccionario.\n", l,
22           pos_of_letter(l));
23 }
```

Programa 84: Ejemplo de definición de tipo a partir de un tipo constante

Aunque esto es posible, en general la constancia es mejor dejarla **explícita**, es decir, definir tipos para lo que necesites, pero escribir el modificador `const` cuando sea necesario, en vez de ocultarlo detrás del tipo redefinido.

«Y el más difícil todavía» decían los maestros de ceremonias de los antiguos circos cuando los trapezistas o los payasos realizaban el truco final. Con `typedef` este truco final es que, como los arrays de distintas dimensiones son tipos distintos, puedes definir un tipo a partir de un tipo array. ¿Qué utilidad tiene esto? A veces un conjunto de cosas de un tamaño concreto es un concepto en sí mismo. Ocho bits son un byte, siete días son una semana y una mano tiene cinco dedos. La sintaxis de esto es un poco distinta a la que podrías pensar, pongo el ejemplo y explico por qué rápidamente.

```
1 #include <stdio.h>
2
3 typedef unsigned char pixel_t[3];
4
5 int main(void)
6 {
7     pixel_t pixel = { 125, 33, 129 };
8     printf("Este pixel tiene los valores: Rojo = %hu, Verde = %hu, Azul
9           = %hu\n", pixel[0], pixel[1], pixel[2]);
10 }
```

Programa 85: Definición de un tipo personalizado a partir de un array

Como puedes ver, `[3]` se pone a la derecha del nombre del nuevo tipo, en vez de a la izquierda, que sería lo intuitivo. Simplemente recuérdalo si quieres hacer esto. De todos modos, al igual que con la constancia, que un tipo sea un array es algo que es recomendable dejar explícito.



La otra herramienta para añadir significado a tus programas, semántica, es el `enum`, o tipo enumerado. Este es un tipo en C que nos permite asignar números correlativos automáticamente a nombres, los ejemplo clásicos (tan clásicos que casi son manidos) son los siguientes: días de la semana, meses del año, colores del arcoíris... La utilidad de esto es que podemos codificar fácilmente conjuntos de nombres como estos a números y utilizarlos para iterar, para indexar arrays, etc. Veamos el ejemplo de los días de la semana.

```
1 #include <stdio.h>
2
3 enum week_days {
4     MONDAY,
5     TUESDAY,
6     WEDNESDAY,
7     THURSDAY,
8     FRIDAY,
9     SATURDAY,
10    SUNDAY
11 };
12
13 int main(void)
14 {
15     printf("Hoy es: %d\n", SATURDAY);
16 }
```

Programa 86: Ejemplo básico de tipo enumerado

Si compilas y ejecutas esto verás que imprime «Hoy es: 5». Esto es porque cuando escribes un enumerado así, el primer elemento recibe el valor cero, el siguiente uno, y así sucesivamente, en este caso de cero a seis. Y los enumerados pueden dar más de sí los combinas con arrays de tipos que ayuden a añadir la semántica, observa:



```
1 #include <stdio.h>
2
3 enum week_days {
4     MONDAY,
5     TUESDAY,
6     WEDNESDAY,
7     THURSDAY,
8     FRIDAY,
9     SATURDAY,
10    SUNDAY
11 };
12
13 const char* WEEK_DAYS_NAMES[] = { "lunes", "martes", "miércoles", "jueves",
14     "viernes", "sábado", "domingo" };
15
16 int is_today_weekend(int day){
17     return day == SUNDAY || day == SATURDAY;
18 }
19
20 int main(void)
21 {
22     for(int ii = MONDAY; ii <= SUNDAY; ++ii){
23         printf("Hoy es: %s y\n", WEEK_DAYS_NAMES[ii]);
24         if(is_today_weekend(ii)){
25             printf(";Es fin de semana!\n");
26         }else{
27             printf("No es fin de semana.\n");
28         }
29     }
30 }
```

Programa 87: Enum combinado con array de nombres

Tener el tipo enumerado nos permite que los programas sean mucho más legibles, que es lo que queríamos, porque como puedes ver, el bucle se lee como «siendo `ii` LUNES, hasta que sea igual a DOMINGO, si `ii` es un día de fin de semana, imprime que lo es, si no, no». Además, a la hora de imprimir los días, podemos saber el nombre de cada día simplemente accediendo al array de nombres. Habrás notado que tanto para `ii` como para la declaración de la función utilizo el tipo `int`. Esto es porque «por dentro» un tipo enumerado es un tipo entero, pero para escribir las cosas más claramente, podemos definir un tipo enumerado como definimos un tipo a partir de un `struct`.

Para ver que la legibilidad el código ha aumentado, mira cómo en la comprobación se puede ver que se lee como «si el día es SÁBADO o DOMINGO, es un fin de semana». Pero el `enum` tiene más flexibilidad, podemos definir el valor de cada etiqueta, o, esto es más útil: definir el valor de la primera, y las siguientes tomarán el valor correlativo correspondiente. Voy a poner en un mismo programa un ejemplo de ambas cosas, los días de la semana y los meses del año. El siguiente programa aúna todas las posibilidades anteriores:



```
1 #include <stdio.h>
2
3 typedef enum week_days_e {
4     MONDAY = 2, TUESDAY = 4, WEDNESDAY = 6, THURSDAY = 8, FRIDAY = 10,
5     SATURDAY = 12, SUNDAY = 14
6 } week_days_t;
7
8 typedef enum months_e {
9     JANUARY = 1, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST,
10    SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER
11 } months_t;
12
13 typedef enum seasons_e {
14     SPRING, SUMMER, FALL, WINTER
15 } seasons_t;
16
17 const char* season_names[] = { "primavera", "verano", "otoño", "
    invierno" };
18 const char* month_names[] = { "NOT USED", "enero", "febrero", "marzo", "
    abril", "mayo", "junio", "julio", "agosto", "septiembre", "octubre", "
    noviembre", "diciembre" };
19
20 int is_today_weekend(week_days_t day) {
21     return day == SUNDAY || day == SATURDAY;
22 }
23
24 seasons_t season(months_t m) {
25     if (m >= MARCH && m <= MAY) {
26         return SPRING;
27     }
28     else if (m >= JUNE && m <= AUGUST) {
29         return SUMMER;
30     }
31     else if (m >= SEPTEMBER && m <= NOVEMBER) {
32         return FALL;
33     }
34     else {
35         return WINTER;
36     }
37 }
38
39 int main(void)
40 {
41     for (months_t ii = JANUARY; ii <= DECEMBER; ++ii) {
42         printf("Estamos en %10s y es %s\n", month_names[ii],
43             season_names[season(ii)]);
44     }
45 }
```

Programa 88: Ejemplo final de enumerados



He tenido que declarar los `enum` en una línea para que quepa en esta página, pero eso no afecta. Mira bien como en el caso de los días de la semana he hecho que cada uno valga un valor arbitrario. Además, en iniciado los meses en el valor uno, no he definido valor del resto, por lo que se sucederán valiendo dos, tres... Esta definición tiene una implicación, en el caso de los días de la semana ya no puedo escribir el mismo bucle `for` para iterar sobre ellos **porque ahora sus valores no son correlativos**, sino que son arbitrarios. Además, si miras el array de los nombres de los meses del año, he tenido que poner al principio una posición que no se usa para que encaje.

Al contrario que `typedef`, los `enum` son un poco situacionales, aunque, como el `do-while` o el `switch`, cuando se da dicha situación son la herramienta ideal.

11.1. Estilo de código

Ahora que ya hemos hablado de muchas herramientas del language, es hora de que establezcamos algunas reglas para escribir nuestros programas que he dejado implícitas y que me gustaría empezar a escribir aquí. En este epígrafe vamos a ver cómo deben escribirse los programas más allá de que funcionen o, incluso, de que sean eficientes. Cuando uno trabaja en programación, hay muchos momentos en que vas a estar más tiempo leyendo el código de otros programadores que escribiendo código nuevo. Y, consecuentemente, debes escribir tu código con la intención de que sea claro para otros programadores que lo lean. Vamos a ver un ejemplo de cómo un fragmento de código puede ser trivial o indescifrable simplemente cambiando cómo está escrito.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main(void)
4 {
5     int** a=malloc(10*sizeof*a);
6     for(int ii=0;ii<10;++ii){
7         (*(a+ii))=malloc(5*sizeof**a);
8         for(int jj=0;jj<5;++jj){
9             (*(a+ii))[jj]=10*ii+jj;
10        }
11    }
12    for(int ii=0;ii<10;++ii){
13        for(int jj=0;jj<5;++jj){
14            printf(" %d ",(*(a+ii))[jj]);
15        }printf("\n");
16    }
17    for(int ii=0;ii<10;++ii){
18        free(*(a+ii));
19    }
20    free(a);
21 }
```

Programa 89: Ejemplo de programa escrito con un mal estilo

Intenta leer el programa 89 y dime si sabes qué hace. Es probable que, después de unos cinco o diez minutos, descifres que simplemente reserva un vector de vectores, lo rellena, lo imprime y lo libera. Como puedes ver, sin espacios, sin las líneas tabuladas (tabular las líneas es poner espacios delante para que queden desplazadas a la derecha), sin líneas en blanco entre estructuras de control y escribiendo algunas líneas al lado de una llave, es imposible de leer. Además, no hemos declarado variables o constantes que nos ayuden a entender si el mismo valor es así por casualidad o por simboliza lo mismo. Además, he mezclado arbitrariamente operadores distintos para acceder al vector. Si aplicamos una serie de mejoras a cómo está escrito el código, quedaría así:




```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  typedef unsigned int uint_t;
5
6  int main(void)
7  {
8      const uint_t ROWS      = 10;
9      const uint_t COLUMNS = 5;
10     int** matrix = malloc(ROWS * sizeof(*matrix));
11
12     for (uint_t ii = 0; ii < ROWS; ++ii) {
13         matrix[ii] = malloc(COLUMNS * sizeof(**matrix));
14         for (uint_t jj = 0; jj < COLUMNS; ++jj) {
15             matrix[ii][jj] = ROWS * ii + jj;
16         }
17     }
18
19     for (uint_t ii = 0; ii < ROWS; ++ii) {
20         for (uint_t jj = 0; jj < COLUMNS; ++jj) {
21             printf("%d ", matrix[ii][jj]);
22         }
23         printf("\n");
24     }
25
26     for (uint_t ii = 0; ii < ROWS; ++ii) {
27         free(matrix[ii]);
28     }
29     free(matrix);
30 }
```

Programa 90: Ejemplo de programa escrito con un buen estilo



Como puedes ver, hemos introducido varias mejoras: los operadores están rodeados por espacios, las utilidades de `sizeof` siempre llevan paréntesis para que sea más legible, hemos utilizado dos constantes para simbolizar el tamaño del vector en vez de escribir cinco y diez todo el rato en el código, hemos utilizado siempre la misma manera de acceder a los vectores y, finalmente, hemos escrito cada instrucción en una línea con la tabulación correcta. Veamos qué pasos o principio hay que seguir para escribir código que sea entendible.

11.1.1. Tabulación

En la introducción de esta sección hemos hablado de la tabulación, y te he dicho que es anteponer a algunas líneas espacios para que salgan desplazadas a la derecha. Es una de las cosas que debes hacer obligatoriamente si quieres que te tomen en serio como programador, pero hay varias variaciones sobre cómo puedes hacerlo. La primera decisión que debes tomar es si tabulas con espacios o con tabuladores. Un tabulador es un carácter especial que indica al editor de texto que alinee las cosas en la siguiente columna de la pantalla. La ventaja de tabular con ellos es que puede configurar el editor para que muestre el tabulador como el número de espacios que más te guste. Este carácter se inserta con la tecla tabulador, que es la que tiene este símbolo:  o tiene la palabra “tab” o similar. La otra opción es tabular con espacios, es decir, en vez de usar este carácter especial, usaremos un número determinado de espacios. En general, no pulsas varias veces la tecla espacio, sino que configuras tu editor de texto para que, al pulsar tabulador, se introduzca el número de espacios concreto.

Esta es una de las cosas que hará que los programadores se peleen entre ellos como fanáticos religiosos o hinchas de un equipo de balompié, por lo que la elección es tuya. Sin embargo; mi modesta opinión es que los espacios son mejores porque permiten alinear el código de líneas largas, por ejemplo, mejor. Además, permiten asegurarte de que el código escrito se verá bien en todos los entornos, porque no hay ambigüedad sobre cuánto mide un espacio.

La norma general para tabular es que los bloques de código deben tener sus instrucciones tabuladas un nivel más que allí donde estén definidos. Puedes observar que esto es así en todos los fragmentos de código que he incluido como ejemplos. Cada nivel de tabulación debe ser igual al anterior, es decir, si has elegido cuatro espacios, debes tabular siempre con cuatro espacios, nunca mezclar, y si utilizas tabuladores, un nivel de tabulación debe ser siempre un único tabulador.

La tabulación también entra en juego cuando una línea es demasiado larga. Te preguntarás quizá qué sentido tiene ponerle un límite a la longitud de las líneas si estamos usando editores de texto que nos permiten que éstas sean tan largas como queramos. Esto es así por varios motivos: las líneas demasiado largas suelen ser más difíciles de interpretar, impiden que tengas varios archivos abiertos a la vez y, además, en el improbable, pero posible caso de que quisieras imprimir tu código, sería difícil que quedase bien. El último caso no te afecta a ti, pero me ha afectado a mí en muchos de los programas que he escrito como ejemplos. La longitud de una línea de código en C normalmente viene siendo de unos ochenta caracteres. Para que te hagas una idea, los fragmentos que he insertado en este manual contienen unos 71.

Bueno, y qué pasa si la línea que estamos escribiendo contiene más de esos caracteres. Pues que hay maneras de escribir la línea de un modo distinto. Esta manera cambia según dónde se dé el problema, claro, así que vamos a ver los sitios más habituales y cómo arreglarlo.

El primer caso es cuando se utilizan funciones con muchos argumentos, o cuyos nombres son muy largos, un ejemplo clásico de esto son las llamadas a `printf`, pero se puede dar con otras funciones. Por ejemplo: imagina un programa que imprime un mensaje un poco largo para el usuario, podría quedar así:



```
1 #include <stdio.h>
2 int main(void)
3 {
4     printf("Este es un mensase importante, por favor, mantente
5         hidratado, bebe agua\n");
6 }
```

Programa 91: Impresión larga

Como puedes ver, la línea no cabe en la pantalla, lo que se hace en estos casos es separar la constante de *string* en varias. Siempre que las escribamos juntas o separadas sólo por espacios y líneas nuevas, C las tomará como una continua. Veamos cómo quedaría:

```
1 #include <stdio.h>
2 int main(void)
3 {
4     printf("Este es un mensase importante, "
5         "por favor, mantente hidratado,"
6         " bebe agua");
7 }
```

Programa 92: Impresión larga

Nota que, aunque las separemos, debemos incluir los espacios entre palabras. Cuando se dividen así cadenas de texto, se suelen poner todas al mismo nivel de tabulación.

El siguiente ejemplo es una función con muchos argumentos, o con argumentos muy largos. Imaginemos un programa tal que:

```
1 int main(int argc, char** argv)
2 {
3     int list[] = {};
4     int the_position = 10;
5     int* res = insert_at(list, ARRAY_SIZE(list), 0, rand());
6     print_array(res, ARRAY_SIZE(list) + 1);
7     free(res);
8 }
```

Programa 93: Muchos argumentos

Como no me interesa lo que hace el programa, sino sólo el formato, no voy a poner la definición de las funciones, pero la función `insert_at` crea un array nuevo con el elemento dado como último argumento en la posición que indica el segundo. Si quisiéramos que la línea donde se llama a esa función fuera separada en varias, lo haríamos generalmente entre los argumentos de la llamada, veamos algunos estilos:



```
1 //1
2 int* res = insert_at(list, ARRAY_SIZE(list),
3                       0, rand());
4
5 //2
6 int* res = insert_at(list,
7                       ARRAY_SIZE(list),
8                       0,
9                       rand());
10
11 //3
12 int* res =
13     insert_at(list, ARRAY_SIZE(list), 0, rand());
```

Programa 94: Cómo acortar líneas con muchos argumentos

En el primer ejemplo hacemos un único corte en un argumento y ponemos el resto de la línea un nivel de tabulador por detrás de donde estaría. Este método es útil cuando nos pasamos del límite por pocos caracteres. El siguiente es mi preferido y básicamente pone cada argumento en una línea y al mismo nivel. En el último, nos aprovechamos de que estamos asignando el resultado de la llamada, utilizamos esta operación para hacer el corte, con una nueva línea y un nivel de tabulador hacia adentro. Los tres se pueden combinar como quieras, pero a mí me gusta el segundo porque impide que queden alineaciones extrañas.

La siguiente localización es la declaración de una función con muchos argumentos, por ejemplo, una función que imprima una fecha dado el año, el mes y el día y un argumento booleano que nos diga si lo hacemos en ese orden y en el inverso:

```
1 void print_date(unsigned short day,
2                 unsigned char month,
3                 unsigned int year,
4                 int order);
```

Programa 95: Acortamiento de una declaración de función

En estos casos se suelen poner todos los argumentos en una línea distinta, además, me he molestado en alinear los nombres de los argumentos para que se cree una especie de tabla, lo que facilita la lectura cuando se deben acortar así las listas de argumentos.

En otros sitios donde se pueden dar estas situaciones es en listas de varios tipos, estoy usando la palabra listas aquí a la ligera, me refiero a las sucesiones de cosas separadas por comas que van entre llaves: listas de inicialización especialmente, en general puedes aplicar lo mismo que con los argumentos en el método primero de cuando acortamos una llamada a función.

11.1.2. Espaciado entre símbolos

En términos generales la mayoría de los espacios en blanco que pondremos en nuestros programas son para facilitar esta legibilidad, porque C es un lenguaje diseñado para que los espacios en blanco no importen. Normalmente, los operadores matemáticos, lógicos o de cualquier tipo deben ir rodeados de espacios, es decir: se prefiere `a = b + 10;` que `a=b+10;`, rodea el operador de asignación y la suma de espacios. Las excepciones a esto son los operadores unarios (la negación lógica, el asterisco de desreferenciación de punteros, etc.). Además, en las declaraciones, el asterisco debe estar siempre pegado o al tipo de dato, o al nombre de la variable. Es una cuestión de estilo, pero yo recomiendo que se adhieran los asteriscos al nombre de la variable, o, en su caso, del argumento de la función. Algunos ejemplos de estas decisiones serían:



1. Espacia los operadores: `int var = a * 3 + ii`
2. Los asteriscos pueden ir pegados a las variables a las que afectan, en declaraciones y desreferenciaciones:

- a) `double *var1;`
- b) `int function(int *arg1, void **arg2);`
- c) `int a = *ptr1 + *ptr2;`

o bien, pegados al tipo de la variable or argumento:

- a) `double* var1;`
- b) `int function(int* arg1, void** arg2);`
- c) `int a = *ptr1 + *ptr2;`

Además, después de todas las comas que haya en listas de inicialización, de los puntos y coma de los bucles `for` o de los argumentos de una función debe haber un espacio y un espacio debe rodear los paréntesis de estructuras de control (nota: llamada a función no es una estructura de control), es decir:

1. `for (int i = 0; i < 10; ++i) {`
2. `int list[] = {1, 2, 3, 4, 5, 6};`
3. `int res = pow(var1, var2);`

11.1.3. Estilo de las llaves

Hay varias combinaciones de estilo de las llaves que definen los bloques de código que componen los cuerpos de bucles, condicionales y demás estructuras de control. Hay dos estilos fundamentales: llaves K&R y Allman. Las primeras se ponen en la misma línea que la estructura de control que las necesita, es decir:

```
1 int main(void) {  
2     return 0;  
3 }
```

Programa 96: Ejemplo de llaves estilo K&R

Las llaves Allman se ven así:

```
1 int main(void)  
2 {  
3     return 0;  
4 }
```

Programa 97: Ejemplo de llaves estilo Allman

Hay otros estilos que combinan las tabulaciones de maneras distinta con las llaves, pero estas son las más importantes. Una manera que se puede ver habitualmente es que las definiciones de funciones (que son principalmente bloques de código de primer nivel) utilicen el estilo Allman, mientras que los bloques internos a éstas usen el estilo K&R. Lo más importante es que utilices un estilo o una combinación racional de ellos y lo mantengas en todo el proyecto. En caso de añadir código a una base existente, aplica la vieja regla de: «donde fueres haz lo que vieres», es decir, sigue el estilo que sigan en el proyecto en el que colabores.



11.1.4. Declaración de variables, estructuras y enumerados

Hasta ahora, cada declaración de variable la hemos hecho en una línea distinta, y cada miembro de una estructura también. Esto no es necesario, se pueden declarar todas las variables de un mismo tipo en la misma línea, e incluso inicializarse. Veamos un ejemplo sencillo. Vamos a reescribir el programa 58: Reserva, uso y liberación de un vector de vectores, pero omitiré parte de él porque no ha cambiado.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void)
5 {
6     int rows = 10, columns = 5, **matrix = NULL;
7     matrix = malloc(rows * sizeof(*matrix));
8
9     for (int ii = 0; ii < rows; ++ii) {
10         //resto del programa...
```

Programa 98: Declaración de variables en una misma línea

Como puedes ver en la línea 6, podemos declarar variables del mismo tipo en la misma línea, separando sus nombres por comas. Además, podemos incluso inicializarlas. Si vas a declarar punteros junto a variables de tipos no puntero, el asterisco que indica que esa variable es un puntero va pegado al nombre de la variable, como puede ver con `matrix`. En el caso de la inicialización del puntero, podría ir en la misma línea, pero lo he movido a otra para impedir que la línea sea excesivamente larga.

En las estructuras se puede hacer lo mismo, es decir, declarar todos los miembros del mismo tipo en la misma línea, con los enumerados también se puede hacer, pero es convención que cada identificador de una estructura vaya en su propia línea. Veamos un ejemplo de ambas cosas:

```
1 struct point_s {
2     double x, y;
3 }
4
5 enum week_days {
6     MONDAY,
7     TUESDAY,
8     WEDNESDAY,
9     THURSDAY,
10    FRIDAY,
11    SATURDAY,
12    SUNDAY
13 }
```

Programa 99: Declaración de miembros de un *struct* en una sola línea



11.1.5. Convenciones de nombres

Una de las cosas más importantes es que debes evitar el uso de los llamados **números mágicos**. Éstos son los valores literales que se incluyen en el código y que no se explican, se asignan a una variable y, en el peor de los casos, se usan en muchas partes del código. Si rescatas el ejemplo de mal código del programa 89, verás que nunca defino variables para definir las columnas y filas de la matriz. Esto lo hago intencionadamente, porque es una mala práctica. Como puedes ver en el ejemplo de buen código, lo primero que hago es definir variables para estos conceptos. En general, debes hacer esto, definir macros, variables y constantes para todo concepto o valor que haya en tu código, **especialmente si aparece varias veces**.

Además, hay una serie de convenciones que vamos a reunir aquí sobre los nombres. Cuando te expliqué el nombre que se le podía poner a una variable, te dije de que existían el *camel case* y el *snake case* para escribir nombres con varias palabras. En C lo normal es usar el segundo, es decir: `my_var`, así que en general será el que me hayas visto aplicar. Por otro lado, las constantes deben estar escritas en mayúscula, es decir: `const int LENGTH = 10;` es preferible a `const int length = 10;`.

En general, los tipos terminan en `_t`, como ya comentamos, ya sean estos renombramiento de tipos básicos o creación de estructuras. A los enumerados se les puede omitir su nombre de enumerado, es decir, es normal escribir:

```
1 typedef enum {
2     MONDAY ,
3     TUESDAY ,
4     WEDNESDAY ,
5     THURSDAY ,
6     FRIDAY ,
7     SATURDAY ,
8     SUNDAY
9 } week_days ;
```

Programa 100: Ejemplo de tipo enumerado con typedef

Las funciones deben tener nombres descriptivos, del mismo modo que sus argumentos, por ejemplo, es preferible:

```
1 int multiply_array_scalar(int *array, int array_size, int scalar);
```

Programa 101: Ejemplo de función descriptiva

a

```
1 int multi(int *a, int s, int n);
```

Programa 102: Ejemplo de función no descriptiva

En el primer caso queda claro qué hace la función y qué es cada argumento con un golpe de vista. En el segundo habría que acudir a la implementación para saber qué hace.



12. Lo que hay detrás de la compilación

Hasta ahora hemos dicho que crear un binario a partir de un archivo de código fuente se llama compilar. Si bien esto es una expresión correcta, no es del todo cierto. Hay varios procesos involucrados en lo que llamamos compilación:

1. **Preprocesado:** Prepara el código fuente para ser compilado, esto incluye eliminar comentarios, expandir macros, ejecutar directivas y eliminar saltos de línea. Más sobre macros, comentarios y directivas a continuación.
2. **Compilado:** Crea el binario de cada archivo de código fuente que se utilice en el proyecto.
3. **Enlazado:** Genera los binarios ejecutables propiamente dichos, para ello utiliza los binarios del paso anterior y crea enlaces entre ellos, ya sea juntándolos en el mismo archivo o simplemente indicando dónde están los que necesita.

Esto introduce muchos conceptos nuevos que explicaré a continuación, pero lo principal es que esta va a ser la sección en que aprendas a hacer programas con más de un archivo de código fuente. Esto es útil porque, como has visto, en el momento en que un programa se hace un poco complejo, empieza a hacerse confuso subir y bajar por el fichero de código buscando lo que necesitas o modificando cosas. Por esto, todo proyecto serio de programación en C contiene varios archivos de código fuente. Algunos proyectos muy grandes pueden llegar a tener hasta decenas de miles o cientos de miles de archivos de código fuente. Voy a explicar ahora los tres procesos.

12.1. Preprocesado

El preprocesado es la etapa que se realiza más discretamente, porque todo archivo de código fuente que se compile pasa primero por ella sin que tengamos que hacer nada. En la introducción de esta sección he comentado que se realizan varias tareas, siendo la primera la eliminación de comentarios. Ya es hora de que te explique qué son los comentarios. Un comentario es una herramienta que nos permite introducir cualquier texto dentro del código de tal modo que podamos explicar cosas, apuntar datos o cosas así. Veamos un ejemplo. Voy a incluir comentarios en el código del programa 37: Ejemplo de resolución de distancia entre puntos usando estructuras



```

1  #include <stdio.h> //for printf
2  #include <math.h>  //for sqrt
3
4  //Definimos la estructura punto bidimensional.
5  struct point_s {
6      double x;
7      double y;
8  };
9
10 int main(void)
11 {
12     struct point_s A;
13     struct point_s B;
14
15     A.x = 1.1;
16     A.y = 3.2;
17     B.x = 2.3;
18     B.y = 5.4;
19
20     double diff_x = A.x - B.x;
21     double diff_y = A.y - B.y;
22     /*Recordemos que la distancia
23     entre dos puntos es la raíz
24     cuadrada de la suma de los cuadrados
25     de la diferencia entre sus coordenadas*/
26     double distance = sqrt(diff_x * diff_x + diff_y * diff_y);
27
28     printf("P1 : [%f, %f]\n", A.x, A.y);
29     printf("P1 : [%f, %f]\n", B.x, B.y);
30     printf("Distance: %f\n", distance);
31 }

```

Programa 103: Example of program with comments

Como habrás adivinado, los textos que salen en verde son los comentarios. Se ve claramente que son cualquier texto que queramos, y que no tienen nada que ver con código en C. El preprocesador los eliminará antes de pasarle el programa al compilador. Hay dos tipos de comentarios:

1. Comentarios de una línea: Se empiezan con `//`, son comentarios que hacen que el preprocesador elimine todo después de las barras inclinadas, éstas incluidas hasta el final de la línea. Se pueden poner varios en líneas correlativas.
2. Comentarios multilínea: Empiezan por `/*`, y **terminan con** `*/`. El comentario ignorará saltos de línea y cualquier carácter que haya hasta que se encuentre el final, como puedes ver en el ejemplo.

En general, puedes poner los comentarios que consideres oportunos, sobre todo, se comentan funciones que tengan algoritmos difíciles o cosas que no sean evidentes. Como nota, al ser el inglés la *lingua franca* de esta época y estar además los lenguajes de programación basados en él, los comentarios del código se hacen siempre en inglés en entornos profesionales.

El siguiente paso es la expansión de macros. Y te estarás preguntando qué es una macro. Una macro es una constante simbólica que definimos en el código y que el preprocesador sustituirá por su valor allá donde la escribamos. Veamos un ejemplo.



```
1 #include <stdio.h>
2
3 #define LIST_LENGTH 100
4
5 int main(void)
6 {
7     int list[LIST_LENGTH];
8     for(int ii = 0; ii < LIST_LENGTH; ++ii){
9         list[ii] = ii;
10    }
11 }
```

Programa 104: Creación de macros

En la línea tres puedes ver la única novedad que tenemos aquí, la directiva `define`. Las directivas empiezan con almohadilla (`#`). Y ya vamos desvelando uno de los misterios que más tiempo llevan en nuestros programas, las líneas que contienen `#include` son, efectivamente, una directiva. Pero hablemos de la directiva `#define`. Esta directiva define (lógico, claro) un símbolo que será sustituido por otro por el preprocesador, en este caso, hemos definido que allá donde escribamos `LIST_LENGTH`, el preprocesador lo sustituirá por `100`. Un uso común para esto es, como puedes ver, definir el tamaño de arrays. ¿Quiere decir esto que cada array debe tener su macro correspondiente? No, sólo si se va a usar en varios sitios o tiene un significado, por ejemplo, el tamaño de una lista de pruebas que vas a hacer al mismo código.

Como las macros se expanden **antes** de la compilación, sí que puedes definir el tamaño de un array y la vez inicializarlo. Esto tiene una ventaja: cuando inicializas un array, pero faltan elementos, el resto se rellena a 0 automáticamente, así que podemos escribir nuestros programas como:

```
1 #include <stdio.h>
2
3 #define LIST_LENGTH 65536
4
5 int main(void)
6 {
7     int list[LIST_LENGTH] = {};
8     for(int ii = 0; ii < LIST_LENGTH; ++ii){
9         if(list[ii] != 0){
10             printf("Hay elementos que no son 0.\n");
11             break;
12         }
13     }
14 }
```

Programa 105: Uso de define con arrays



Ya te expliqué que una variable no se podía usar para indicar el tamaño de un array que inicializaras con una lista de inicialización porque esa variable podía valer lo que fuera (el compilador no puede saberlo, porque eso sólo se sabría al ejecutar el programa). Quizás te preguntes si una variable constante, por ejemplo, `const int LENGTH=100;` podría ser usada para esto. No, una variable marcada con `const` sigue siendo una variable, porque hay mecanismos, algunos normales y otros producto de abusos del lenguaje C, que permiten que el valor de esa variable cambie. En cuanto al hecho de que cuando la lista de inicialización es más pequeña que el tamaño del array, se inicializan a cero, haz una prueba: compila y ejecuta ese programa de ejemplo algunas veces (no compiles cada vez, sólo necesitas compilar si cambias el código). Verás que jamás se imprime el mensaje. Ahora, elimina la inicialización del array, deja la línea 7 como `int list[LIST_LENGTH];` y después de compilar verás que casi siempre se imprimirá. Esto es porque, recuerda, lo que no inicializas tiene valores aleatorios.

Pero las macros tienen una potencia increíble, porque admiten argumentos. Habrás notado que hay cosas que hacemos muy comúnmente, como por ejemplo hallar el cuadrado de un número, para eso podemos invocar a la función `pow`, lo que implica compilar enlazando la biblioteca correspondiente, como vimos, o escribiendo `var*var`, con una macro, podemos hacerlo más legible.

```
1 #include <stdio.h>
2
3 #define SQUARE(a) a*a
4
5 int main(void)
6 {
7     for(int ii = 0; ii < 10; ++ii){
8         printf("%d^2 = %d\n", ii, SQUARE(ii));
9     }
10 }
```

Programa 106: Uso de macro con argumentos

Como puedes ver, este programa funciona como se espera. Siendo esto así de práctico, ¿por qué no se usan macros para todo? El primer motivo es que se pueden convertir en un auténtico infierno para encontrar errores o problemas en tus programas. Esto es porque, recuerda, no son más que una sustitución simbólica. Imagínate que hiciéramos un bucle con macros (aunque esto no es completamente posible), al final, ese bucle se traduciría en tantas líneas como iteraciones tenga el bucle. Así que si alguna de ellas es incorrecta, tienes que depurar sobre líneas que no existen ni puedes ver porque sólo existen después del preprocesado. Además, al ser un copiar y pegar automático, se pueden producir problemas porque no se comprueban tipos ni se respetan preferencias de operadores por ser una sustitución simbólica. Cambiemos un poco el programa anterior:

```
1 #include <stdio.h>
2
3 #define SQUARE(a) a*a
4
5 int main(void)
6 {
7     for(int ii = 0; ii < 10; ++ii){
8         printf("%d^2 = %d\n", ii, SQUARE(ii+1));
9     }
10 }
```

Programa 107: Ejemplo de error por una macro



Si lees este programa, pensaríamos que debería imprimir: 1, 2, 4, 9... Pero no es lo que pasa, imprime: 1, 3, 5, 7, 9... Y esto es así porque esta macro está mal escrita, haz el ejercicio de sustituir `a` en la macro por `ii+1` mentalmente, verás que sale: `ii+1*ii+1`. Eso no es lo que querías escribir. La solución a esto es escribir `a` entre paréntesis en la macro, quedando `#define SQUARE(a) (a)*(a)`, pero aunque en este caso sea sencillo, sirva este ejemplo para demostrarte que las macros son peligrosas. Puedes usar macros con argumentos, pero se recomienda que sea para cosas sencillas, por ejemplo este cuadrado de un número, un valor absoluto, etc.

Además, hay una versión de las macros que pone como una cadena de texto lo que se introduzca como argumento. Se hace con la almohadilla de nuevo, veamos el ejemplo.

```
1 #include <stdio.h>
2
3 #define PRINT_INT(a) printf("#a=\"%d\\n", a);
4
5 int main(void)
6 {
7     for(int ii = 0; ii < 10; ++ii){
8         PRINT_INT(ii);
9     }
10 }
```

Programa 108: Uso de macros con cadenas de texto

Como puedes apreciar, al poner `#a` le estamos diciendo que sustituya `a` por `"a"`. Como vimos en el ejemplo de cómo separar una línea en varias (programa 92), dos cadenas de texto escritas como literales, juntas, se convierten en una, es decir: `"Hola" "Mundo"` es equivalente a `"Hola Mundo"`. Y después hemos escrito `a` para que la sustituya de manera normal.

Esta es una aplicación de macros para convertir los arguments en strings, en general, para poder crear una macro que convierta cualquier argumento que se le introduzca en un literal de string, se suelen utilizar dos macros, vamos a ver un ejemplo de cómo se podría utilizar.

```
1 #include <stdio.h>
2
3 #define STR_HELPER(a) #a
4 #define STR(a) STR_HELPER(a)
5 #define PRINT_INT(integer) printf(STR(integer)" = %d\\n", integer)
6
7 int main(void)
8 {
9     for (int ii = 0; ii < 10; ++ii) {
10         PRINT_INT(ii);
11     }
12 }
```

Programa 109: Macro para convertir a string

Aquí se puede ver cómo se utilizan las macros para imprimir primero el nombre de la variable y después el valor. Esto, además, es útil cuando se tienen macros con valores numéricos que se quieren concatenar con un string, un caso sería este:



```
1 #include <stdio.h>
2
3 #define STR_HELPER(a) #a
4 #define STR(a) STR_HELPER(a)
5
6 #define ARRAY_SIZE 10
7
8 int main(void)
9 {
10     int array[ARRAY_SIZE];
11     printf("The size of the array is "STR(ARRAY_SIZE)".\n");
12 }
```

Programa 110: Conversión de macros numéricas a string

El detalle en que quiero que te fijes es que, en este caso, al ser una macro y no una variable lo que se le pasa a la macro `STR`, se sustituye por su **valor** y no por su nombre. Puedes utilizar esto para definir strings, independientemente de que se use con `printf` o no.

La siguiente directiva que nos importa es, por fin, la directiva `include`. Es la directiva que hemos usado para poder utilizar una variedad de funciones como `printf` o `malloc`. Esto es por lo que hace la directiva: incrusta el contenido de un archivo de código fuente en el tuyo. Sí, lo has oído bien, cuando escribes `#include <stdio.h>`, lo único que estamos haciendo es pegar en este archivo de código los contenidos de un archivo distinto, en este caso, `stdio.h`. Es extraño que sea un archivo con extensión `.h`, si todos los programas los hemos escrito en un archivo acabado en `.c`. Esto es porque este archivo es un archivo de **cabeceras**, en este tipo de archivo, del que veremos más adelante ejemplos y te enseñaré a hacer, se escriben sólo definiciones, es decir: declaraciones de funciones, declaraciones de variables globales, de tipos nuevos...

Pero, de momento, puedes experimentar con esta directiva escribiendo estos dos archivos: `main.c`, el que ya teníamos, y `other.c`. En el primero escribe esto:

```
1 //main.c
2 #include <stdio.h>
3 #include "other.c"
4
5 int main(void)
6 {
7     int a = 10;
8     printf("a ahora vale: %d\n", a);
9     multiply(&a, 2);
10    printf("a ahora vale: %d\n", a);
11 }
```

Programa 111: Ejemplo de directiva `include`, archivo principal

Como puedes ver, al incluir `other.c`, estamos usando comillas en vez de los signos de menor que y mayor que. Esto es porque cuando a la directiva le indicamos el nombre del archivo entre menor que y mayor que, los busca en los directorios predefinidos que el sistema tiene para cabeceras. Si pones comillas, los busca en donde está este archivo de código fuente que lleva la directiva `include`. Por ello, cuando utilices archivos que no estén en esos directorios, debes tener esto en cuenta. En el archivo `other.c` irá este contenido:



```
1 //other.c
2 void multiply(int* a, int b){
3     *a *= b;
4 }
```

Programa 112: Ejemplo de directiva include, archivo incluido

Si miras el contenido de ambos archivos, simplemente estamos «sacando» la función a otro archivo y hemos usado la directiva para que desde el archivo principal se escriba todo el contenido del segundo en el primero. No obstante; una regla de oro es que nunca debes incluir archivos de código fuente, sino esos archivos de cabecera que te presenté antes. Sin embargo; para poder hacer esto, debemos primero ver los dos pasos siguientes de la creación del binario.

Antes de llegar allí, sin embargo; quiero que veas unas cuantas directivas muy interesantes: `ifndef`, `ifdef` y `endif`, que siempre van juntas. Su nombre es más o menos explicativo, pero se trata de lo siguiente, estas directivas comprueban si una macro está definida, y si según esté o no, el código entre `ifdef/ifndef` y `endif` se incluirá o no. Veamos un ejemplo sencillo.

```
1 //main.c
2 #include <stdio.h>
3
4 int main(void)
5 {
6     #ifndef MY_MACRO
7         printf(";Hola mundo!\n");
8     #else
9         printf(";Adiós Mundo!\n");
10    #endif
11 }
```

Programa 113: Uso de directivas ifdef e ifndef

Aquí puedes ver que usamos `ifndef`, `else` y `endif`, creo que es muy explicativo, pero simplemente, cuando `MY_MACRO` **no** está definida, el código que resulta del preprocesado imprimirá «¡Hola, mundo!»; cuando sí lo está (en el `else`), el código resultante imprimirá «¡Adiós Mundo!».

Las macros se pueden definir desde la línea de comandos a la hora de compilar (ten en cuenta que aquí ni siquiera nos interesa su valor, tan solo si existe o no) así que esto permite modelar nuestra compilación a distintos entornos. Para definir una macro en el comando de compilación con gcc simplemente agregar: `-DNOMBRE_MACRO=valor`, por ejemplo, en el caso que acabo de presentarte: `-DMY_MACRO=0`. Pruébalo, compila el programa anterior con esta orden: `gcc -o main.exe main.c` y ejecútalo, imprimirá «¡Hola, mundo!», si lo compilas con esta: `gcc -o main.exe main.c -DMY_MACRO=0`, imprimirá «¡Adiós, mundo!». Si hubiéramos usado `ifdef` en lugar de `ifndef`, el caso sería el contrario.

Por ejemplo, se usa para hacer que existan o no impresiones por pantalla que sólo deben verse en compilaciones usadas para hallar errores en el código, pero no en la compilación final, la que se vendería, por ejemplo. Estas directivas son muy importantes por algo que veremos en la sección siguiente.

12.2. Compilación de objetos

El siguiente paso es el de la compilación propiamente dicha. Es un paso muy sencillo, pero con implicaciones muy interesantes. Lo que has venido haciendo hasta ahora es compilar un binario de un único fichero de código fuente. Pero si ejecutaras la orden de compilación de otro modo:

```
$ gcc -c <fichero de código fuente>
```




Esto generará un fichero con el mismo nombre que el de código, pero con la extensión `.o`. Esto es un fichero de código **objeto**. Estos ficheros son un punto intermedio entre el código fuente en C y el ejecutable, es un punto intermedio porque estos archivos tienen aún información sobre símbolos (variables, funciones...). Si compilas con esta opción el programa Hola Mundo que escribimos al principio, verás que **no** puedes ejecutar este archivo de código objeto, pero estos archivos son los ingredientes que usaremos en nuestra marmita para poder componer por fin un ejecutable con varios códigos objeto.

Para hacer esto quiero que recuperes los ficheros `main.c` y `other.c` y **elimines** la línea donde incluías `other.c`. Si ahora intentas compilar el archivo `main.c` con la orden `gcc -c main.c` el compilador lanzará un *warning* como este:

```
main.c: In function 'main':
main.c:8:5: warning: implicit declaration of function 'multiply' [-Wimplicit-function-declaration]
    8 |     multiply(&a, 2);
      |           ^~~~~~
```

Nos dice que hemos declarado implícitamente la función `multiply`, eso significa que el compilador no ha encontrado la definición de la función en el código fuente, es decir, te advierte que esta función se queda pendiente de existir tal y como está usada aquí (nombre, tipo de retorno y argumentos). Quizás esto te parezca una locura, pero así es, en C se pueden declarar funciones simplemente llamándolas porque se espera que **estén en otros códigos objeto**. De ahí que los códigos objeto aún guarden información sobre los nombres de las funciones, porque así cuando los juntas todos, se juntan las piezas de este puzzle que es nuestro ejecutable.

Ahora, debemos crear el código objeto del otro archivo, simplemente ejecuta el mismo comando, pero sobre `other.c`. Ahora tendrás dos archivos: `main.o` y `other.o`. Ya tenemos las piezas, para juntarlas, simplemente ejecutarías:

```
gcc -o main.exe main.o other.o
```

Esto generará un ejecutable que podrás ejecutar normalmente y verás que, efectivamente, funciona. Como comprenderás, es muy mala idea que cuando compilas un archivo de código fuente haya funciones que existen simplemente porque las usas, imagínate el horror de tener que compilarlo todo para luego descubrir que has llamado mal a una función (por ejemplo, con un número equivocado de argumentos). Si has experimentado con los ejercicios, habrás visto que muy sencillo cometer esos errores y que el compilador es, de hecho, tu mejor aliado para señalártelos. En la solución a esto es donde entran los archivos de cabecera y los archivos de código. Como ya vimos en la sección 7, se pueden declarar en un sitio y definir en otro, y es aquí donde esto nos resulta más útil. Todas las funciones pueden ir declaradas en un fichero de cabeceras, de tal manera que el compilador **conozca** la cabecera de la función y pueda generar el objeto comprobándola. Para ello, vamos a escribir el archivo de cabeceras que corresponde a `other.c`, es decir, `other.h`. Es muy sencillo, como sólo tenemos una función, el archivo será tal que así:

```
1 //other.h
2 void multiply(int* a, int b);
```

Programa 114: Archivo de cabecera

Hay que cambiar `other.c`, simplemente **incluyendo** la propia cabecera. Esto lo hacemos así porque, si nos equivocamos al definir la función, por ejemplo, imagínate que se nos olvida el asterisco, el compilador nos dirá que hemos redefinido la función porque, al haberla declarado en dos maneras distintas, sería como dos funciones con el mismo nombre, y eso no está permitido. Quedaría así:



```
1 //other.c
2 #include "other.h"
3 void multiply(int* a, int b) {
4     *a *= b;
5 }
```

Programa 115: Archivo de definiciones con cabecera incluida

Finalmente, en `main.c` incluiremos el archivo de cabeceras. Quedaría así:

```
1 //main.c
2 #include <stdio.h>
3 #include "other.h"
4
5 int main(void)
6 {
7     int a = 10;
8     printf("a ahora vale: %d\n", a);
9     multiply(&a, 2);
10    printf("a ahora vale: %d\n", a);
11 }
```

Programa 116: Archivo principal con cabeceras incluidas

Ahora, para generar el binario, simplemente tenemos que generar ambos objetos y después el ejecutable con estas órdenes:

```
gcc -c main.c
gcc -c other.c
gcc -o main.exe main.o other.o
```

Si observas, el compilador ya no nos lanza el *warning* de que hemos usado la función sin una declaración previa. Pero hay un problema, el código de `other.h` será incrustado en todos los archivos que utilicen la función, porque en todos ellos estará la directiva de inclusión. Si dejamos esto tal y como está, no podríamos usar la función en otro archivo, porque el compilador vería esto como una redefinición. Incluyo aquí todos los archivos para que tengas toda la información.



```
1 //point.h
2 struct point_s{
3     double x;
4     double y;
5 };
6
7 typedef struct point_s point_t;
8
9 double distance(const point_t* a, const point_t* b);
```

Programa 117: Ejemplo de redefinición – point.h

```
1 //point.c
2 #include "circle.h"
3 #include <math.h>
4 #include <stddef.h>
5
6 double distance(const struct point_s* a, const struct point_s* b) {
7     double res = 0.0;
8     struct point_s origin = { .x = 0.0 , .y = 0.0 };
9     if (NULL == a) {
10         a = &origin;
11     }
12     if (NULL == b) {
13         b = &origin;
14     }
15     double diff_x = a->x - b->x;
16     double diff_y = a->y - b->y;
17     res = sqrt(diff_x * diff_x + diff_y * diff_y);
18     return res;
19 }
```

Programa 118: Ejemplo de redefinición – point.c

```
1 //circle.h
2 #include "point.h"
3
4 #define PI ((double)3.141592)
5
6 struct circle_s {
7     point_t center;
8     double radius;
9 };
10
11 typedef struct circle_s circle_t;
12
13 double area(const circle_t* c);
14
15 double diameter(const circle_t* c);
```

Programa 119: Ejemplo de redefinición – circle.h



```
1 //circle.c
2 #include "circle.h"
3 double area(const circle_t* c) {
4     return c->radius * c->radius * PI;
5 }
6
7 double diameter(const circle_t* c) {
8     return 2 * PI * c->radius;
9 }
```

Programa 120: Ejemplo de redefinición – circle.c

```
1 //main.c
2 #include <stdio.h>
3 #include "point.h" //for using points
4 #include "circle.h" //for using circles
5
6
7 int main(void)
8 {
9     point_t a = {1.1, 2.3};
10    point_t b = {4.1, 3.3};
11    printf("La distancia entre a y b es: %f\n", distance(&a, &b));
12
13    circle_t circle = {a, 1};
14    printf("El círculo tiene un área de: %f\n", area(&circle));
15 }
```

Programa 121: Ejemplo de redefinición – main.c

Si intentas generar los objetos de todos los archivos de código fuente con estas órdenes (los archivos de cabeceras no se compilan):

```
gcc -c point.c
gcc -c circle.c
gcc -o main.exe main.c circle.o point.o -lm
```

Puedes ver que en la última orden hemos creado el ejecutable indicando el nombre del fichero de código fuente en vez del objeto para main.c. Esto es una manera de ahorrarte un paso en la compilación del archivo principal (el que contiene la función main). Verás que a la hora de compilar main.c éste da errores sobre redefiniciones. Esto es porque, si sigues el «rastreo» de las inclusiones, verás que main.c incluye circle.h y éste incluye point.h. Por otro lado, el propio main.c incluye point.h, esto provoca que el contenido de éste último esté presente dos veces. Si bien en este ejemplo se podría solucionar sencillamente eliminando la inclusión de point.h en el archivo principal, pero no podemos hacerlo, porque entonces no habría modo de usar las estructuras definidas en esa cabecera aquí si decidimos que no queremos utilizar circle.h, habría que añadirlo aquí de nuevo, es decir, se crearía la necesidad de ir monitorizando cada cambio en includes que se haga. Además, es mejor que cada fichero incluya los archivos de los que depende aunque estos estén a su vez en otros, así se sabe con qué se relaciona. En proyectos con más archivos es una tarea imposible rastrear estos problemas para comprobar si includes un archivo más de una vez. Por esto, se usan los llamados *include guards*.



Éstos son simplemente el uso de directivas de tipo `ifndef` `endif` para hacer que, si un archivo de cabeceras se incluye más de una vez, las repeticiones sean archivos vacíos. Es costumbre que todos los archivos de cabeceras de un proyecto lleven uno, así que para cuando uses varios archivos de código te recomiendo que te acostumbres desde ahora a usarlos. Veamos cómo añadir un *include guard* a `point.h`

```
1 //point.h
2 #ifndef POINT_H
3 #define POINT_H
4
5 struct point_s{
6     double x;
7     double y;
8 };
9
10 typedef struct point_s point_t;
11
12 double distance(const point_t* a, const point_t * b);
13
14 #endif
```

Programa 122: Ejemplo de *include guard*

Como puedes ver, lo que se hace es encerrar todo el contenido del archivo en un condicional del preprocesador. Si la macro `POINT_H` no está definida, la definiremos, y con ella todo el código de la cabecera. Así, cuando se incluya por segunda vez, como dicha macro ya está definida, el `ifndef` no se cumplirá y, a efectos del compilador, este archivo estará vacío. El nombre de la macro suele ser el nombre del archivo, sustituyendo los puntos por guiones bajos, si usaras directorios dentro de tu proyecto, lo ideal sería que la macro tuviera la ruta completa del archivo relativa al directorio donde guardes tu proyecto, por ejemplo, si este archivo estuviera en: `proyecto/lib/math/geometry/include`, el nombre de la macro del *include guard* debería ser: `LIB_MATH_GEOMETRY_INCLUDE_POINT_H`.

12.3. Enlazado

El enlazado es la fase final del proceso de creación de un ejecutable. En él, lo que se hace es juntar los códigos objeto y las diferentes **bibliotecas** (en inglés: *libraries*) necesarias para el funcionamiento del ejecutable. Ya sabemos qué son los códigos objeto, y ya has usado varios para crear un ejecutable, pero ahora vamos a centrarnos en las bibliotecas.

Una biblioteca es, en términos conceptuales, un conjunto de funcionalidades que se compilan y distribuyen en un paquete que el usuario de la misma puede utilizar en sus programas. La principal ventaja es que el código fuente de una biblioteca es prácticamente imposible de reconstruir a partir de la misma, y, sobre todo, de una manera comprensible. Esto es una ventaja porque el código fuente está sujeto a propiedad intelectual, y, si bien es común que existan proyectos de código abierto, muchas bibliotecas comerciales se distribuyen sin acceso al código.

Además, las bibliotecas trasladan la responsabilidad del código de las mismas a quien las vende o distribuye, descargando al usuario de éstas de la tarea de fabricar el binario cuando cambie el código. Son una herramienta de distribución de *software* imprescindible. De hecho, ya has usado bibliotecas, todas las funciones que has usado hasta ahora que venían dadas simplemente poniendo alguna directiva de inclusión de una cabecera (`malloc`, `printf`...) residen en distintas bibliotecas **proporcionadas como parte de tu sistema operativo Linux**. Cuando el sistema operativo se actualiza, estas bibliotecas podrían cambiar y por tanto su funcionalidad verse actualizada. Esto puede implicar que necesites recompilar tus programas para ver esos cambios o no, dependiendo de qué tipo de bibliotecas utilices, habiendo dos tipos:



1. Bibliotecas dinámicas: Son aquéllas que se cargan en memoria cuando el programa se ejecuta, de ahí su nombre, pues se cargan sólo cuando son necesarias.
2. Bibliotecas estáticas: Son aquéllas que se combinan con el binario en tiempo de compilación. Esto implica que cuando se cambia una biblioteca de este tipo se deben recompilar los binarios que la usen.

Las bibliotecas, como el ejecutable, **son gestionadas por el sistema operativo**, es decir, los ejecutables que utilicen bibliotecas solicitarán al sistema operativo (sin que el programador deba hacer nada a nivel de código fuente) que las cargue cuando sea necesario. Además, es él el que gestiona las versiones, permitiendo al programador de un binario indicar que su ejecutable sólo funcionará con determinada versión de una biblioteca dinámica pero no con otras.

En general, cuando se generan bibliotecas se utilizan herramientas que automatizan la compilación de todos los códigos objeto y el enlazado de bibliotecas. Pero voy a mostrarte cómo se haría «a mano».

Supongamos que queremos hacer una biblioteca con nuestra estructura punto y nuestra estructura círculo y que la vamos a llamar `libGeometry`. Es convencional que todas las bibliotecas empiecen por `lib`. El primer paso es generar el código objeto, pues de él nos valemos tanto para crear ejecutables como para crear bibliotecas.

```
$ gcc -c point.c
$ gcc -c circle.c
```

Una vez hecho eso, creamos la biblioteca, para ello utilizamos este comando:

```
gcc -shared -o libGeometry.so point.o circle.o
```

Utilizamos la opción `shared` para indicar que estamos compilando una biblioteca dinámica. Esto creará nuestro archivo de biblioteca, `libGeometry.so`, ahora podemos compilar el ejecutable utilizando la biblioteca en vez de los códigos objeto directamente. Para ello se usaría este comando:

```
$ gcc -L. -Wl,-rpath=. -o main.exe main.o -lGeometry -lm
```

Este comando es un poco complicado, la opción `-L` nos permite indicar en qué fichero deben buscarse las bibliotecas para la compilación, esto se hace indicando la ruta punto (`.`), que significa este directorio. Por otro lado, la opción `-Wl,-rpath=.` nos permite indicar dónde debe buscarse el archivo de la biblioteca en el momento de la ejecución, igualmente, escribimos punto. El resto del comando es igual, pero añadimos el enlazado de las dos bibliotecas: la nuestra y la biblioteca matemática, con la opción `-lm`. Ahora, si ejecutas el programa con el comando `./main.exe`, funcionará perfectamente. Para comprobar que la biblioteca se enlaza dinámicamente, prueba a eliminarla, puedes hacerlo con el comando `rm libGeometry.so`. Si ahora ejecutas el programa, no funcionará, porque lo buscará en tiempo de ejecución.

En el caso de una biblioteca estática, éstas se crean de manera más sencilla, crea los objetos como antes, y ahora crea la biblioteca estática:

```
$ ar rvs libGeometry.a point.o circle.o
```

Ahora se puede compilar incluyendo la biblioteca (tenemos que seguir añadiendo `-lm` porque usamos la función `sqrt` que está en la biblioteca matemática):

```
$ gcc -o main.exe main.o libGeometry.a -lm
```

Para comprobar que es estática, eliminar `libGeometry.a` y verás que el ejecutable sigue funcionando. Esto es porque la biblioteca se incrusta en el momento en que realizas el ejecutable con todo sus códigos objeto. Esto hace que el resultado sea similar a simplemente utilizar todos los códigos objeto, pero sigue permitiendo la **distribución** sencilla del *software*. Además, así nos permite crear unidades conceptuales de software mayores que los códigos objeto de un único fichero, simplificando nuestros procesos de creación y distribución de los programas.



13. Funciones de la biblioteca estándar

La biblioteca estándar de C es una biblioteca que se incluye en todos los programas compilados en C en Linux. Esto es así porque contiene la mayoría de funciones que son implescindibles para realizar tareas básicas, por ejemplo: `malloc` y `free` están en ella. Aunque haya una cabecera que se llame `stdlib.h`, la mayoría de funcionalidades que se pueden usar sin enlazado de bibliotecas extra (como la matemática) están en la biblioteca estándar. En esta sección vamos a hablar de algunas de estas funciones y a demostrar por qué son necesarias incluso en niveles básicos.

13.1. Manejo de memoria

Aunque ya hemos visto las funciones más básicas para el manejo de memoria: `malloc` y `free`, hay otras funciones que son útiles que está bien que conozcas. Éstas son `calloc`, `realloc` y `memset`. Las dos primeras sirven para reservar memoria y la última sirve para poner a un mismo valor todos los bytes de una zona de memoria. Suelen verse mucho en programas con muchas operaciones de memoria.

La primera de ellas: `calloc` es una función que nos permite indicar la reserva de varios fragmentos de un tamaño concreto, que se reservarán en una zona contigua. Para empezar, veamos la declaración de la función:

```
1 void *calloc(size_t nmemb, size_t size);
```

Programa 123: Declaración de la función `calloc`

Como puedes ver, al igual que su «hermana» `malloc`, devuelve un puntero a `void`, que después podrá ser asignado a cualquier tipo de puntero. Sin embargo, recibe dos argumentos: el número de elementos que vas a reservar y el tamaño de los elementos. Si estás pensando que una llamada a esta función es equivalente a un `malloc` multiplicando los dos argumentos, tienes razón, pero hay **una diferencia**: la memoria reservada con `calloc` será inicializada a **ceros**. Esto provoca que algunos programadores utilicen una llamada a `calloc` con el primer argumento valiendo uno para reservar una zona de memoria que esté inicializada a ceros.

La siguiente función, `realloc`, es más interesante, es una función que nos permite **redimensionar** y automáticamente mover, si fuera necesario, una zona de memoria, su declaración es la siguiente:

```
1 void *realloc(void *ptr, size_t size);
```

Programa 124: Declaración de la función `realloc`

Como puedes ver, recibe un puntero como primer argumento, éste es el puntero de la zona de memoria **que queremos redimensionar** y, como segundo argumento, recibe el tamaño nuevo, **en bytes**, de la zona de memoria. En este caso hay dos posibilidades, que estés ampliando la zona inicial o que la estés encogiendo. En ninguno de los dos casos tienes garantizado que la zona de memoria sea la misma, así que debes comprobar que no ha devuelto `NULL` y además volver a guardar el valor del puntero, porque ha podido cambiar. Además, si el puntero que se le da a esta función es nulo, simplemente se comporta como `malloc`, esto es útil para poder usarla en bucles sin tener que mezclarla con una llamada a `malloc` inicial. Para ilustrar el uso de esta función, implementaremos con ella el programa 56 donde demostramos el primer uso de reserva de memoria dinámica.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int* erase_reps(int* array, int array_length, int* final_length) {
5     *final_length = 0;
6     int* result = realloc(NULL, sizeof(*result) * array_length);
7     for (int ii = 0; ii < array_length; ++ii) {
8         int unique = 1;
9         for (int jj = 0; jj < ii; ++jj) {
10             if (array[ii] == array[jj]) {
11                 unique = 0;
12             }
13         }
14         if (unique) {
15             result[*final_length] = array[ii];
16             ++(*final_length);
17         }
18     }
19
20     return realloc(result, *final_length * sizeof(*result));
21 }
22
23
24 int main(void)
25 {
26     int array[] = { 20,1,2,3,4,5,6,5,8,7,9,6,6,5,4,1,2,3,8,5,4,4,5,6 };
27     int length;
28     int* result = erase_reps(array, 24, &length);
29     if (NULL == result) {
30         printf("Ha habido un error de memoria\n");
31         return -1;
32     }
33     for (int ii = 0; ii < length; ++ii) {
34         printf("%d\n", result[ii]);
35     }
36     free(result);
37 }
```

Programa 125: Utilización de realloc

Si comparas ambos programas, verás que en la primera tuvimos que declarar un array para poder tener un sitio en el que guardar los datos hasta que sepamos cuántos hay que alojar. La desventaja de esto es que después tenemos que copiar los datos a la nueva zona que vamos a devolver y reservamos con malloc, en este caso en que usamos realloc, dejamos que sea el gestor de memoria del sistema operativo el que se preocupe de esto, y, siendo sensatos, es poco probable que al encoger una zona de memoria se mueva el contenido, así que podemos asumir que nos ahorraremos la copia la mayoría de veces. Ojo, repito: poco probable; pero posible.

La siguiente función es también interesante, memset es una función que se encuentra declarada en la cabecera string.h. Y esto tiene sentido porque, al poner todos los bytes de una zona de memoria al mismo valor, se usa mucho cuando se utilizan cadenas de texto, para permitir rellenar un texto con el mismo carácter. La función tiene esta forma:



```
1 void *memset(void *s, int c, size_t n);
```

Programa 126: Declaración de la función `memset`

El primer argumento es el puntero es donde vas a escribir los cambios, el segundo es el valor que vamos a escribir **en cada byte** y el último el número de bytes que se van a escribir. El típico ejemplo de uso para esto es cuando necesitas inicializar a ceros una zona de memoria. Esto es típico cuando reservas memoria para alguna estructuras cuyo valor necesitas controlar o que requiere que sea inicializada así (por eso `calloc` lo hace). El ejemplo que voy a poner, sin embargo, es más original. Imagina que quieres imprimir una barra de progreso en modo de texto que tenga este aspecto:

```
[#####.....]
```

Imagino que ves por dónde voy. Para que la barra se imprima de manera «bonita», tengo que presentarte un nuevo caracter especial: `\r`, que se llama retorno de carro, es decir, lleva el punto de impresión al principio de la línea, permitiéndote sobreescribirlo. Además, voy a usar una función llamada `usleep`, que nos permite pausar el programa durante algún tiempo, si no, el programa haría la secuencia muy deprisa. Finalmente, la función `fflush` te permite **forzar** a la terminal a que imprima caracteres que estén pendientes, esto lo hacemos porque, como no imprimimos ninguna línea nueva, la terminal imprimiría sólo cada cierto tiempo, fastidiando el efecto.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5
6 #define ARRAY_SIZE(array) ((sizeof((array)))/(sizeof((array)[0])))
7 #define BAR_LENGTH ((size_t)100)
8
9 void print_bar(int progress) {
10     //el resto de chars serán cero
11     char bar[BAR_LENGTH + 3] = { '[' };
12     //dejamos espacio al final para el último cero
13     bar[ARRAY_SIZE(bar) - 2] = ']';
14     //escribimos las almohadillas que indican completado
15     memset(bar + 1, '#', progress);
16     //escribimos los puntos que indican la parte sin completar
17     memset(bar + 1 + progress, '.', BAR_LENGTH - progress);
18     //imprimimos y volvemos al principio de la línea
19     printf("%s\r", bar);
20     //obligamos a la terminal a actualizarse inmediatamente
21     fflush(NULL);
22 }
23
24 int main(void)
25 {
26     for (int ii = 0; ii <= 100; ++ii) {
27         print_bar(ii); //imprimimos la barra
28         usleep((unsigned int)(2.50 * 100000)); //esperamos
29     }
30     //imprimimos una línea nueva para que el prompt salga
31     //en la siguiente
32     printf("\n");
33 }
```

Programa 127: Utilización de la función `memset`

Como puedes ver, la función que imprime una barra dado determinado progreso es muy simple. Debido a que aparecen varias cosas que no había explicado antes, he comentado el código exhaustivamente. Aquí lo hago en español, pero, como ya dijimos, en cualquier entorno profesional los comentarios se hacen en inglés.

Otra función muy útil es la que nos permite **copiar** lo que hay en una zona de memoria a otra. Esto es especialmente útil porque nos permite copiar de una parte a otra cualquier tipo de dato sin tener que hacer un bucle, cuando un algoritmo copia muchas veces de un sitio a otro, al final esos simples bucles pueden hacer el código más difícil de leer. Como con las demás, vamos a ver su declaración:

```
1 void *memcpy(void *dest, const void *src, size_t n);
```

Programa 128: Declaración de `memcpy`



Recibe tres argumentos, el primero es la zona de memoria donde copiaremos los datos, el segundo la zona de memoria desde la que los copiaremos y el tercero el número de bytes que queremos copiar. Para recordar qué argumento va primero (si el destino o el origen), yo recuerdo que funciona como una asignación, es decir, el destino va a la izquierda. Veamos un ejemplo, imagínate una función que nos permite insertar un elemento en la posición que queramos de un array.

```
1 int *insert_at(int *list,
2               int list_size,
3               int position,
4               int element)
5 {
6     int *res = malloc(sizeof(*res) * (list_size + 1));
7     memcpy(res, list, sizeof(*res) * position);
8     memcpy(res + position, &element, sizeof(*res));
9     memcpy(res + position + 1,
10           list + position,
11           sizeof(*res) * (list_size - position));
12     return res;
13 }
```

Programa 129: Utilización de la función `memcpy`

La función es bastante sencilla, simplemente copiamos desde la primera posición a la posición de inserción, copiamos el elemento que queremos insertar (esto lo podríamos hacer con un operador de asignación, pero ya que estamos en el negocio...) y después copiamos los elementos que vienen después del que queríamos insertar.

13.2. Manejo de cadenas de texto

Quizás hayas notado que, debido a que los *strings* en C son simplemente arrays y a que no podemos hacer una serie de cosas con ellos de manera sencilla, siempre ocurre que es complicado manejarlos. Si no, ya te comunico que utilizar cadenas de texto en C intensivamente es algo ligeramente (si no mucho) más engorroso que con otros lenguajes. Pero hay muchas funciones que nos ayudan a manejarlos. Veamos de qué funcionalidades disfrutamos:

1. Comparar cadenas y ordenarlas lexicográficamente
2. Saber la longitud de cadenas de texto.
3. Duplicar una cadena.
4. Crear cadenas nuevas con formato determinado.

Como puedes ver, tenemos una enorme cantidad de funcionalidades a nuestra disposición para manipular cadenas de texto. Estas funciones están todas en las cabeceras `string.h` y `stdio.h` (la misma que `printf`). Vayamos funcionalidad por funcionalidad.

En general, las variables básicas se comparan con el operador `==`, pero las cadenas de texto en C son punteros. Si hicieras una comparación con este operador, simplemente compararías las direcciones de ambas cadenas que, salvo que fueran la misma, nunca serían iguales. Supongo que, con lo que ya sabes del lenguaje, podrás deducir cómo se hace la comprobación: simplemente un bucle que compare carácter a carácter hasta que se llegue al final de uno de los dos, que debería ser el mismo número de iteraciones para que sean iguales. La función para hacer esto es `strcmp`.



En general, cuando comparas cosas en un lenguaje, invocas un operador o una función que devuelve cierto si son iguales y falso si son distintos. Por ejemplo: `a == b`. Pero en este caso, `strcmp` devuelve un número menor que cero si la primera cadena es anterior a la segunda en orden lexicográfico (alfabético), **cero si son iguales** y un número mayor que cero si la segunda es anterior a la primera. Veamos un ejemplo de su uso, vamos a realizar un programa que indique si una sucesión de cadenas está en orden alfabético.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 int main(int argc, char const* argv[])
6 {
7
8     if (argc < 3) {
9         printf("Necesitamos al menos dos palabra.\n");
10        return EXIT_FAILURE;
11    }
12
13
14    for (int ii = 1; ii < argc - 1; ++ii) {
15        if (0 < strcmp(argv[ii], argv[ii+1])) {
16            printf("Las palabras introducidas como argumento no"
17                " están en orden alfabético\n");
18            return EXIT_FAILURE;
19        }
20    }
21    printf("ok\n");
22    return EXIT_SUCCESS;
23 }
```

Programa 130: Ejemplo de uso de `strcmp`

El condicional es un poco confuso, pero suele pasar cuando utilizamos `strcmp`, queremos que los argumentos anteriores sean menores que los posteriores, por lo que debemos comprobar si el resultado de `strcmp` es mayor que cero (o, tal y como está escrito, si cero es menor que ese resultado) porque indicaría que es la palabra en la siguiente posición la que iría antes, es decir, que están desordenadas. Por otro lado, mira el bucle, empezamos desde la posición uno, porque en la cero está el nombre del programa y terminamos en la anterior a la última, porque dentro del bucle comprobamos la siguiente posición, y no hay siguiente a la última.

La siguiente función que nos ocupa es `strlen`, que nos dice la longitud de una cadena de texto, eso sí: **sin contar el carácter nulo**. Para empezar, antes de ver la *signature* de la función, vamos a hacer el ejercicio de hacer una función que realice esta sencilla tarea:

```
1 size_t my_own_strlen(const char* string) {
2     size_t iter = 0;
3     while ('\0' != string[iter]) {
4         iter++;
5     }
6     return iter;
7 }
```

Programa 131: Propia versión de `strlen`



La función es muy simple, pero quiero que notes una cosa que se ve bien en esta implementación (implementación es una manera de decir realización, es decir, cómo está hecho): si el *string* no contiene ningún carácter nulo, esta función leerá lo siguiente que haya en memoria sin parar, lo que, como ya sabes, suele provocar que los programas fallen y se cierren. Por esto, ten cuidado al usar la función, porque llamarla sobre un puntero a `char` sin un carácter nulo provocaría errores en el programa.

La *signature* de la función `strlen` de verdad es la siguiente.

```
1 size_t strlen(const char *s);
```

Programa 132: Declaración de la función `strlen`

Rápidamente, la función que duplica cadenas de texto es `strdup`, y su *signature* es ésta:

```
1 char *strdup(const char *s);
```

Programa 133: Definición de `strdup`

Simplemente es una función a la que le pasas por argumento la cadena que quieres duplicar y devuelve un puntero a una zona de memoria con este mismo contenido. Sin embargo, debes liberar los dos, si el primero se reservó con memoria dinámica, porque no es una operación que mueve, sino que copia.

Por ejemplo, un uso común para esta función es cuando estás creando una estructura de datos que almacene cadenas de texto, como ya vimos en el programa 61: Uso de punteros constantes como argumentos de función, es útil que ciertas estructuras de datos se manejen sólo mediante llamadas a funciones, por ello, en vez de hacer lo que hacemos allí: primero el `malloc` y después la copia, podemos usar `strdup`, para hacerlo en menos líneas. Un buen ejercicio sería que reescribieras ese programa utilizando esta nueva función.

Y llegamos al punto más importante de esta sección, el formateado de una cadena de texto a partir de variables. Esto ya lo has estado haciendo con la función `printf`. Para poder hacer esto, existen varias funciones:

1. `sprintf`: Permite hacer lo mismo que `printf`, pero a un puntero a `char`.
2. `fprintf`: Es la misma idea, pero con ficheros, del mismo modo de las funciones de escritura y lectura de archivos, recibe un puntero de tipo `FILE`.

De nuevo, este es un manual de C, y pretendo que siga siendo más o menos estrictamente eso, pero me es conveniente presentarte un ejemplo de caso de uso para estas funciones. En informática, a veces, para transmitir información, por ejemplo, por Internet o entre máquinas de cualquier modo realizamos un proceso que se llama **serialización**. Este proceso es la conversión de datos residentes en la memoria de alguna máquina a texto. Esto se hace porque las máquinas pueden usar maneras distintas para guardar información en su memoria, por ejemplo, hay ordenadores cuyos bytes están ordenados «al revés». Esto quiere decir que el número 10.669 que en hexadecimal es: 0x29AD se compone de dos bytes (recuerda, cada byte son dos dígitos hexadecimales), en la memoria de algunos ordenadores estará guardado como 0x29 AD y en otras como 0xAD 29. Para evitar este tipo de confusiones, convertiremos nuestros datos en cadenas de texto.

Volviendo al ejemplo del programa 61. Podríamos crear una función que creara esta serialización de la estructura, por ejemplo, en el caso de una persona llamada José Pérez Martínez, querríamos serializarlo así:

```
{
    "name": "José",
    "last_name_1": "Pérez",
    "last_name_2": "Martínez"
}
```



```
}
```

Veamos como quedaría la función (recuerda que sería añadida al programa que he citado antes) a la que llamaremos `person_to_string`. Esta función quedaría como:

```
1 char* person_to_string(const person_t *p){
2     char preliminar[1024] = {};
3
4     sprintf(preliminar,
5             "{\n"
6             "\t\"name\": \"%s\", \n"
7             "\t\"last_name_1\": \"%s\", \n"
8             "\t\"last_name_2\": \"%s\" \n"
9             "}",
10    p->name, p->last_name_1,
11    p->last_name_2);
12
13     return strdup(preliminar);
14 }
```

Programa 134: Ejemplo básico de `sprintf`

Para que se lea mejor, he escrito todos los argumentos de la función en una línea distinta. En el caso del segundo argumento, el formato, lo he dividido en varias, si observas un poco verás que no hay comas en esas líneas. Eso es porque dos literales de *string* escritos juntos son como uno solo. Ese «juntos» incluye si sólo los separan espacios en blanco. Ten en cuenta que esas líneas nuevas no aparecen en el *string*, por eso debemos poner `\n` al final igualmente. Además, observa como utilizamos un array para formatear el texto porque, como ya es habitual, no sabemos cuánto mide, una vez lo hemos formateado, usamos `strdup` para devolver una cadena del tamaño correcto.

No obstante este uso de la función es simple, aún queda un detalle: el valor que devuelve la función. Tanto `printf` como `sprintf` y todas las funciones de esta familia devuelven un entero que indica **el número de caracteres impresos** (sin contar el caracter nulo que incluyen para que el *string* resultante esté bien formado). Esto es de suma utilidad cuando quieres imprimir varias cosas en el mismo *string*. Veamos un ejemplo, imagina un programa que, análogamente, serializa un array de enteros, si el array es 1,2,3,4,5 la serialización quedaría como:

```
[
    1,
    2,
    3,
    4,
    5
]
```

La solución más inmediata sería usar `sprintf` para imprimir todos los enteros, pero tenemos un problema, no sabemos cuántos números hay, así que no podemos escribir en el formato que pide la función los especificadores necesarios. Por eso vamos a imprimir con un bucle que utilice el valor de retorno de la función.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define MAX_STRING_SIZE ((size_t) 65536)
6 #define ARRAY_SIZE(array) ((sizeof((array)))/(sizeof((array)[0])))
7
8 char* integer_array_to_string(const int* array, size_t array_size)
9 {
10     char res[MAX_STRING_SIZE] = {};
11     int printed_chars = 0;
12
13     printed_chars += sprintf(res + printed_chars, "[\\n");
14     for (size_t ii = 0; ii < array_size; ++ii) {
15         char* separator;
16         if (ii != array_size - 1) {
17             separator = ",\\n";
18         }
19         else {
20             separator = "\\n";
21         }
22         printed_chars += sprintf(res + printed_chars,
23                                 "\\t%d%s", array[ii], separator);
24     }
25     printed_chars += sprintf(res + printed_chars, "];");
26
27     // desbordamiento
28     if (printed_chars >= MAX_STRING_SIZE) {
29         return NULL;
30     }
31
32     return strdup(res);
33 }
34
35 int main(void)
36 {
37     int list[] = {1,2,3,4,5,6};
38     char *serialization = integer_array_to_string(list,
39                                                  ARRAY_SIZE(list));
40     printf("%s\\n", serialization);
41     free(serialization);
42 }
```

Programa 135: Ejemplo de uso avanzado de sprintf

Si ves cómo hemos escrito la función, aprovechamos este valor de retorno para concatenar cada impresión, el mecanismo es muy sencillo, si hemos impreso, por ejemplo, tres letras, debemos sumar a la posición inicial ese número. Además, usamos un condicional dentro del bucle para impedir que se imprima una coma en el último elemento. Después del bucle, imprimimos el corchete de cierre y, finalmente, hay un condicional que comprueba que no hemos impreso más caracteres de los que habíamos previsto. Nota que utilizamos `>=` porque debemos provisionar que el último char es un cero (`\0`).



No obstante, tenemos un problema, aunque somos capaces de decir cuándo se ha desbordado el *buffer* inicial que proveímos, **no podemos impedir que se desborde**. Esto tiene implicaciones muy serias, porque una vez escribes en zonas de memoria en que no deberías, no sabes qué puede ocurrir. Por suerte, hay una variación de la función que estamos usando que nos ayudará en este propósito. Ésta se llama `snprintf`, y tiene la misma *signature* que la anterior, pero añade un argumento: el número **máximo** de caracteres que debe imprimir, de este modo, nunca desbordará nuestro *buffer*. Veamos cómo se implementaría el mismo programa usando esta función.



```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #define MAX_STRING_SIZE ((size_t) 65536)
5  #define ARRAY_SIZE(array) ((sizeof((array)))/(sizeof((array)[0])))
6
7  char* integer_array_to_string(const int* array, size_t array_size)
8  {
9      char res[MAX_STRING_SIZE] = {};
10     int  printed_chars         = 0;
11
12     printed_chars += snprintf(res + printed_chars,
13                             MAX_STRING_SIZE,
14                             "[\n");
15     for (size_t ii = 0;
16         ii < array_size && printed_chars < MAX_STRING_SIZE - 2;
17         ++ii)
18     {
19         char* separator;
20         if (ii != array_size - 1) {
21             separator = ",\n";
22         } else {
23             separator = "\n";
24         }
25         printed_chars += snprintf(res + printed_chars,
26                                 MAX_STRING_SIZE - printed_chars,
27                                 "\t%d%s",
28                                 array[ii],
29                                 separator);
30     }
31     printed_chars += snprintf(res + printed_chars,
32                             MAX_STRING_SIZE - printed_chars,
33                             "]\n");
34     // desbordamiento
35     if (printed_chars >= MAX_STRING_SIZE) {
36         return NULL;
37     }
38     return strdup(res);
39 }
40
41 int main(void)
42 {
43     int list[24] = { };
44     char* serialization =
45         integer_array_to_string(list, ARRAY_SIZE(list));
46     if (serialization != NULL) {
47         printf("%s\n", serialization);
48     } else {
49         printf("Error: límite excedido\n");
50     }
51     free(serialization);
52 }

```

Programa 136: Ejemplo de uso de snprintf



Como puedes ver, el segundo argumento de la función es el límite de caracteres que podemos seguir imprimiendo. Siempre restamos a la longitud del buffer lo que ya llevamos impreso, así, si ya hemos impreso 100 letras, nos quedan $65536 - 100$. Por otro lado, ten en cuenta que `snprintf` devuelve siempre el número de caracteres que se imprimirían, es decir, sin contar con que se realice la impresión o no. A la hora de imprimir, el carácter nulo del final cuenta, es decir, si a una llamada a `snprintf` le pasas como límite un tres e intentas imprimir "abc", devolverá tres, pero no escribirá la última letra, porque siempre se escribe el carácter nulo (salvo que el límite que se le pase sea cero).

Ten cuidado, debes comprobar que el argumento del límite no es negativo, porque está definido como un tipo sin signo, por lo que si le pasaras un número negativo, sería un número positivo y aleatorio. En este código, lo comprobamos en el bucle.

13.2.1. Especificadores posicionales

Al principio del manual te enseñé a imprimir cosas para que pudieras probar tus programas, pero lo hice de una manera básica para no abrumarte al inicio de este manual. No obstante; aprovecho que hemos vuelto a utilizar funciones de manipulación de texto para desarrollar algo que me dejé entonces en el tintero. Estos son los especificadores posicionales, ya sabemos qué es un especificador, indican, dentro del formato, qué tipo dato tiene el argumento que queremos escribir en esa posición. El problema de este sistema, sencillo, es que provoca problemas cuando se repite la misma variable varias veces.

Si, por ejemplo, necesitamos imprimir varias veces la misma variable, no tiene ningún sentido que le pasemos varias veces a la función de impresión. Por ejemplo, imagínate una función que simulara una partida de nacimiento, incluyendo el nombre de los padres, es decir, para un padre llamado Fernando García Pérez y una madre llamada María Fernández López, si el nombre de pila del niño fuera Federico, su nombre sería Federico Garía López y deberían incluirse los tres. La función es trivial, vamos a implementarla simplemente con lo que sabemos:



```
1 #include <stdbool.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5
6 typedef struct person_s {
7     char *name;
8     char *last_name_1;
9     char *last_name_2;
10 } person_t;
11
12 char *son_name(const person_t *father, const person_t *mother,
13               const char *first_name) {
14     const int STRING_LENGTH = 65536;
15     char res[STRING_LENGTH];
16     char *format = "===PARTIDA DE NACIMIENTO===\n"
17                   " - Nombre del Padre: %s %s %s\n"
18                   " - Nombre de la madre: %s %s %s\n"
19                   " - Nombre del hijo: %s %s %s\n";
20     snprintf(res, STRING_LENGTH, format, father->name,
21              father->last_name_1, father->last_name_2, mother->name,
22              mother->last_name_1, mother->last_name_2, first_name,
23              father->last_name_1, mother->last_name_1);
24     return strdup(res);
25 }
26
27 int main(void) {
28     char* text = NULL;
29     person_t father = {"Fernando", "García", "Pérez"};
30     person_t mother = {"María", "Fernández", "López"};
31     text = son_name(&father, &mother, "Federico");
32     printf("%s\n", text);
33     free(text);
34 }
```

Programa 137: Ejemplo de impresión con argumento repetido

La función es sencilla, pero repetimos argumentos, como he introducido antes, eso no sólo no es eficiente, sino que puede inducir a errores, porque tienes que ir contando los argumentos. Cuando hay pocos, unos cinco o menos, es factible, si son muchos más y además hay repeticiones, es sencillo perderse, por ello, puedes indicar que se imprima el argumento de determinada posición. Esto nos permite pasárselos sólo una vez a la función. Veamos cómo.



```

1 typedef struct person_s {
2     char *name;
3     char *last_name_1;
4     char *last_name_2;
5 } person_t;
6
7 char *son_name(const person_t *father, const person_t *mother,
8               const char *first_name) {
9     const int STRING_LENGTH = 65536;
10    char res[STRING_LENGTH];
11    char *format = "===PARTIDA DE NACIMIENTO===\n"
12                  " - Nombre del Padre: %s %s %s\n"
13                  " - Nombre de la madre: %s %s %s\n"
14                  " - Nombre del hijo: %s %2$s %5$s\n";
15    snprintf(res, STRING_LENGTH, format, father->name,
16             father->last_name_1, father->last_name_2, mother->name,
17             mother->last_name_1, mother->last_name_2, first_name);
18    return strdup(res);
19 }
20
21 int main(void) {
22     char *text = NULL;
23     person_t father = {"Fernando", "García", "Pérez"};
24     person_t mother = {"María", "Fernández", "López"};
25     text = son_name(&father, &mother, "Federico");
26     printf("%s\n", text);
27     free(text);
28 }

```

Programa 138: Ejemplo de impresión con argumento repetido y especificador posicional

El primer cambio es que ahora los dos últimos especificadores son especiales, un especificador posicional empieza como todos, con un signo de porcentaje, después la posición en forma de número, un símbolo de dolar y el indicador del tipo, en este caso, una *ese*, de *string*. Como puedes ver, nos ahorramos la repetición de los argumentos.

13.3. Manejo de errores y manual

Ahora que hemos visto muchas funciones y sabemos cómo se compila un programa quiero volver a un ejemplo de programa muy sencillo de algunas secciones atrás, el programa 79: Ejemplo de programa que usa la función `remove`. Si incluimos el archivo de cabecera `errno.h` podemos reescribir el programa de este modo:



```
1 #include <errno.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(int argc, char const *argv[]) {
6     if (argc != 2) {
7         printf("Usage: ./main <path to the file>");
8     }
9
10    int error = remove(argv[1]);
11
12    if (error == 0) {
13        return EXIT_SUCCESS;
14    }
15    switch (errno) {
16        case ENOENT:
17            printf("No such file or directory.\n");
18            break;
19        case EACCES:
20            printf("Permission denied\n");
21            break;
22        default:
23            printf("Undetermined error\n");
24            break;
25    }
26    return EXIT_FAILURE;
27 }
```

Programa 139: Ejemplo de programa que usa la variable errno

Como has visto en el programa 79 compruebo el valor de retorno de la función que he llamado y, si es cero (lo que suele indicar éxito) devuelvo yo mismo `EXIT_SUCCESS`, terminando el programa. En cambio, si el valor devuelto no es cero, entramos en un `switch` sobre una variable que no conocemos. En él, utilizamos una serie de valores que tampoco están presentes en el programa. Esto es porque hemos incluido la cabecera `errno.h`. Esto nos permite utilizar la variable global `errno`.

Esta variable existe para que cuando llamemos a alguna función que la utilice para notificar errores, su valor será escrito en consecuencia del error a uno de los valores también definidos como macros en la cabecera. Por ejemplo, aquí hemos contemplado algunos de los casos. Sin embargo; se presenta la pregunta de cómo saber qué funciones utilizan esta variable global y cuáles no, y qué valores hay. Para esto se utiliza el **manual**. Esta es una función de los sistemas operativos Linux en la que puedes invocar el comando `man` para encontrar información sobre una función o cabecera. Por ejemplo, prueba a escribir en una terminal

```
$ man errno
```

Verás que sale un resultado que empieza por:



NAME

errno - number of last error

SYNOPSIS

```
#include <errno.h>
```

DESCRIPTION

The <errno.h> header file defines the integer variable `errno`, which is set by system calls and some library functions in the event of an error to indicate what went wrong.

`errno`

The value in `errno` is significant only when the return value of the call indicated an error (i.e., -1 from most system calls; -1 or NULL from most library functions); a function that succeeds is

allowed to change `errno`. The value of `errno` is never set to zero by any system call or library function.

For some system calls and library functions (e.g., `getpriority(2)`), -1 is a valid return on success. In such cases, a successful return can be distinguished from an error return by setting `errno`

to zero before the call, and then, if the call returns a status that indicates that an error may have occurred, checking to see if `errno` has a nonzero value.

<continúa>

En las pantalla de manual te mueves con las flechas del teclado y sales de ellas pulsando la tecla q. Si quieres saber el contenido de cualquier cabecera (funciones, variables...) o la *signature* de cualquier función, sólo debes escribir `man` seguido del nombre de la cabecera o función. Si consigues resultados que no son lo que buscabas, ejecuta `man 3 <nombre>` en su lugar. El manual tiene información sobre otras cosas (como comandos), pero la sección tres es la que habla de funciones de C.

13.4. Ejercicios de la sección

Ej. 18: Reescribe el ejercicio 15 prescindiendo del array estático de punteros a `char`. (Usa `realloc` y `strdup`).

Ej. 19: Escribe un programa que reciba un número indeterminado de palabras como argumentos y los ordene alfabéticamente y que, después, los imprima.

Ej. 20: Haz un programa que reciba como argumento una palabra y un número. Si el número es cero, debe convertir la palabra a minúscula, si el número es distinto de cero, debe convertirla a mayúscula. Por ejemplo:

```
$ ./main.exe Anthony 0
anthony
$ ./main.exe USA 0
usa
$ ./main.exe spqr 1
SPQR
```

Pista: Consulta la tabla ASCII para saber qué distancia hay entre una letra mayúscula y una minúscula.



Ej. 21: Crea un programa que dado un número como argumento imprima una pirámide como esta de tantos pisos como el número indicado:

```
$ ./main.exe 5
%%%%%%%%
 %%%
  %%
   %
    %
   %
  %
 %
%
```

Nota: utiliza la función `memset`.

Ej. 22: Escribe un programa que reciba una serie de puntos y de nombres para cada uno y después los imprima en orden de su distancia al origen de menor a mayor. Ejemplo de ejecución:

```
$ main.exe 2 3 Valencia 4 5 Cuenca -1 3 Vizcaya
-1 3 Vizcaya
2 3 Valencia
4 5 Cuenca
```

Nota: crea una estructura llamada `tagged_point` que maneje los *strings* como se ve en el programa 62, pero utilizando la función `strdup`.



14. Lógica avanzada

Hasta ahora nos hemos conformado con utilizar cualquier tipo entero (generalmente `int`) para almacenar valores lógicos, pero esto se puede evitar, ahorrando espacio y creando un tipo de dato que nos permita almacenar propiamente un tipo lógico. Este tipo es el tipo `bool`. Para poder usar este tipo debes incluir la cabecera `stdbool.h`. Además, este tipo añade dos nuevas palabras reservadas: `true` y `false`. Palabras que simbolizan, como puedes imaginar, un valor lógico cierto y uno falso.

Veamos un ejemplo de uso de este tipo en acción:

```
1 #include <stdio.h>
2 #include <stdbool.h>
3
4 int main(void)
5 {
6     bool it_is_difficult = 10;
7     printf("%d\n", it_is_difficult);
8 }
```

Programa 140: Ejemplo del uso del tipo `bool`

No existe un especificador para imprimir booleanos, pero si usas el especificador `%d`, verás que ese programa imprime «1». Es decir, si somos diligentes almacenando los valores lógicos en el tipo booleano, podemos asumir que un valor lógico cierto siempre será igual a 1. Sin embargo, seguimos teniendo un problema, que es la utilización de un byte entero para un valor que podría ser almacenado en un bit. Este problema viene del hecho de que los ordenadores no se direccionan porciones de memoria más pequeñas que bytes. Sin embargo; esto no quiere decir que no podamos manipular los bits concretos de un byte, por ejemplo, poner a cero un bit concreto, invertirlos... para eso utilizamos operadores a nivel de bit.

14.1. Operadores a nivel de bit

Voy a ser sincero, esta sección no sabía muy bien dónde encajarla, es decir, no encontraba el sitio donde incluirla en el manual. Al principio pensé incluirla después de los operadores lógicos, por la similitud gráfica que ahora verás, pero la utilidad de éstos era muy difícil de enseñar en ese momento del manual, o en ese momento de tu aprendizaje, mejor dicho. Así que mejor los vemos ahora. Hay varios operadores a nivel de bit, estos son:

1. Conjunción a nivel de bit, denotada en matemáticas por el punto medio (\cdot), se realiza con el operador `&`.
2. Disyunción a nivel de bit, denotada por $+$, se realiza con el operador `|`.
3. Negación a nivel de bit, denotado del mismo modo que la negación lógica, se realiza con el operador `~`.
4. Desplazamiento hacia la izquierda, operador binario que mueve todos los bits de un tipo entero a la izquierda tantas posiciones como indique el segundo operando. Se realiza con el operador `<<`.
5. Desplazamiento hacia la derecha, operador binario que mueve todos los bits de un tipo entero a la derecha tantas posiciones como indique el segundo operando. Se realiza con el operador `>>`.



Lo que quiere decir que lo hagan a nivel de bit es que, aplicando estos operadores a dos tipos enteros, el operador genera otro entero cuyo valor será el resultado de realizar la operación lógica indicada por el operador, pero bit a bit. Es decir, la conjunción a nivel de byte entre dos números es otro número en que cada uno de sus bits será el resultado de la conjunción de los bits en esa posición en los operandos.

Para que entiendas esto bien tienes que entender cómo están representados los enteros en el ordenador. Ya sabes cómo están representados los enteros sin signo, en binario natural, es decir, como lo que vimos en la sección 8.1. Sin embargo; los números negativos se representan de un modo especial. Hay varias maneras de representar los números con signo en binario, la más ingenua es dedicar un bit (el primero, generalmente) al signo y el resto al valor. De este modo, en un entero de ocho bits el número 7 sería 00000111 y el número -7 es 10000111, pero el problema de esta representación es que tenemos el número 0 y el -0. Y como somos informáticos, nos molesta sobremanera desperdiciar un preciado número. Por esto, los números con signo se representan en **complemento a dos**.

En este sistema de representación, los números positivos coinciden con su representación en binario natural, sin embargo, los números negativos se representan con su complemento, para hallar el complemento a dos de un número hay que seguir este proceso.

1. Invertir todos sus bits (el decir, pasar los ceros a unos y los unos a ceros).
2. Sumarle uno al resultado anterior.

Por ejemplo, volvamos al número 7 y a los ocho bits:

$$7_{(10)} = 00000111_{(2)}; \overline{00000111} = 11111000; + \frac{1}{11111001} \rightarrow -7_{(10)} = 11111001_{(CA2)}$$

La ventaja de esta representación es que, además, el primer bit sólo es uno cuando el número es negativo, así que es el primer bit sigue indicando el signo, como en la representación ingenua que te comenté al principio. Otra ventaja de esto es que sólo existe una representación del cero, que cuenta como número positivo, porque su primer bit es cero. Esto explica por qué cuando enunciamos los rangos de los tipos básicos, los tipos enteros siempre llegaban a números con valores absolutos una unidad mayores en el lado negativo que en el positivo, el `char`, por ejemplo, tiene un rango que va desde -128 a 127.

Volvamos a los operadores a nivel de bit, sea un `char` que valga -7 y otro que valga 12, veamos cómo se calcularía su conjunción a nivel de bit:

$$a = -7_{(10)} = 11111001_{(CA2)} \quad b = 12_{(10)} = 00001100_{(CA2)} \quad a \cdot b = \begin{array}{r} 11111001 \\ \cdot 00001100 \\ \hline 00001000 \end{array}$$

En cuanto a los desplazamientos, es sencillo, sea por ejemplo el propio número 7, es decir: 00000111, si le aplicamos un desplazamiento hacia la izquierda de dos bits, se convertiría en 00011100, en decimal: 28. Si no lo has notado ya, te lo digo yo, desplazar bits hacia la izquierda es una manera rápida de multiplicar por dos. Por otro lado, si desplazamos a la derecha, quedaría 00000001, los unos que ya no caben, se eliminan, como puedes ver. Análogamente a lo anterior, desplazar a la derecha es equivalente a dividir entre dos (en división entera, claro).

La mayor utilidad para esto es que nos permite establecer algo que los informáticos llamamos *flags*. Es decir, nos permite utilizar los bits de una variable entera para indicar sendas variables lógicas. Por ejemplo, podemos crear una función que serialice un objeto de tipo persona, con estas opciones:

1. Serializar con nombres legibles para personas, es decir, «apellido» en lugar de, por ejemplo «last_name_1».
2. Serializar con espacios o no entre caracteres de control, es decir, "last_name_1" : "Johnson" en vez de "last_name_1": "Johnson".



3. Serializar en varias líneas.

En general, si estas opciones **fuera**n excluyentes, no se pudieran dar dos juntas, podríamos codificarlas simplemente con un enumerado, pero como no lo son, deben ser variables lógicas separadas, el problema es que esto obligaría a que la función recibiera tres argumentos booleanos. Para esto utilizaremos las *flags*, vamos a asignar a cada opción un bit, y vamos a crear un tipo enumerado donde cada opción tenga el valor de un entero con ese bit puesto a uno. Para pasar varias opciones el usuario simplemente debe hacer una disyunción a nivel de bit con las opciones. Veamos la implementación de todo esto:



```
1 #include <stdbool.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5
6 typedef enum {
7     LEGIBLE_NAMES      = 1 << 0,
8     CONTROL_ESPACES    = 1 << 1,
9     MULTIPLE_LINES     = 1 << 2
10 } serialization_options_t;
11
12 typedef struct person_s {
13     char *name;
14     char *last_name;
15     unsigned int age;
16 } person_t;
17
18 char *serialize_person(const person_t *person,
19                       serialization_options_t opts) {
20     char *field_names_legible[] = {"Nombre", "Apellido", "Edad"};
21     char *field_names_normal[]  = {"name", "last_name", "age"};
22     char **field_names          = NULL;
23     char *separator             = NULL;
24     char *line_end              = NULL;
25
26     bool legible    = opts & LEGIBLE_NAMES;
27     bool espacios   = opts & CONTROL_ESPACES;
28     bool multiline  = opts & MULTIPLE_LINES;
29
30     field_names = legible ? field_names_legible : field_names_normal;
31     separator   = espacios ? " : " : ":";
32     line_end    = multiline ? "\n" : "";
33
34     char *fmt = "{%7$s\"%1$s\"%8$s\"%4$s\",%7$s"
35                "\"%2$s\"%8$s\"%5$s\",%7$s"
36                "\"%3$s\"%8$s%6$u%7$s}";
37
38     char res[65536];
39     sprintf(res, fmt, field_names[0], field_names[1], field_names[2],
40               person->name, person->last_name, person->age, line_end,
41               separator);
42     return strdup(res);
43 }
44
45 int main(void) {
46     person_t myself = {"Francisco", "Rodríguez", 26};
47     char *text = serialize_person(&myself, MULTIPLE_LINES |
48                                LEGIBLE_NAMES | CONTROL_ESPACES);
49     printf("%s\n", text);
50     free(text);
51 }
```

Programa 141: Implementación de opciones con operaciones a nivel de bit



El tipo enumerado de la línea 8 quizás asusta un poco, pero si recuerdas cuando los explicamos, a un enumerado le puedes explicitar el valor numérico de cada valor, aquí utilizamos el operador de desplazamiento a la izquierda para crear valores que sólo tengan un bit a uno, en posiciones distintas. Podría haberles asignado a mano los valores 1, 2 y 4, pero con el desplazamiento veo mejor y no me equivoco al calcular porque sé que siempre tiene que salir 1 << y después el número consecutivo.

Después está la definición de la estructura persona, que no tiene nada especial. En la línea 20 llegamos a la función de serialización, como es normal, recibe un puntero constante a la estructura que va a serializar y las opciones. Al principio declaramos variables, nada fuera de lo normal. Lo interesante empieza en la línea 28, estoy calculando variables booleanas para cada una de las opciones, es decir, estoy decodificando las *flags*. El modo es simple, al hacer la conjunción de las opciones con cada opción individual, el resultado será cierto si esa *flag* está levantada, falso en otro caso. Vamos a verlo rápidamente con la opción de la multilínea. Si `MULTIPLE_LINES = 1 << 2`, es decir 00000100, y las opciones son, por ejemplo, 00000101, al hacer la conjunción quedaría 00000100, es decir, un valor distinto de cero y por tanto cierto.

En la siguientes línea utilizo un artefacto del lenguaje que es útil cuando tienes muchas operaciones que se basan en valores lógicos. Se llama operador ternario, y su nombre radica en que es un operador con tres operandos, su sintaxis es como sigue: `condition ? valor1 : valor2`, el operador devolverá el `valor1` si la condición es cierta, si no, devolverá el `valor2`. Ten cuidado, porque ambos valores deben tener el mismo tipo. Aquí lo utilizo para definir los separadores de campo, de línea y los nombres de los campos en la serialización. El final de la función no tiene nada que no hayas visto antes, me aprovecho mucho de utilizar aquí especificadores posicionales.

Finalmente, en la función `main` llamamos a la de serialización con las opciones, simplemente las unimos todas con una disyunción a nivel de bit. Ten cuidado de no confundirte cuando utilices opciones y utilizar sin intención un operador lógico.

Por otro lado, hay una operación que me gustaría comentarte aparte, que es cuando tenemos la necesidad de bajar un *flag* concreto de un conjunto de opciones, por ejemplo, imagina que queremos serializar primero con todas las opciones y después sin la opción de multilínea, podríamos crear las opciones en ambas llamadas, pero si quisiéramos, podríamos guardar las opciones en una variable, usarlas y después bajarle esa bandera. Para hacer esto hay que hacer una conjunción con la negación de la bandera. Veamos cómo se haría:

```
1 int main(void) {
2     person_t myself = {"Francisco", "Rodríguez", 26};
3     char *text      = NULL;
4
5     serialization_options_t opts =
6         LEGIBLE_NAMES | CONTROL_ESPACES | MULTIPLE_LINES;
7
8     text = serialize_person(&myself, opts);
9     printf("%s\n", text);
10    free(text);
11
12    opts = opts & ~CONTROL_ESPACES;
13
14    text = serialize_person(&myself, opts);
15    printf("%s\n", text);
16    free(text);
17 }
```

Programa 142: Ejemplo de bajada de una bandera



Dicho así queda un poco contraintuitivo, veámoslos con este ejemplo, en la línea 12, `opts` vale 00000111, la opción `CONTROL_ESPACES` es 00000010, si la negamos, quedaría 11111101, si haces la conjunción bit a bit de ese valor, verás que todos quedarían como estuvieran en `opts`, salvo el correspondiente a `CONTROL_ESPACES`, que será forzosamente cero porque cualquier valor al que se le haga la conjunción con cero será cero. Así es como bajas una bandera en una serie de opciones. Ten en cuenta que para todas las operaciones a nivel de bit existen sus equivalentes de asignación, es decir: `|=`, `&=`, `~=`, `<<=` y `>>=`, así que podríamos escribir la línea en cuestión aquí de este modo:

```
opts &= ~CONTROL_ESPACES;
```



15. Algoritmos

Un algoritmo, como ya explicamos en la introducción, es el conjunto de pasos que debes seguir para conseguir un objetivo. Los programas que hemos hecho tienen una serie de objetivos, que cumplen mediante algoritmos. En esta sección quiero dar las primeras pinceladas sobre ellos, y, sobre todo, presentar algunos que te permitan interiorizar algunos patrones de código. Además, en secciones posteriores podremos utilizar estos simples algoritmos para introducir conceptos más complicados (y útiles) del lenguaje.

Lo primero que vamos a ver es cómo expresar un algoritmo, un algoritmo se a veces se expresa en lenguaje natural, es decir, como hablamos las personas, si ya tenemos el código que lo ejecuta, también estamos expresando el algoritmo, pero a veces es necesario utilizar herramientas intermedias, primero: porque puede ser difícil codificar el algoritmo directamente sin pensarlo antes y, segundo: porque hacer esto nos permite pensar en él sin tener que pensar en los artefactos concretos del lenguaje que vamos a utilizar, lo que nos permite dejar ese trabajo para más tarde.

Estas maneras intermedias son variopintas, por ejemplo, los diagramas del flujo que utilicé en su momento son una de ellas. Otra manera es el **pseudocódigo**, éste es un concepto que permite expresar los algoritmos de modo estructurado, utilizando artefactos básicos de cualquier lenguaje de programación: condicionales, bucles, llamadas a función... pero de una manera más laxa. Veámoslo con un ejemplo: el algoritmo que nos permite eliminar las repeticiones en un array expuesto en el programa 56: Ejemplo de reserva dinámica.

```
algoritmo eliminar_repeticiones :=  
  entrada: array  
  solución = {}  
para ii desde 0 hasta tamaño(array) - 1:  
    elemento = array[ii]  
    único = CIERTO  
    para jj desde 0 hasta ii - 1:  
      elemento2 = array[jj]  
      si elemento es igual a elemento2:  
        único = FALSO  
    si único  
      añadir elemento a solución  
retornar solución
```

Como puedes ver, se entiende mejor lo que hace, porque nos estamos librando de varios aspectos del lenguaje que no nos interesan, por ejemplo: asumimos que podemos saber el tamaño del array sin necesidad de preocuparnos de dónde viene; no tenemos que convertir las variables lógicas a números, podemos asumir que añadir un elemento a un array es autoexplicativo, podemos ignorar que hay reserva dinámica y simplemente decir que devolvemos el array. Visto así, es un poco inútil, pero piensa que escribir esto **antes** de la labor de codificación nos habría ayudado.

El problema del pseudocódigo es que podemos hacer «trampas», es decir, siempre podemos obviar varias cosas importantes que, a la hora de traducirlo a código real, no sean triviales. Por eso, puedes decidir tú hasta qué punto obvias o incluyes los artefactos del lenguaje. En este caso, por ejemplo, podríamos incluir el asunto de que las dimensiones de los arrays no pueden ser sabidos desde dentro de una función *per se*.



15.1. Recursividad

A la hora de definir algunos algoritmos, se definen utilizándolos a ellos mismos, es decir, parte de ellos incluye su propia utilización. Un ejemplo clásico de este tipo de algoritmos es el número factorial. Sea n un número entero, n factorial se denota como $n!$, que es igual a la multiplicación de todos los números desde 1 hasta n . Es decir:

$$n! = \prod_{i=1}^n i = 1 \cdot 2 \cdot 3 \cdot 4 \dots n$$

Siguiendo con el pseudocódigo, la manera de definir esto más evidente es la **iterativa**, es decir, la que utiliza repeticiones mediante bucles, las soluciones iterativas se contraponen frecuentemente a soluciones **recursivas**. Veamos el pseudocódigo de esta manera de resolver el problema:

```
algoritmo factorial :=  
  entrada: n  
  resultado = 1  
para i desde 1 hasta n:  
    resultado = resultado*i  
retornar resultado
```

Sin embargo; podemos sencillamente definir el factorial a partir de sí mismo, volvamos a su definición matemática:

$$n! = \prod_{i=1}^n i = n \cdot \prod_{i=1}^{n-1} i = n(n-1)!$$

Entonces, si seguimos con este razonamiento, podemos definir el pseudocódigo como:

```
algoritmo factorial :=  
  entrada: n  
  si n es igual a 0:  
    retornar 1  
  retornar factorial(n - 1) * n
```

Si saltamos a la implementación, veremos que es tan sencillo como que la función se llama a sí misma. Como ya bien sabes, una función puede llamar a otra, y así sucesivamente, pero nada dice que esa función no se llame a sí misma. El único problema que tiene esto es que, si se diseña mal, se puede incurrir en llamadas infinitas, esto provocaría que, como cada llamada sin finalizar que se acumula añade a datos a la pila, ésta terminara por acabarse y nuestro programa fallaría. Para ver la comparación en la implementación con ambas soluciones, crearemos dos versiones de la misma función, la iterativa y la recursiva, a la primera la llamaremos `factorial_iterative` y a la segunda `factorial_recursive`.



```
1 unsigned long factorial_recursive(int n)
2 {
3     if (n == 0) {
4         return 1;
5     }
6     return factorial_recursive(n - 1) * n;
7 }
8
9 unsigned long factorial_iterative(int n)
10 {
11     if (n == 0) {
12         return 1;
13     }
14     unsigned long res = 1;
15     for (int i = 1; i <= n; ++i) {
16         res *= i;
17     }
18     return res;
19 }
```

Programa 143: Comparación de algoritmos recursivos e iterativos

Este ejemplo es tan clásico porque se aprecian bien las diferencias de las dos formas de resolver el problema y, además, se aprecian bien las partes de una función recursiva. Una función recursiva, en términos generales, debe tener dos componentes: un caso base y la llamada (o llamadas) recursivas. El primero está aquí representado por el condicional, una función recursiva se basa en utilizar una definición circular, es decir, defines una función usándola a ella misma, para poder salir de ese ciclo, debe haber un momento en que demos la respuesta por sabida. Este caso base es cuando n es igual a cero, caso en que la respuesta es inmediata porque sabemos por definición matemática que es 1. La llamada recursiva es la siguiente línea.

Comparando las dos soluciones, la primera es lo que se llama más «elegante», en el sentido de que es una definición más legible (tanto así que podría decirse que es trivial) mientras que la iterativa es más larga y menos trivial. Sin embargo; es en general más eficiente la versión iterativa de un algoritmo que su versión recursiva. Esto es porque las sucesivas llamadas a función tienen cierto coste, obligan a hacer copias de datos, cada vez que invoques a la función, n será copiada en la pila, una copia encima de otra, hasta que llegues a la invocación del caso base. En ese momento, la llamada del caso base retornará y se empezará a desenrollar esa sucesión de llamadas a la función.

Por contraposición, la solución iterativa no tiene esa necesidad de copiar nada, ni de ir llamando a funciones una y otra vez, es un simple bucle. Es razonable pensar que la primera será más lenta. Déjame que lo compruebe y te dé los resultados para que puedas verlo. La solución recursiva tarda 1,7431 veces más que la solución iterativa, por lo que se puede apreciar que es más lenta. Además, recuerda la pila, utiliza más memoria. En resumen: los algoritmos iterativos son más rápidos que sus contrapartes recursivas, pero menos elegantes.

Otro ejemplo más claro de las diferencias entre ambos métodos se puede ver en el cálculo de la sucesión de Fibonacci. Someramente: esta sucesión fue creada por un matemático italiano en el siglo XIII, cuyo nombre da a esta sucesión. Es una sucesión que se define así: los dos primeros elementos son 1, y los demás se definen como la suma de los dos anteriores (ya se ve que aquí hay un caso base y una definición recurrente). Matemáticamente se definiría así:

$$a_1 = 1, a_2 = 1, \forall_{n=3}^{\infty} (a_n = a_{n-1} + a_{n-2})$$



Creo que entiendes que esto, iterativamente, se puede resolver con un bucle, como el cálculo del factorial. Así que saltamos a la definición recursiva:

```
1 unsigned long fibonacci(unsigned int n) {  
2     if (n < 3) {  
3         return 1;  
4     }  
5     return fibonacci(n - 1) + fibonacci(n - 2);  
6 }
```

Programa 144: Función para el cálculo de la sucesión de Fibonacci

Como puedes ver, seguimos el mismo patrón: un condicional que comprueba el caso base y una llamada recursiva. El caso base es lógico, el primer y segundo elemento son uno, así que se devuelve uno siempre que n sea menor que tres. Por otro lado, esto hace que también devolvamos uno cuando se introduce cero, pero esto es una simplificación que nos podemos permitir. Es la llamada recursiva la que nos interesa, como puedes ver, hay dos llamadas a esta función por cada llamada que no sea un caso base. Esto es importante, porque debes tener claro en qué orden se ejecutan estas llamadas, por esto, vamos a representarlo aquí.

- Llamamos a la función para el valor 5
 - Se llama a la función para el valor 4
 - Se llama a la función para el valor 3
 - ◊ Se llama a la función para el valor 2
 - ◊ Se devuelve 1.
 - ◊ Se llama a la función para el valor 1
 - ◊ Se devuelve 1.
 - Se devuelve 2
 - Se llama a la función para el valor 2
 - Se devuelve 1.
 - Se devuelve 3
 - Se llama a la función para el valor 3
 - Se llama a la función para el valor 2
 - Se devuelve 1.
 - Se llama a la función para el valor 1
 - Se devuelve 1.
 - Se devuelve 2
- Se devuelve 5

Estas listas anidadas expresarían cómo funciona la secuencia de llamadas a las funciones recursivas. Puedes observar varias cosas, la primera es que, incluso para una llamada pequeña, el quinto elemento, se realiza un gran número de llamadas a función. Por otro lado, si lo miras bien, se repiten un montón de cálculos, la llamada para tres se repite dos veces, la llamada a dos tres veces... y esto sólo para el valor de n igual a cinco. Esto te da la medida de que la implementación recursiva de este tipo de funciones es muy ineficiente. Técnicas más avanzadas permiten mitigar esto. En conclusión, es muy poco recomendable realizar esta implementación.



Y, si esto es así, si los algoritmos recursivos son así de ineficientes, lo lógico sería pensar que no se utilizan. Sí, se utilizan porque hay algoritmos que están definidos simplemente de un modo recursivo, es decir, que no es posible implementar iterativamente (o dicha implementación sería una simulación de la recursividad que aportaría poco), pero esos algoritmos son más complicados y serían problemáticos como primeros ejemplos del concepto de recursividad.

15.2. Algoritmos de ordenación

Uno de los conjuntos de algoritmos más elementales que existen en el conjunto básico de algoritmos de ordenación, entre ellos veremos los siguientes:

1. De burbuja.
2. Selección.
3. Inserción.
4. *Quick sort*.

Esto es así porque esta tarea (ordenar un vector o array) es muy común y permite una enorme cantidad de innovación, muchos autores de trabajos científicos se esfuerzan en proponer mejoras en estos algoritmos. Además, es una operación muy común y básica en la informática. Los primeros algoritmos son más bien sencillos, así que hablaremos directamente sobre la implementación en C, pero los dos últimos necesitarán de una explicación más pausada.

El algoritmo de la burbuja se basa en comparar cada elemento con el siguiente y, si no están en orden, intercambiarlos. Veamos cómo se haría:

```
1 void bubble_sort(int *list, int list_size) {  
2     for (int ii = 0; ii < list_size - 1; ++ii) {  
3         for (int jj = 0; jj < list_size - 1; ++jj) {  
4             if (list[jj] > list[jj + 1]) {  
5                 int aux = list[jj];  
6                 list[jj] = list[jj + 1];  
7                 list[jj + 1] = aux;  
8             }  
9         }  
10    }  
11 }
```

Programa 145: Implementación del algoritmo de la burbuja

Es el algoritmo de ordenación más simple, porque, como puedes ver, se compone de dos bucles anidados y un condicional. Rápidamente: recorres el array, pero como estás comparando con el siguiente elemento, paras una posición antes. Si el elemento anterior (`list[jj]`) es mayor que el que viene después, los cambias. Para cambiarlo simplemente guardas uno en una variable auxiliar, asignas el otro al primero y finalmente pones la variable auxiliar en el hueco libre. Esto, hecho una sola vez, provocaría que el elemento más grande subiera al último puesto del array. Y he usado el verbo subir porque es de este tránsito del que deriva el nombre del algoritmo, porque los elementos suben como burbujas de aire en el agua. De hecho, puedes probar a ejecutar la función cambiando la condición del primer bucle a `ii < 1`. Y verás como sólo el último elemento está siempre en orden.



Este algoritmo tiene una ventaja fundamental: utiliza muy poca memoria. Es más, si obviamos las variables auxiliares, los argumentos en la pila y demás, la única memoria ocupada es la variable auxiliar, es decir, el tamaño del tipo de dato que estés ordenando, en este caso: cuatro bytes. En informática casi siempre se hace un compromiso (en inglés: *trade-off*) entre memoria y velocidad. Los algoritmos que consumen más memoria para hacer lo mismo suelen ser más rápidos que los diseñados para ocupar poca memoria. El algoritmo de la burbuja es un ejemplo claro.

Como añadido, vamos a prestar atención a la realización del intercambio de elementos: como es algo que voy a usar en todos o casi todos los algoritmos, vamos a crear una función llamada `swap`, que nos permita intercambiar rápidamente dos elementos de tipo entero. Veamos cómo quedaría el algoritmo de la burbuja y cómo sería la definición de la función.

```
1 void swap(int *a, int *b)
2 {
3     int aux = *b;
4     *b = *a;
5     *a = aux;
6 }
7
8 void bubble_sort(int* list, int list_size)
9 {
10     for (int ii = 0; ii < list_size - 1; ++ii) {
11         for (int jj = 0; jj < list_size - 1; ++jj) {
12             if (list[jj] > list[jj + 1]) {
13                 swap(&list[jj], &list[jj + 1]);
14             }
15         }
16     }
17 }
```

Programa 146: Implementación de `swap` y uso en algoritmo de la burbuja

El siguiente algoritmo es el de selección. Este algoritmo se basa en elegir (seleccionar) el elemento menor y ponerlo en la última posición del array que tengamos ordenada. Al principio no hay ninguna, cuando hayas puesto una, deberás poner el siguiente elemento después de esa, y así sucesivamente, veámoslo:

```
1 void selection_sort(int *list, int list_size)
2 {
3     int ordered = 0;
4     while (ordered < list_size) {
5         int min_value = list[ordered], min_pos = ordered;
6         for (int jj = ordered; jj < list_size; ++jj) {
7             if (list[jj] < min_value) {
8                 min_value = list[jj];
9                 min_pos = jj;
10            }
11        }
12        swap(&list[ordered], &list[min_pos]);
13        ++ordered;
14    }
15 }
```

Programa 147: Algoritmo de selección



Si te das cuenta, este algoritmo ocupa marginalmente más memoria que el de la burbuja. En este caso debemos mantener siempre en memoria el valor y la posición del elemento más pequeño. En cambio, es un poco más rápido, ¿por qué? Porque en el peor de los casos, este algoritmo hace $\frac{n(n-1)}{2}$ comparaciones (siendo n el tamaño del array), las mismas que la burbuja y $n - 1$ intercambios. En el algoritmo de la burbuja haremos las mismas comparaciones y, sin embargo, haremos muchos más intercambios (tantos como comparaciones), porque los elementos van «pasito a pasito», no a su lugar directamente como en este caso. Consecuentemente el algoritmo de la burbuja ejecutaría, en el peor caso, tantos intercambios como comparaciones, sin embargo, este algoritmo sólo ejecutaría $n - 1$ intercambios.

El siguiente algoritmo es el de inserción. En el anterior elegíamos el elemento más pequeño y lo poníamos al principio, en este, el razonamiento es el contrario, crearemos una lista que definiremos como ordenada, inicialmente vacía, después, cogeremos un elemento de la lista original y lo insertaremos en la posición que le corresponda en la lista ordenada. De este modo, la lista ordenada siempre permanecerá así. Veamos un pseudocódigo más claro:

```
algoritmo insertion_sort :=
  entrada: array
  lista_ordenada = {}
para i desde 0 hasta size(array):
    insertar_ordenado(lista_ordenada, array[ii])
array = lista_ordenada
```

El problema de esto es que insertar en una lista ordenada es algo lo suficientemente complejo como para necesitar su propia definición. Vamos a definir un algoritmo que inserte un elemento en una lista, pero que **cuenta con que ya está reservada la memoria**.

```
algoritmo insertar_ordenado :=
  entrada: array, elemento
  insertado = 0
para i desde 0 hasta size(array):
  si array[ii] es mayor que elemento:
    insertar(array, elemento, ii)
    insertado = 1
    break
si no insertado:
  insertar(array, elemento, ii)
```

Pero seguimos teniendo el problema de cómo insertar, en general, un elemento en una posición dada, para esto utilizaremos otro algoritmo:

```
algoritmo insertar :=
  entrada: array, elemento, posición
  insertado = 0
para i desde size(array) hasta 0 en paso de -1:
  array[ii] = array[ii - 1]
array[posición] = elemento
```



Es un poco confuso, pero lo que hacemos es, primero, mover los elementos posteriores a la posición donde queremos insertar una posición a la derecha, abriendo así el hueco para el nuevo elemento, después, insertamos el elemento. Como puedes ver, ya en este algoritmo tenemos que hacer acopio de una serie de funciones intermedias. Aunque este algoritmo se suele implementar de un modo más sencillo, aquí lo vamos a hacer así para que empieces a ver la utilidad de extraer el comportamiento en funciones más pequeñas que se entiendan mejor en vez de escribir funciones muy largas. La implementación de las funciones auxiliares sería tal que:

```

1 void insert_at(int* list, int list_size, int position, int element)
2 {
3     for(int ii = list_size; ii != position; --ii){
4         list[ii] = list[ii-1];
5     }
6     list[position] = element;
7 }
8
9 void insert_at_ordered(int* list, int list_size, int element) {
10     int inserted = 0;
11     for(int ii = 0; ii < list_size; ++ii){
12         if(list[ii] > element){
13             insert_at(list, list_size, ii, element);
14             inserted = 1;
15             break;
16         }
17     }
18     if(!inserted){
19         insert_at(list, list_size, list_size, element);
20     }
21 }

```

Programa 148: Algoritmos auxiliares al de inserción

```

1 void insertion_sort(int* list, int list_size)
2 {
3     int ordered_list[list_size];
4     int ordered_size = 0;
5     for (int ii = 0; ii < list_size; ++ii) {
6         insert_at_ordered(ordered_list, ordered_size, list[ii]);
7         ordered_size++;
8     }
9     memcpy(list, ordered_list, sizeof(int) * list_size);
10 }

```

Programa 149: Algoritmo de inserción

Este algoritmo es el que usamos inconscientemente para ordenar objetos físicos cuando los tenemos. Si tuvieras los números {5,4,6,9,8}, probablemente cogerías el cinco y lo pondrías después del cuatro, verías que el seis está en orden, después verías el nueve y lo pondrías detrás del ocho. De todos modos, si has pensado que lo que tú harías sería selección, sí, sería posible que te resultara cómodo hacer eso también.

Pero la manera en que lo hemos implementado es muy ingenua, utilizamos muchos más pasos de los necesarios. Hay una manera más sencilla si nos damos cuenta de que podemos realizar todas las operaciones en la misma lista. Para ello voy a ponerte un ejemplo, imagínate la lista:

2	6	2	5	7
---	---	---	---	---



Las celdas en negrita son los elementos de la lista ordenada, lo que vamos a hacer es guardarnos el siguiente elemento después de ellos en una variable, es decir, el dos. Después, vamos a mover todos los elementos de la lista ordenada mayores que dos una posición a la derecha, quedando el array así:

2		6	5	7
----------	--	----------	---	---

. Con una posición vacía, como puedes ver. Ese hueco se utiliza para colocar el elemento que está en la variable que hemos mencionado antes, quedando el array así:

2	2	6	5	7
----------	----------	----------	---	---

. Si sigues a mano esta ejecución verás que nos guardaríamos el cinco, moveríamos el seis, pondríamos el cinco a la izquierda del seis; nos guardaríamos el siete volveríamos a mover el seis, pondríamos siete en el hueco libre y el array quedaría ordenado. La implementación de esto sería, por ejemplo, la siguiente:

```
1 void insertion_sort_optimized(int *array, int array_size)
2 {
3     int element;
4     for (int ii = 1; ii < array_size; ii++) {
5         element = array[ii];
6         int jj;
7         for (jj = ii - 1; jj >= 0; --jj) {
8             if (array[jj] <= element) {
9                 break;
10            }
11            array[jj + 1] = array[jj];
12        }
13        array[jj + 1] = element;
14    }
15 }
```

Programa 150: Implementación alternativa de ordenación por inserción

La lógica de esto quizá es menos evidente. El bucle exterior es muy sencillo, recorremos todo el array **desde la segunda posición**. Nos guardamos el elemento siguiente a los ordenados en la variable `element`. Después, en este bucle `for` interno moveremos todos los elementos de la lista ordenada a la derecha una posición, en orden inverso, por eso `jj` disminuye de uno de uno. Cuando hemos encontrado el elemento que es mayor, paramos (`break`). Finalmente, ponemos el elemento en su sitio.

Finalmente, llegamos al algoritmo *Quick Sort*, que, como puedes ver, su nombre no se basa en cómo lleva a cabo la tarea de ordenación sino por una cualidad muy interesante del mismo: es un algoritmo muy rápido. Si recuerdas lo que he dicho unos párrafos antes, en informática, velocidad y memoria ocupada son valores ortogonales (perpendiculares, cuando uno se propicia, es en detrimento del otro). Me interesa especialmente que lo veas porque **es un algoritmo recursivo**. Y es uno de esos algoritmos que es puramente así. Además, como es más complejo (como probablemente hayas podido deducir), vamos a ver primero su pseudocódigo.



```
algoritmo quick_sort :=
entrada: array
lista_pequeña = {}
lista_grande  = {}
pivote       = array[0]
si size(array) es igual a 1:
    res = {pivote}
    retornar res
para i desde 0 hasta size(array):
    si array[i] es menor que pivote:
        añadir(array[i], lista_pequeña)
    en otro caso:
        añadir(array[i], lista_grande)
lista_pequeña_ordenada = quick_sort(lista_pequeña)
lista_grande_ordenada  = quick_sort(lista_grande)
res = juntar(lista_pequeña_ordenada, pivote, lista_grande_ordenada)
retornar res
```

Este algoritmo se resume en lo siguiente: el caso base es cuando la lista es de tamaño uno, porque una lista de un elemento siempre está ordenada. Cuando no estamos en el caso base, creamos dos listas: en una irán los elementos menores que el pivote, y en otra los demás. El pivote es sencillamente un elemento cualquiera de la lista, a este nivel es irrelevante cuál, aunque su elección influye mucho en la eficiencia del algoritmo en casos especiales, es decir, con arrays concretos. Después, llamaremos a *Quick Sort* sobre ambas listas y la lista ordenada se compondrá de concatenar la lista ordenada de elementos más pequeños que el pivote, el pivote y la lista ordenada de elementos más grandes que el pivote. La implementación de este algoritmo es la siguiente:



```
1 int *quick_sort(int *list, int list_size)
2 {
3     int *biggers          = NULL, *smallers          = NULL,
4     *biggers_ordered      = NULL, *smallers_ordered = NULL,
5     *res                  = malloc(list_size * sizeof(int));
6
7     int bigger_size = 0, smaller_size = 0, pivot = 0;
8
9     if (list_size == 1) {
10         *res = list[0];
11         return res;
12     }
13
14     if(list_size == 0){
15         return NULL;
16     }
17
18     pivot = list[0];
19     biggers = malloc(sizeof(int) * list_size);
20     smallers = malloc(sizeof(int) * list_size);
21
22     for (int ii = 1; ii < list_size; ++ii) {
23         if (list[ii] < pivot) {
24             smallers[smaller_size] = list[ii];
25             smaller_size++;
26         }
27         else {
28             biggers[bigger_size] = list[ii];
29             bigger_size++;
30         }
31     }
32 }
33
34 biggers = realloc(biggers, sizeof(int) * bigger_size);
35 smallers = realloc(smallers, sizeof(int) * smaller_size);
36
37 biggers_ordered = quick_sort(biggers, bigger_size);
38 smallers_ordered = quick_sort(smallers, smaller_size);
39
40 memcpy(res, smallers_ordered, sizeof(int) * smaller_size);
41 res[smaller_size] = pivot;
42 memcpy(res + smaller_size + 1,
43        biggers_ordered, sizeof(int) * bigger_size);
44
45 free(biggers);
46 free(smallers);
47 free(smallers_ordered);
48 free(biggers_ordered);
49 return res;
50 }
```

Programa 151: Implementación de *Quick Sort*



El algoritmo es un poco más complicado así que lo que era en el pseudocódigo, pero debes seguirlo paralelamente a aquél. Lo primero es la declaración de todas las variables. De nuevo, vamos a empezar a declarar varias juntas, o si no las funciones de harían interminables. Nota que inicializamos sin comprobar nada la variable `res` (que es donde guardaremos el resultado) llamando a `malloc` para reservar un vector del mismo tamaño de la lista. No hay peligro porque llamar a `malloc` para reservar cero bytes es totalmente seguro. Después, como ya es normal en nuestras funciones recursivas, comprobamos si estamos o no en el caso base, el caso base es tanto una lista que mida uno como una lista vacía. Ambas listas están ordenadas por definición.

Después, empieza el caso recursivo, lo primero es leer el valor del pivote, lo hacemos ahora porque antes de comprobar el tamaño de la lista no sabemos si hay primer elemento. Ahora reservamos memoria para las listas de los elementos mayores y menores que el pivote respectivamente. Esto también lo hemos visto antes: como no sabemos cuánto pueden medir, utilizamos un umbral superior, en este caso es evidente que ninguna de estas listas puede medir más que el tamaño de la lista original. Después entramos en el bucle que cribará qué elementos van a cada lista. Como aquí estamos limitados por las restricciones de C, tenemos que hacerlo así: copiamos el elemento en cuestión a la posición siguiente al final de la lista (que es igual a su tamaño) y aumentamos en uno el tamaño. Nota cómo el bucle empieza en la posición uno, esto es así porque de no ser así estaríamos duplicando el pivote. Cuando hemos terminado el bucle, procedemos a redimensionar las listas de elementos mayores y menores al tamaño real que deben tener. Del mismo modo que con `malloc`, si se llama con un tamaño de cero no hay problemas, el puntero devuelto sigue siendo válido para pasarse a esta función.

Ahora que ya tenemos ambas listas, simplemente llamamos al mismo *Quick Sort* que nos otorgará las listas ordenadas. Del mismo modo que cuando te expliqué cómo utilizar `memcpy`, copiamos juntos la lista de elementos menores, el pivote y la lista de elementos mayores. Una vez hecho esto, sólo nos queda liberar la memoria que hemos utilizado para almacenar datos auxiliares. En este caso, las cuatro listas. Es evidente que con todas estas reservas de memoria este algoritmo consume mucho más que los anteriores, pero, además, recuerda lo que pasa con las funciones recursivas, esta memoria que hemos reservado se quedará bloqueada hasta que se terminen las llamadas recursivas, es decir, cada llamada tendrá sus propias copias de las listas. Esto explica, además, por qué *Quick Sort* es un algoritmo que no modifica la lista que se le pasa sino que **devuelve el resultado en otra**.

Ahora que ya hemos visto el algoritmo conceptual y la implementación de los algoritmos de ordenación, vamos a comparar su rendimiento en términos de tiempo de ejecución. Lo que voy a hacer es crear un vector de un tamaño grande, lo voy a ordenar con un algoritmo, volveré a introducir datos aleatorios en el vector, lo volveré a ordenar y así sucesivamente. Los resultados son, para 262.144 elementos:

Algoritmo	Tiempo (s)
Burbuja	243,0596
Selección	71,2807
Inserción	56,7917
Inserción (optimizado)	37,1698
<i>Quick Sort</i>	0,0656

Tabla 11: Tiempos de ejecución de los distintos algoritmos



Como puedes ver, en cada algoritmo se realiza una mejora bastante importante respecto al anterior, pero es muy llamativo que *Quick sort* destaque tanto. Como reflexión: yo no le pondría a un algoritmo que he hecho yo un nombre tan poco modesto si no fuera al menos mayormente cierto. Sin embargo; hay ciertos aspectos que hay que tener en cuenta en un algoritmo como estos más allá de que sean rápidos, por ejemplo, como ya dijimos, *Quick sort* consume mucha más memoria que todos los demás. Si utilizáramos este algoritmo en ciertas máquinas quizás nos viéramos con problemas de memoria. Además, hay una cualidad de los algoritmos llamada **estabilidad**, que representa cómo de constante para diferentes casos es el uso de tiempo, memoria, o ambas. En el caso de estos algoritmos, los únicos estables son el de burbuja y el de selección. Esto quiere decir que los resultados de los demás pueden variar mucho en ciertos casos, por lo que si trabajas en sistemas donde esos casos ocurren con cierta frecuencia o donde la estabilidad es más importante que la velocidad media, quizás debas usar uno de los algoritmos más lentos.

El ejemplo más flagrante de esto es lo que le pasa al último algoritmo con el vector más desfavorable. Para *Quick sort* el peor caso es, irónicamente, un array que ya esté ordenado (o que esté ordenado inversamente). Si vuelves a su descripción o a su implementación, verás fácilmente por qué, al elegir el primer elemento como pivote, si está ordenado, consistentemente todos los elementos caerán en la lista de elementos mayores, es decir, por cada nivel de recursividad sólo disminuirémos el vector en una posición. Eso implica que, en este caso, para un vector de mil posiciones, haremos mil llamadas recursivas que harán sendas copias del vector midiendo éstas mil, 999, 998, etc. Eso es insostenible a la mínima que el vector sea muy grande. Vamos a comparar el algoritmo de selección (que es estable), con *Quick Sort* en un vector ya ordenado.

El primer hecho que te llamará la atención es que no puedo utilizar la misma cantidad de elementos que en la comparación anterior, esto es porque mi ordenador no cuenta con memoria suficiente para que *Quick Sort* termine bajo estas condiciones. He utilizado un vector de 65.536 elementos. El algoritmo de selección tarda 4,58 segundos y *Quick Sort* tarda, ni más ni menos que 21,55. Aquí es donde entra en juego la estabilidad, selección es de media peor, pero nunca tarda mucho más. De hecho, como sí que puedo ejecutar el algoritmo de selección sobre 262.144, vamos a que cuánto tarda y compararlo con el resultado bajo el vector aleatorio.

Selección tarda 72,05 segundos, como puedes ver, casi exactamente lo mismo que con un vector aleatorio. Además, el consumo de memoria es siempre el mismo, cualidad que comparte con los algoritmos anteriores a él. Esto demuestra que hay que tener en cuenta más factores aparte de lo rápido que sea un algoritmo, pero todas estas cosas sería objeto de un libro en sí mismo y de un curso de algoritmia que no ha lugar, el motivo de esta sección es más bien enunciar la existencia de este tipo de problemas y mostrar la importancia de la elección e implementación de algoritmos para tareas incluso tan prosaicas como ordenar un array.

15.3. Búsqueda

La búsqueda es otra de las tareas más importantes de un informático o programador, pero la búsqueda depende de dónde estés buscando. Hasta ahora sólo conocemos la estructura del array o del vector, que en relación a la búsqueda son indistinguibles. La búsqueda es, dado un valor, encontrar en la estructura el valor dado, o la primera ocurrencia de él. Es evidente cómo resolver este problema en general, un bucle que compruebe si el elemento concreto del vector es igual al elemento que queremos buscar. Si esto es así, se devuelve el índice donde se encuentra, si no, se devuelve un valor que generalmente es -1 o cualquier negativo.

Pero hay una manera de buscar más rápidamente en un array o vector, si éste está ordenado, puedes buscar con la búsqueda binaria. Se llama binaria porque te mueves en dos direcciones: empiezas en el medio del array y compruebas si éste es menor que el que buscas, vas hacia delante en el array un cuarto de su longitud, repites la operación y vuelves a dividir entre dos la distancia que saltas. Veamos el pseudocódigo.



algoritmo búsqueda_binaria :=
entradas: array, objetivo

mientras array *no es igual* a {}:
 pos = centro(array)
 elemento = array[pos]
 si elemento *es igual* a objetivo:
 retornar pos
 si elemento *es menor que* objetivo:
 array = mitad_después(array, pos)
 si elemento *es mayor que* objetivo:
 array = mitad_antes(array, pos)

retornar -1

Básicamente, aprovechamos que el array o vector está ordenado para que, cuando elegimos una posición, sepamos que necesariamente el elemento que buscamos está en la mitad del array que quede después o antes de este elemento. Dividimos pues el tamaño del problema entre dos en cada iteración de este proceso hasta que queda sólo el elemento que buscamos (si sólo hay uno) o nada, y por tanto no existe el elemento. Veamos la implementación:

```
1 int binary_search(int* array, int array_size, int target)
2 {
3     int low_end = 0;
4     int high_end = array_size;
5     while (high_end != low_end) {
6         int pos = (high_end - low_end) / 2 + low_end;
7         int element = array[pos];
8         if (element == target) {
9             return pos;
10        }
11        else if (element < target) {
12            low_end = pos;
13        }
14        else if (element > target) {
15            high_end = pos;
16        }
17    }
18    return -1;
19 }
```

Programa 152: Implementación del algoritmo de búsqueda binaria

Como en C dividir arrays es complicado, vamos a realizar el juego con las posiciones. Definimos un umbral inferior y otro superior que serán los límites de la zona donde queremos buscar (la parte del array donde puede estar el elemento buscado). Lo que hacemos es siempre situarnos en el medio de esa zona del array y comprobar si hemos dado con el elemento que queremos, si no, vemos si es menor o mayor que donde estamos. Si el objetivo es mayor que donde estamos, movemos el límite inferior aquí, decir: sabemos que lo que buscamos no puede estar antes de la posición actual. Si el objetivo es menor, sabemos que debe estar antes. Si en algún momento los límites colisionan, es que nos hemos quedado sin zona que buscar y devolveremos -1.



La ventaja que tiene este algoritmo reside en un comentario que he dejado caer unos pocos párrafos atrás, cada vez que comprobamos si un elemento es el que buscamos, dividimos el problema entre dos. Esto quiere decir que, de media, haremos $\log_2(n)$ comprobaciones. En el caso de la búsqueda «normal», en cualquier array, de media haremos $\frac{n}{2}$ comprobaciones. Esto hace que sea un algoritmo más deseable, aunque hay que tener en cuenta que el array debe estar ordenado, y si no lo estaba, esto tiene un coste. Lo ideal es que se utilice con arrays que se mantengan siempre ordenados, es decir, donde las inserciones se hagan ordenadas, pero eso tiene un coste a su vez porque hay que mover los elementos en la operación de inserción (de una manera similar a como hicimos en el algoritmo de ordenación por selección).



16. Algoritmos genéricos

Ahora que ya sabemos qué es un algoritmo en sus términos más elementales y que conocemos algunos con una complicación suficiente como para retornos, podemos dar un paso más: hacer que estos algoritmos sean más «puros», sean simplemente un conjunto de instrucciones, pero sean ignorantes de qué tipo de dato van a recibir o de qué operación van a realizar. Dicho así, esto es confuso, pero lo vas a entender fácilmente. En la sección anterior hemos implementado varios algoritmos que manejan arrays, y, aunque no lo he dicho, si retrocedes ahora y miras el código de ejemplo, siempre verás que son arrays o vectores de enteros, nunca de otro tipo de dato. El problema es que, como supondrás, el proceso de ordenación de un array de enteros es similar al de un array de cualquier tipo de dato que admita ordenación. Por ejemplo, de decimales.

Pensarás que, entonces, siempre podemos copiar el código de la ordenación de elementos de tipo `int` y sustituir `int` por `double`. Podríamos, pero esta manera de proceder tiene un defecto muy problemático: duplica código. Duplicar código es malo por dos motivos: hace nuestros ejecutables más grandes, y nuestro código en general **menos mantenible**. Dicho en términos entendibles, es más difícil que le pases ese código a alguien y lo entienda rápido y, en caso de haber un error o comportamiento no definido, lo resuelva rápido. Imagínate que encontramos un error en la implementación de la función de ordenación, sea la que sea. Tendríamos que rastrear a mano todas las copias para eliminar ese error. Además, somos programadores: nuestro objetivo es hacer más con menos.

Para esto tenemos dos herramientas genéricas, la primera es el puntero a `void`. Cuando te presenté el puntero a `NULL` y la reserva de memoria dinámica te dije que `malloc` devuelve un puntero a `void` que luego se convertirá a cualquier tipo que necesites. Es decir, el puntero a `void` se convierte implícitamente en el que tú quieras, con la asignación. Además, después, `realloc` o `free` reciben punteros a `void` que no necesitas convertir para pasárselo. En resumen: en C el puntero a `void` tiene conversión implícita a todos los tipos, y todos los tipos a `void`.

Esto tiene una potencia enorme, porque podemos recibir un puntero a `void` cuando no sepamos lo que vamos a recibir. El problema es que un puntero de este tipo **no se puede desreferenciar**. Como ya te dije en su momento, no se puede interpretar un puntero a `void`. Sólo se puede interpretar si asumimos algún tipo y asignamos este puntero sin tipo a uno con él. Por ejemplo, veamos una función que invierta un array **de cualquier tipo**.

```
1 void invert_array(void* array, int array_size, int type_size)
2 {
3     void *var = malloc(type_size);
4     for (int ii = 0; ii < array_size / 2; ++ii) {
5         void* element = array + (ii * type_size);
6         void* opposite = array + (array_size - 1 - ii) * type_size;
7         memcpy(var, element, type_size);
8         memcpy(element, opposite, type_size);
9         memcpy(opposite, var, type_size);
10    }
11    free(var);
12 }
```

Programa 153: Inversión de arrays de cualquier tipo



Como puedes ver, se parece mucho a como lo harías sabiendo su tipo, pero hay que recibir como argumento el tamaño del tipo de dato. Lo que hacemos en el bucle es crear dos punteros que representan los dos elementos que tenemos que intercambiar, no podemos hacerlo con variables porque no sabemos el tipo. Estos punteros no son necesarios, pero, siendo sinceros, sin ellos las líneas quedan totalmente ilegibles. Después, usamos la función `memcpy` para poder mover el elemento en la posición `ii` a una variable auxiliar, el elemento en la posición opuesta (`ii - 1 - array_size`) a la `i`-ésima y después desde la variable al elemento opuesto.

Lo primero que se nota es algo evidente: una función sencilla enseguida se convierte en algo más complicado de leer, porque no podemos usar operadores, sino que tenemos que usar llamadas a función. Pero debes admitir que, ahora, podemos invertir todos los tipos de arrays sin duplicar nada de código. También hay una cuestión importante: mientras que el operador de asignación y saber el tipo provoca que el compilador pueda mapear las operaciones que escribas con operaciones del procesador para mover bytes en paquetes de cuatro u ocho, por ejemplo, haciéndolo así, `memcpy` se ve obligado a copiar byte a byte. Esto hace que los algoritmos genéricos creados así sean más lentos que sus contrapartes específicas.

Hay muchos ejemplos de algoritmos que bien pueden utilizar esta técnica, pero sólo con esto no podemos terminar de solucionar el problema. Vamos a crear una función que imprima un array de elementos genéricos. La función, con las herramientas que tenemos, quedaría así:



```

1 void print_array_generic(void*      array,
2                          size_t     array_size,
3                          const char* separator,
4                          type_t     type)
5 {
6     char* specifier[] = { "%d%s", "%f%s", "%lf%s", "%c%s" };
7
8     for (size_t ii = 0; ii < array_size; ++ii) {
9         switch (type) {
10            case int_enumerate:
11                {
12                    int* var = array + (ii * sizeof(int));
13                    printf(specifier[type], *var, separator);
14                }
15                break;
16            case float_enumerate:
17                {
18                    float* var = array + (ii * sizeof(float));
19                    printf(specifier[type], *var, separator);
20                }
21                break;
22            case double_enumerate:
23                {
24                    double* var = array + (ii * sizeof(double));
25                    printf(specifier[type], *var, separator);
26                }
27                break;
28            case char_enumerate:
29                {
30                    char* var = array + (ii * sizeof(char));
31                    printf(specifier[type], *var, separator);
32                }
33                break;
34            }
35        }
36    }

```

Programa 154: Imprimir arrays de varios tipos

Aquí necesito saber el tipo, no sólo su tamaño, por una razón: el especificador de impresión necesita el tipo exacto. Lo que hago es declarar un enum para poder indexar los tipos en un array de especificadores. Además, en el switch utilizo un truco: cada caso está en un bloque de código diferente (recuerda que se pueden crear bloques de código sin que estén asociados a una estructura), así en cada uno la variable `var` puede ser del tipo que queramos. Esto es un paso hacia delante, pero sigue siendo una función bastante mala. Es como si hubiéramos pegado todas las funciones para cada tipo y las hubiéramos metido en la misma. Es un avance, sí, porque, aunque mantenemos los mismos fragmentos de código duplicado, al menos ahora están todos juntos.

Pero lo ideal es que pudiéramos recibir desde fuera la manera de imprimir, sería deseable que el usuario de la función nos diera una función que a su vez contuviera el comportamiento de impresión. Es decir, de algún modo debemos poder convertir un comportamiento, un algoritmo, una función, al final del día, en algo que se pueda pasar, mover, trasladar de un sitio a otro. En C hay un mecanismo concreto para hacer esto que nos otorga un poder enorme: los punteros a función.



Un puntero a función es un puntero (es decir, una dirección de memoria) que apunta a las instrucciones que se ejecutarán en esa función. Estos punteros nos permiten, como formulé antes, transferir, comunicar, a funciones de nuestro programa comportamientos específicos. Cada tipo de función que se puede declarar es un tipo de puntero distinto. Una función, como vimos en su momento, se define por su tipo de retorno y por el tipo de los argumentos que recibe. Esto quiere decir que, por ejemplo, estas dos funciones son iguales a estos efectos.

```
1 int sum(int a, int b);  
2 int multiply(int x, int y);
```

Si cada función es un tipo, quizás estés pensando, debe tener un nombre por el que referenciarlo y una manera de declararlo y de usarse. Sí, pero no, las funciones no se pueden inicializar como variables, si quisieras guardar el puntero de una función, podrías asignarlo a un puntero a `void`. Sin embargo; se recibe como argumento. Para que una función reciba un puntero a función como argumento se usa este esquema:

```
1 tipo_de_retorno (nombre) (tipo1, tipo2...)  
2 // por ejemplo  
3 int(foo)(int, int)
```

Con un ejemplo todo se ve mejor, veamos algo sencillo: cómo hacer una función que reciba otra función y la ejecute. Por ejemplo, vamos a hacer una función que reciba otra con esta signatura: `void (void)` y la ejecute diez veces.

```
1 void execute_10_times(void (foo)()) {  
2     for (int ii = 0; ii < 10; ++ii) {  
3         foo();  
4     }  
5 }
```

Programa 155: Ejemplo primero de puntero a función como argumento

Como puedes ver, lo único distinto es la declaración de la función, que ya hemos tratado. La llamada a la función `foo` se hace como cualquier otra. Nos queda la otra cara de esta moneda, cómo se llama a la función `execute_10_times`. Esto es bien sencillo, porque el puntero de una función es, simplemente, su nombre sin los paréntesis, así que la llamada quedaría como:

```
1 execute_10_times(print_a);
```

Programa 156: Llamada a una función que recibe un puntero a función

Hecho esto, volvamos a la función de impresión genérica, pero esta vez haremos que, a su vez, reciba una función que ejecute la impresión de un único elemento. Tenemos que elegir la signatura de esta función, como es una función que imprime, lo normal es que no devuelva nada y que reciba sólo el elemento que queramos imprimir. Si recibiera el elemento en sí mismo, volveríamos al problema de que hay que definir su tipo. Lo que haremos es una función que reciba un puntero a `void` y no devuelva nada. Ojo, esa función será hecha por quien use nuestra función de impresión de arrays, no por nosotros, salvo que seamos el mismo individuo, claro. Para el ejemplo, enseñaré ambas funciones.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void print_int(void* num)
5 {
6     printf("%d", *((int*)num));
7 }
8
9
10 void print_array_generic(void*      array,
11                          size_t     array_size,
12                          size_t     type_size,
13                          const char* separator,
14                          const char* end,
15                          void(print_foo)(void*))
16 {
17     for (size_t ii = 0; ii < array_size; ++ii) {
18         void* element = array + (ii * type_size);
19         print_foo(element);
20         if (ii != array_size - 1) {
21             printf("%s", separator);
22         }
23     }
24     printf("%s", end);
25 }
26
27 int main(int argc, char** argv)
28 {
29     int array[] = {1,2,3,4,5,6,7,8,9,0};
30     print_array_generic(array,
31                         ARRAY_SIZE(array),
32                         sizeof(*array),
33                         " ",
34                         "\n",
35                         print_int);
36 }
37 }
```

Programa 157: Definición de función de impresión genérica

La función genérica de impresión es sencilla, es simplemente un bucle que recorre el array y le pasa el puntero correspondiente a la función de impresión que se le pasa como argumento. Como es lógico, necesito el puntero al array, la longitud del mismo y, al ser un puntero a `void`, necesito el tamaño del tipo. Como puedes ver, para darle un poco de vidilla he hecho que la función reciba dos cadenas: una como separador, que imprimiré después de todos los elementos (menos el último, de ahí el condicional) y un terminador, que se imprimirá después del array. La que podríamos denominar función de impresión específica, es decir: `print_int`, es una función terriblemente simple, sólo llama a `printf` haciendo un cásting a puntero a entero y desreferenciándolo.



Es cierto que este modelo de función nos lleva al mismo problema, si queremos imprimir tipos distintos, tendremos que definir funciones distintas. Sí, pero piensa una cosa: hemos hecho que el código que se duplique sea ínfimo, porque son funciones triviales de una línea, además, una vez salimos de los tipos básicos que podríamos, contando sus variantes, agrupar en 10 funciones, aproximadamente, se acaba la duplicidad de código. Esto es así porque cualquier otra estructura requeriría una función o bien de impresión o bien de conversión a cadena de texto hecha a medida para ella.

Como habrás podido notar, la sintaxis para declarar que una función recibe otra como argumento es complicada y, además, rompe el patrón de una lista de argumentos que, hasta ahora, era siempre una sucesión de tipos y nombres separados por comas. Con esta sintaxis para punteros a función se incluyen varios paréntesis. C permite definir un tipo para los punteros a función. Es decir, aún descubrimos otra faceta de la poderosa palabra `typedef`. Veamos cómo se haría y pondré comentarios de algunos ejemplos de funciones que pertenecerían a ese tipo.

```
1 typedef void(print_fun_t)(void*); //ej: void print_int(void* a);
2 typedef void(*malloc_t)(void); //ej: void* malloc(void);
3 typedef int(sum_t)(int, int); // ej: int sum(int a, int b);
```

Programa 158: Definición de tipos puntero a función

Presta atención porque, si la función devuelve un puntero, el asterisco va dentro de los paréntesis, junto al nombre del tipo, no fuera. Si hiciéramos en el programa la primera definición, podríamos cambiar nuestra función genérica por:

```
1 void print_array_generic(void*      array,
2                          size_t     array_size,
3                          size_t     type_size,
4                          const char* separator,
5                          const char* end,
6                          print_fun_t print_foo);
```

Programa 159: Ejemplo final de función que recibe un puntero

Queda mucho más claro, porque el último argumento se identifica, como cualquier otro, por un tipo y un nombre.

La utilidad de esto se puede ver muy bien en funciones que ya hemos tratado, las funciones de ordenación. Ahora mismo esas funciones siempre ordenan vectores de enteros y, además, siempre de menor a mayor. Esto presenta varias posibles mejoras, la primera es evidente, tenemos que poner punteros a `void` y utilizarlos, pero el otro es más interesante. Esta segunda mejora es: sólo podemos utilizar una relación de orden. Es decir, sólo podemos ordenar números y de menor a mayor, no podemos comparar estructuras, no podemos comparar cadenas de texto alfabéticamente, pero podríamos si utilizáramos estas nuevas herramientas. Para generalizar una función de ordenación necesitaríamos el tamaño del tipo que vamos a ordenar y una función de comparación.

Las funciones de comparación son un tipo muy concreto, se llaman predicados, y son funciones que devuelven un valor lógico ante un conjunto de argumentos. Un predicado básicamente evalúa una proposición (recuerda la sección sobre lógica), pero sobre sus argumentos. En nuestro caso, un predicado para ordenar sería: función que compruebe que el primer elemento es menor que el segundo. Así que el prototipo de la función que tenemos que recibir sería una que devolviera un valor lógico y recibiera dos punteros a `void`. De nuevo, recibe dos punteros a `void` para ser compatible con nuestra función genérica, aunque el predicado sí debe saber qué tipo está comparando, lógicamente.



Vamos a usar el algoritmo más sencillo de ordenación que tenemos, el de la burbuja, para ilustrar esto, esto es así porque una implementación genérica de, por ejemplo, *Quick Sort* sería más compleja y larga, y me interesa que veas el concepto del puntero a función y de punteros a void trabajando, no que te pierdas en una función de 40 líneas. Además, para verlo mejor, vamos a utilizar un caso concreto: una función que ordene cadenas de texto alfabéticamente, usando `strcmp`.

```
1 typedef int(comparator_t)(const void*, const void*);
2
3 void generic_swap(void* one, void* other, size_t type_size)
4 {
5     char aux[type_size];
6     memcpy(aux, one, type_size);
7     memcpy(one, other, type_size);
8     memcpy(other, aux, type_size);
9 }
10
11
12 void generic_bubble_sort(void*      array,
13                          size_t     array_size,
14                          size_t     type_size,
15                          comparator_t comparator)
16 {
17     for (int ii = 0; ii < array_size - 1; ++ii) {
18         for (int jj = 0; jj < array_size - 1; ++jj) {
19             void* element = array + (jj * type_size);
20             void* next_element = array + ((jj + 1) * type_size);
21             if (!comparator(element, next_element)) {
22                 generic_swap(element, next_element, type_size);
23             }
24         }
25     }
26 }
```

Programa 160: Definición de bubble_sort genérico

Como puedes ver, definimos el tipo de nuestra función de comparación, la cual devolverá un entero y recibirá dos punteros constantes a void, y esto es importante, la definición de un tipo puntero a función no tiene conversiones implícitas de ningún tipo. Esto es: como hemos definido la función tal que recibirá dos punteros constantes a void, una función que reciba punteros no constantes no será de este tipo y no se podrá usar como tal, ten esto en cuenta.

Después tenemos la función de intercambio en su versión genérica, es decir, en vez de usar el operador de asignación, utilizaremos la función de copia de memoria con el tamaño del tipo. Y después, la función genérica de ordenación. Como puedes ver, simplemente hemos sustituido el condicional por la negación de la llamada a la función. Recuerda cómo funcionaba el algoritmo de la burbuja: cuando el elemento *i*-ésimo es **mayor** que el siguiente, se intercambian. Es decir: cuando **no** se cumple el predicado de que elemento *i*-ésimo sea menor que el siguiente. La ventaja de que esto sea un predicado es que ahora podemos comprobar si es mayor, en vez de menor, para ordenar al revés el vector, o podemos usar, como hemos dicho, funciones de comparación para estructuras concretas.

Dentro del bucle debemos calcular primero los punteros de los elementos. Esto es así por legibilidad, pero podríamos escribir las expresiones en la propia función de intercambio. Ten en cuenta que debemos multiplicar, de nuevo, *ii* por el tamaño del dato. Recuerda: son punteros a void, no entra en juego la aritmética de punteros, son direcciones de memoria absolutas. Una vez calculados simplemente llamamos a la función de intercambio.



Debemos tener en cuenta también la función de comparación. En el caso de un *string* es interesante porque uno puede confundirse debido a que los punteros se suman sobre punteros. Veamos cómo es la función de comparación:

```
1 int compare_strings(const void* one, const void* two) {
2     char* const* str1 = one, * const* str2 = two;
3     return strcmp(*str1, *str2) < 0;
4 }
```

Programa 161: Función auxiliar de comparación de *strings*

Es muy interesante porque puedes ver la primera línea, que introduce algo que no habíamos visto. Esta función recibe dos punteros constantes de `void`. Estos punteros son, en realidad, punteros a punteros a `char`, es decir: `char**`. Pero como los hemos recibido como constantes, no podemos hacerles *cástring* a ese tipo, el compilador nos diría, hablando claro: «estás haciendo *cástring* de un puntero constante a uno que no lo es, podrías modificar el contenido». Pero si pusiéramos el modificador `const` primero de todo como hemos hecho siempre el compilador seguiría lanzándonos esa advertencia. La clave es que lo que es constante es lo que, por ejemplo, `one` apunte, es decir, la constancia está pegada al contenido de `one` y `two`. Si escribiéramos `const char**` seguiríamos pudiendo modificar el contenido al que apunta tal dirección. Vamos a verlo con un dibujo:

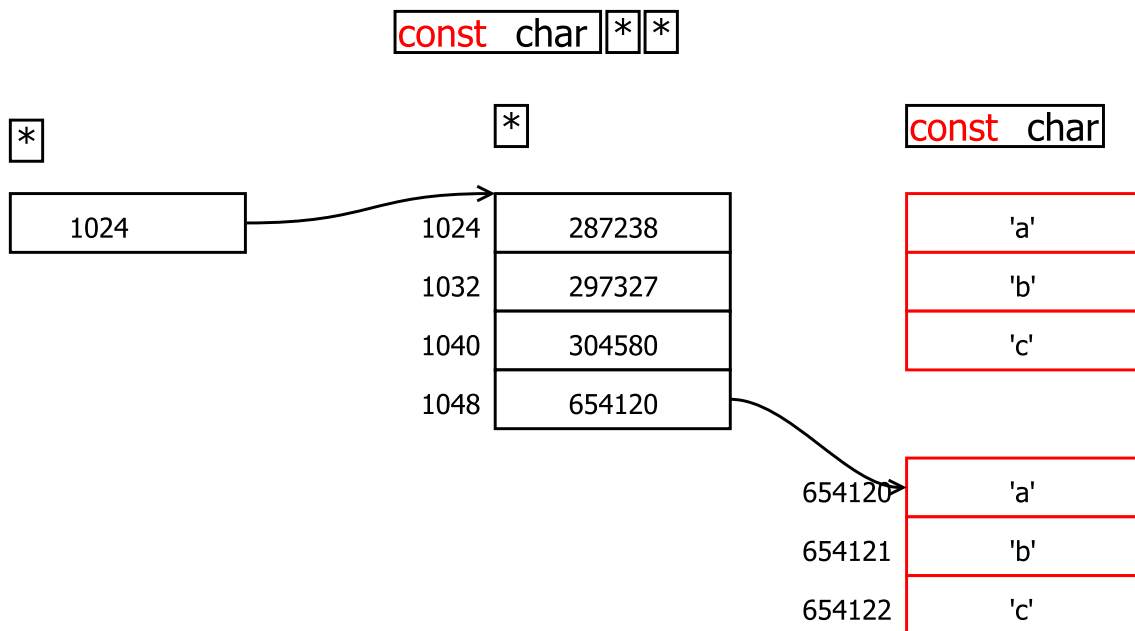


Figura 11: Puntero a puntero a char constante

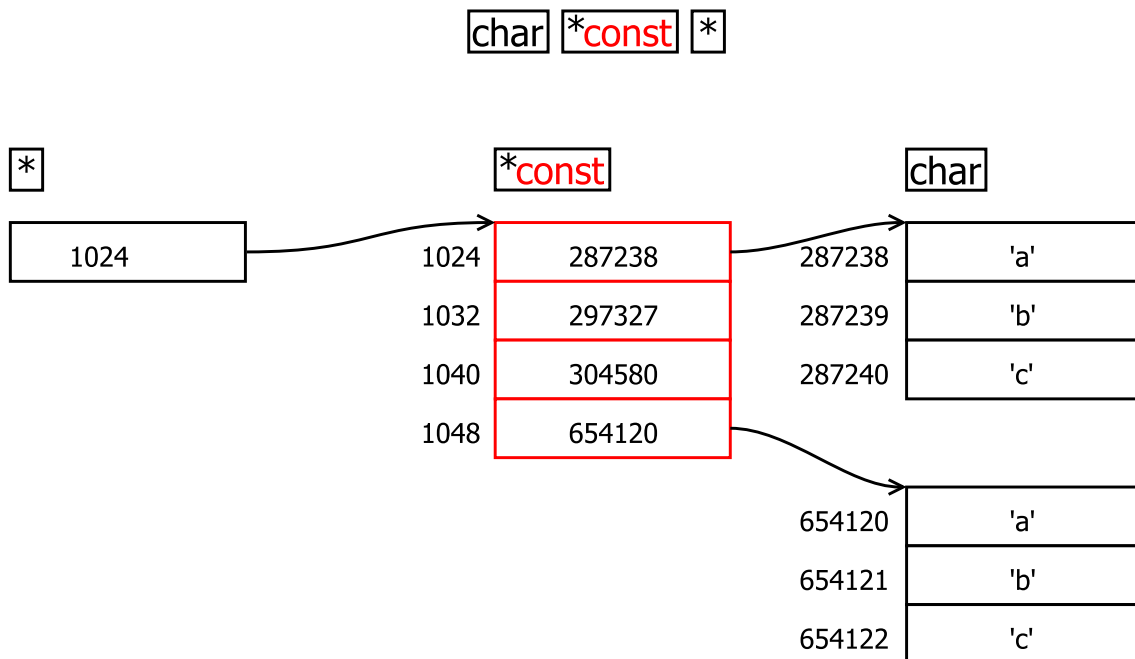


Figura 12: Puntero a puntero constante a char

Vamos a explicarlo despacio, si miras el primer dibujo, es «a lo que estamos acostumbrados», cada asterisco es un nuevo nivel de puntero, así que puedes leer la declaración desde la izquierda y construir los tipos. Empecemos: nos encontramos `const`, lo que venga ahora es constante, después `char`, ahora llega un asterisco, el asterisco inicia un nuevo nivel, así que este puntero **no** será constante, porque no tiene un `const` a la derecha y, finalmente, otro nivel de puntero, que tampoco será constante. Ahora que ya tienes los tres grupos, los inviertes, es decir: puntero no constante, a puntero no constante, a char constante.

En el caso siguiente tenemos un `char`, después un asterisco, es decir, un nivel de puntero, lleva `const` a la derecha, así que es constante, y después otro puntero, sin constancia. Es decir, invirtiéndolo: puntero no constante a puntero constante a char no constante. En ambos casos, en el diagrama, he señalado en rojo los valores que no puedes cambiar, como puedes ver, en el superior no podemos cambiar los *strings*, pero sí los punteros intermedios. En el caso de abajo, por el contrario, podemos modificar los caracteres, pero no los punteros del array intermedio.

Una de las implicaciones de las funciones genéricas es la siguiente: se introduce una sobrecarga inevitable, por dos motivos. El primero es que las funciones que utilizan punteros a `void` tienen que hacer cálculos explícitos que se harían implícitamente. No voy a entrar en detalles de arquitectura de computadores, pero los ordenadores tienen en sus procesadores instrucciones que manejan datos como enteros de cuatro bytes y números decimales (y algunos más). Al tener que copiar byte a byte, impedimos que se utilicen y, además, tenemos que darle más vueltas al bucle de copia, lo cual es más costoso. El segundo es que cuando se llama a una función de manera normal el compilador cuenta con ello para saber cómo generar el binario. Cuando ésta es un argumento, esta tarea se le hace más complicada, porque no sabe qué función es hasta el momento de la ejecución. Para hacer esto patente, vamos a hacer una comparación con el tiempo que tardan ambas versiones en ordenar 65.536 y 131.072 elementos. Vamos a comparar ambas cargas de trabajo porque quiero que veas una cosa.



Función	N=35.536	N=131.072
Específica	16,21	64,61
Genérica	41,20	164,44
Ratio	0,39	0,39

Tabla 12: Tiempos de ejecución de los distintos algoritmos

Como puedes ver, la versión genérica tarda más, pero he calculado un dato importante a ese respecto: el ratio entre el tiempo del algoritmo específico y el algoritmo genérico. Como puedes ver, aunque el algoritmo genérico es peor que el específico, la buena noticia es que esa diferencia es constante, es decir: no empeora con el tamaño del vector. Esto hace que, si podemos asumir el aumento de tiempo, la solución sea escalable, que es una manera que se tiene en informática de decir que puedes hacer crecer algo sin quedarte sin recursos rápidamente.

Un ejercicio muy interesante sería que programaras la versión genérica de *Quick Sort* y que, además, hicieras estas mismas mediciones. Para medir el tiempo puedes utilizar este código:

```
1 #include <stdio.h>
2 #include <time.h>
3
4 double timespec_to_double(const struct timespec* tm)
5 {
6     return tm->tv_sec + tm->tv_nsec / 1000000000.0;
7 }
8
9 int main(int argc, char** argv)
10 {
11     double start, stop;
12     struct timespec start_ts, stop_ts;
13
14     clock_gettime(CLOCK_REALTIME, &start_ts);
15     start = timespec_to_double(&start_ts);
16     // Aquí el código que quieres medir.
17     clock_gettime(CLOCK_REALTIME, &stop_ts);
18     stop = timespec_to_double(&stop_ts);
19     printf("Hemos tardado: %lf\n", stop - start);
20 }
```

Programa 162: Cómo medir el tiempo

La función `clock_gettime` es una función para medir el tiempo de un modo peculiar, en sistemas Linux se mide el tiempo desde el primero de enero de 1970. Así, la estructura `timespec` indica el tiempo pasado desde entonces como un conjunto de segundos más los correspondientes nanosegundos en sus dos miembros. Como eso es poco práctico he creado una pequeña función para convertirlo a número decimal y así poder restarlo cómodamente. Después, simplemente mido el tiempo antes y después del código que quiero saber cuánto tarda y los resto, como puedes ver.



17. Ejemplo completo de programa

Esta sección está al final porque, si hasta ahora hemos visto cada parte del lenguaje en detalle y por sí misma, en esta vamos a intentar montar todas las piezas en una gran fotografía. Para esto vamos a utilizar y refinar un ejemplo que ha sido recurrente en el manual: la gestión de una estructura que almacena los datos de una persona, pero vamos a conseguir separar bien al usuario de la funcionalidad interna del código que se encarga de eso.

Lo que haremos es crear un archivo de código fuente llamado `person.c` y su correspondiente archivo de cabeceras, `person.h`, en este archivo incluiremos funcionalidad para crear una estructura persona, cambiar sus atributos, leerlos y serializarla. Además, vamos a ver un interesante artefacto del lenguaje para poder impedir que el usuario se entrometa en nuestra estructura y pueda alterar los datos de manera incorrecta. Por ejemplo: asignando los punteros a una zona de memoria que no controlemos desde estas funciones proporcionadas para manipular la estructura de datos.

Lo primero que voy a hacer es crear el archivo de cabeceras porque ya hemos definido de una manera muy concreta la funcionalidad de este código fuente. Aquí hay una cosa interesante que podremos comentar, veamos el archivo:

```
1 #ifndef PERSON_H
2 #define PERSON_H
3
4 typedef struct person_s person_t;
5
6 person_t *create_person(const char *name, const char *last_name,
7                        unsigned int age);
8
9 void destroy_person(person_t *p);
10
11 void person_set_name(person_t *p, const char *name);
12
13 void person_set_last_name(person_t *p, const char *last_name);
14
15 void person_set_age(person_t *p, unsigned int age);
16
17 const char *person_get_name(const person_t *person);
18
19 const char *person_get_last_name(const person_t *person);
20
21 unsigned int person_get_age(const person_t *person);
22
23 char *person_to_string(const person_t *p);
24
25 #endif
```

Programa 163: Ejemplo final de programa – `person.h`

Y aquí puedes ver una de las cosas interesantes de este ejemplo final: estamos declarando el tipo `person_t`, pero no el `struct` al que da nombre, esto quiere decir que cualquier archivo de código fuente que incluya este **no** podrá saber la definición de tal `struct`. La implicación de esto es que no podrá declarar variables de este tipo, tan solo punteros, puede declarar un puntero, porque todos los punteros tienen el mismo tamaño. Si intentáramos declarar una variable de este tipo, el compilador lanzaría un error como el siguiente:



```
main.c: In function 'main':
main.c:5:14: error: storage size of 'francis' isn't known
    5 |         person_t francis;
      |         ^~~~~~
```

Este es el mecanismo que nos permite impedir que el usuario altere el contenido de la estructura fuera de nuestro control (como comentamos en el programa 64) porque, del mismo modo que no conoce el tamaño del tipo, tampoco conoce los miembros de esta estructura, así que no puede accederse a ellos. Nota, además, como no hemos incluido ninguna cabecera en `person.h`. Si necesitáramos cabeceras, por ejemplo, la cabecera `stdint.h` contiene definiciones de tipo útiles como aquéllos de tamaño fijo: `int8_t`, `int16_t`, etc.; si quisiéramos definir alguna función con un argumento de este tipo o de tipo de retorno, sí sería necesario que incluyéramos esta cabecera. Si las necesitamos en las implementaciones (en las declaraciones de estructuras, en las definiciones de funciones...), será en el archivo de código fuente (en el `.c`) donde las incluiremos.

El siguiente archivo es, precisamente, este archivo de código fuente: `person.c`. Es bastante largo, así que vamos a incluirlo en tres secciones: la sección de declaración de tipos (que sólo contendrá uno), las funciones de manipulación del contenido de la estructura y, finalmente, la de recuperación de la información.



```
1 #include <string.h> //strdup, memset
2 #include <stdlib.h> //malloc
3 #include <stdio.h> //snprintf
4 #include "person.h"
5
6 struct person_s
7 {
8     char *name;
9     char *last_name;
10    unsigned int age;
11 };
12
13 person_t *create_person(const char *name,
14                        const char *last_name,
15                        unsigned int age)
16 {
17     person_t *res = malloc(sizeof(*res));
18     memset(res, 0, sizeof(*res));
19
20     res->age = age;
21     res->name = strdup(name);
22     res->last_name = strdup(last_name);
23
24     return res;
25 }
26
27 void destroy_person(person_t *p)
28 {
29     free(p->name);
30     free(p->last_name);
31     free(p);
32 }
```

Programa 164: Ejemplo final de programa – person.c definiciones

Aquí podemos ver la definición del tipo del que en la cabecera hicimos un typedef, este estilo de declaración de un tipo se llama declaración anticipada o, en inglés, *forwarding declaration*. Aquí, aparte de la definición del tipo propiamente dicho, tenemos las funciones que lo crean y que lo destruyen. Como esta estructura contiene elementos reservados con memoria dinámica, debemos proveer al usuario una manera de liberar los recursos de la estructura. Como puedes ver, en las funciones de creación reservamos espacio **para la propia estructura** y para sus campos.

Debemos reservar nosotros dinámicamente la estructura aparte de sus campos porque, recordemos, fuera de este archivo de código fuente no podremos declarar más que punteros, y ese puntero no tendrá espacio para nada si no lo declaramos. Después, reservamos memoria para el contenido al que apuntarán los **miembros** de la estructura. En la función de destrucción, simétricamente, liberamos primero los contenidos y después la propia estructura. Nota, además, cómo hemos declarado todos los argumentos que hemos podido como constantes, para que el usuario no tenga dudas de si vamos a modificar datos que nos proporcione.

Las siguientes funciones son las que nos permiten sobrescribir los datos:



```
1 void person_set_name(person_t *p, const char *name)
2 {
3     free(p->name);
4     p->name = strdup(name);
5 }
6
7 void person_set_last_name(person_t *p, const char *last_name)
8 {
9     free(p->last_name);
10    p->last_name = strdup(last_name);
11 }
12
13 void person_set_age(person_t *p, unsigned int age)
14 {
15     p->age = age;
16 }
```

Programa 165: Ejemplo final de programa – person.c manipulación

Como puedes ver, las funciones son simples, liberamos la memoria de los campos y después le asignamos la duplicación del argumento que se nos pasa. De nuevo, observa cómo hemos definido como constantes los argumentos del mismo modo que hicimos en la función de creación. Las funciones no devuelven nada (void) porque no tendría sentido. Aunque siempre podrían devolver un entero que actuara como código de error, por ejemplo si la reserva de memoria fallara, se podría indicar devolviendo un número menor que cero.



```
1  const char *person_get_name(const person_t *p)
2  {
3      return p->name;
4  }
5
6  const char *person_get_last_name(const person_t *person)
7  {
8      return person->last_name;
9  }
10
11 unsigned int person_get_age(const person_t *person)
12 {
13     return person->age;
14 }
15
16 char *person_to_string(const person_t *p)
17 {
18     #define MAX_STRING_SIZE ((unsigned int)1024)
19
20     char res[MAX_STRING_SIZE + 1];
21     snprintf(res, MAX_STRING_SIZE, "{ \"name\": \"%s\", \"
22                                     \"last_name\": \"%s\", \"
23                                     \"age\": %u }",
24             p->name, p->last_name, p->age);
25     return strdup(res);
26     #undef MAX_STRING_SIZE
27 }
```

Programa 166: Ejemplo final de programa – person.c recuperación

Aquí debes notar que devolvemos punteros constantes a `char`, precisamente para impedir que el usuario libere, manipule o cambie el contenido de los campos del *struct*. Sin embargo; en la función de serialización (que he reducido a su versión más simple) devuelvo un puntero no constante porque la responsabilidad de liberar es del usuario de la funcionalidad, no de esta biblioteca. Además, en esta última función puedes ver que podemos **eliminar** una macro con la directiva `#undef`. Esto es útil cuando necesitas inicializar un array, como aquí, pero no quieres contaminar de símbolos el código fuente. Así, si otra función usara strings de otro tamaño, podríamos usar el mismo nombre, como si la macro fuera una variable distinta. De nuevo: ten cuidado, las macros trabajan a nivel de preprocesado, por lo que no estás definiendo ninguna variable en la función, sólo una región de código donde un símbolo se sustituirá por otro.

Finalmente, en el archivo principal podemos utilizar la funcionalidad:



```
1 #include "person.h"
2 #include <stdlib.h>
3 #include <stdio.h>
4
5 int main(void)
6 {
7     person_t* person = create_person("John", "Smith", 18);
8
9     char* serialization = person_to_string(person);
10
11     printf("%s\n", serialization);
12     free(serialization);
13
14     person_set_name(person, "Michael");
15     person_set_last_name(person, "Johnson");
16     person_set_age(person, 33);
17
18     serialization = person_to_string(person);
19     printf("%s\n", serialization);
20     free(serialization);
21     destroy_person(person);
22 }
```

Programa 167: Ejemplo final de programa – main.c

Aquí se puede ver cómo se utilizan estructuras con este patrón de diseño. Primero la reservas, después la usas, la puedes manipular y, finalmente, la liberas, todo ello con las funciones proporcionadas junto con el tipo de dato. Con este patrón, el usuario de la funcionalidad que hemos programado tiene menos capacidad para «hacer algo mal».

Ahora, vamos a ver rápidamente cómo se podría compilar, para recordarlo. Primero lo haremos utilizando el código objeto y, después, crearemos una biblioteca dinámica y la enlazaremos. Para compilar utilizando el código objeto seguiremos estos pasos:

1. Crear el código objeto de person.c

```
$ gcc -c person.c
```

2. Crear el código objeto de main.c

```
$ gcc -c main.c
```

3. Crear el ejecutable con ambos códigos objeto

```
$ gcc -o main.exe main.o person.o -g -Wall -Wextra
```

Para la biblioteca, seguiremos estos pasos:



1. Crear el código objeto de `person.c`

```
$ gcc -c person.c
```

2. Crear una biblioteca con este código objeto:

```
$ gcc -shared -o libperson.so person.o
```

3. Crear el código objeto de `main.c`

```
$ gcc -c main.c
```

4. Crear el ejecutable usando la biblioteca:

```
$ gcc -L. -Wl,-rpath=. -o main.exe main.o -lperson
```

En este ejemplo final se han visto ejemplos de la mayoría de conceptos que se han explicado en el manual: variables, punteros, memoria, reserva dinámica, estructuras, macros, enlazado, compilación y constancia y signo. Es mucha información en pocas páginas, pero permite tener una foto global de todo si ya se ha leído antes con detenimiento.



18. Anexo A: soluciones a ejercicios

Ej. 1: Escribe un programa y declara en él una estructura que defina un círculo en dos dimensiones (su centro y su radio). Y haz que el programa declare una variable de ese tipo y calcule su área.

```
1 #include <stdio.h>
2
3 struct circle_s {
4     double x;
5     double y;
6     double r;
7 };
8
9 int main(void)
10 {
11     struct circle_s circle = { 1 , 1 , 3.4 };
12     double area = 3.141592 * circle.r * circle.r;
13     printf("El área del círculo en el punto [%f, %f] con un radio de %
14           f es: %f\n", circle.x, circle.y, circle.r, area);
15 }
```

Programa 168: Solución al ejercicio 1

Ej. 2: Haz un programa que, basándose en el struct punto presentado en el ejemplo, declare e inicialice un array de ellos y vaya diciendo las direcciones que hay que seguir para ir de uno a otro.



```
1 #include <stdio.h>
2 struct point_s {
3     double x;
4     double y;
5 };
6
7 int main(void)
8 {
9     struct point_s points[10] = { {-1.056171, 3.401877},
10                                     {2.984400, 2.830992},
11                                     {-3.024486, 4.116474},
12                                     {2.682296, -1.647772},
13                                     {0.539700, -2.222253},
14                                     {1.288709, -0.226029},
15                                     {0.134009, -1.352155},
16                                     {4.161951, 4.522297},
17                                     {2.172969, 1.357117},
18                                     {1.069689, -3.583974} };
19
20     for(int ii = 1; ii < 10; ++ii){
21         if (points[ii - 1].x < points[ii].x) {
22             printf("Derecha");
23         }else if(points[ii - 1].x == points[ii].x){
24             printf("Quieto");
25         }else if(points[ii - 1].x > points[ii].x){
26             printf("Izquierda");
27         }
28         printf(", ");
29         if (points[ii - 1].y < points[ii].y) {
30             printf("Arriba");
31         }else if(points[ii - 1].y == points[ii].y){
32             printf("Quieto");
33         }else if(points[ii - 1].y > points[ii].y){
34             printf("Abajo");
35         }
36         printf("\n");
37     }
38 }
```

Programa 169: Solución al ejercicio 2

Ej. 3: Haz un programa que declare un array bidimensional y calcule la suma de sus filas y sus columnas.



```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     int array[3][3] = { {1,3,6},{7,3,6},{1,2,4} };
6
7     for (int ii = 0; ii < 3; ++ii) {
8         for (int jj = 0; jj < 3; ++jj) {
9             printf("%d ", array[ii][jj]);
10        }
11        int suma = 0;
12        for(int jj = 0; jj < 3; ++jj){
13            suma += array[ii][jj];
14        }
15        printf("= %d\n", suma);
16    }
17    for(int ii = 0; ii < 3*2; ++ii){
18        printf("-");
19    }
20    printf("\n");
21    for(int ii = 0; ii < 3; ++ii){
22        int suma = 0;
23        for(int jj = 0; jj < 3; ++jj){
24            suma+=array[jj][ii];
25        }
26        printf("%d ", suma);
27    }
28    printf("\n");
29 }
```

Programa 170: Solución al ejercicio 3

Ej. 4: Haz un programa que haga lo siguiente para los números del 1 al 100 ambos incluidos: si el número es divisible entre 2, debe imprimirse por pantalla «fizz», si es divisible entre 5, «buzz», y si es divisible entre los dos, «fizzbuzz», no imprimir nada en otro caso.



```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     for (int ii = 1; ii <= 100; ++ii){
6         int end_of_line = 0;
7         if (ii % 2 == 0){
8             printf("fizz");
9             end_of_line = 1;
10        }
11
12        if(ii % 5 == 0){
13            printf("buzz");
14            end_of_line = 1;
15        }
16        if(end_of_line){
17            printf("\n");
18        }
19    }
20 }
```

Programa 171: Solución al ejercicio 4

Ej. 5: Escribe una función que calcule si un número es primo o no.

```
1 #include <stdio.h>
2
3 int is_prime(int number) {
4     int prime = 1;
5     for (int ii = 2; ii < number / 2 && prime; ++ii) {
6         if (0 == number % ii) {
7             prime = 0;
8         }
9     }
10    return prime;
11 }
12
13 int main(void)
14 {
15     for(int ii = 2; ii < 100; ii++){
16         printf("El número %d ", ii);
17         if(is_prime(ii)){
18             printf("es primo.");
19         }else{
20             printf("no es primo");
21         }
22         printf("\n");
23     }
24 }
```

Programa 172: Solución al ejercicio 5



Ej. 6: Escribe una función que calcule la distancia entre dos estructuras punto de las usadas en la sección anterior.

```
1 #include <stdio.h>
2 #include <math.h>
3 struct point_s {
4     double x;
5     double y;
6 };
7
8 double distance(struct point_s a, struct point_s b) {
9     double res = 0.0;
10    double diff_x = a.x - b.x;
11    double diff_y = a.y - b.y;
12    res = sqrt(diff_x * diff_x + diff_y * diff_y);
13    return res;
14 }
15
16 int main(void)
17 {
18     struct point_s a = {1.2, 4.3};
19     struct point_s b = {3.4, 5.5};
20    printf("La distancia entre [%f, %f] y [%f, %f] es: %f\n", a.x, a.y,
21           b.x, b.y, distance(a,b));
22 }
```

Programa 173: Solución al ejercicio 6

Ej. 7: Escribe una función que reciba un array de enteros y un caracter separador que imprima los elementos del array separados por ese caracter.

```
1 #include <stdio.h>
2 void print_separated(int array[], int array_size, char separator){
3     for(int ii = 0; ii < array_size; ++ii){
4         printf("%d%c", array[ii], separator);
5     }
6 }
7
8 int main(void)
9 {
10     int my_array[] = {1,2,3,4,5,6,7,8,9,0};
11     print_separated(my_array, 10, '|');
12     printf("\n");
13 }
```

Programa 174: Solución al ejercicio 7

Ej. 8: Escribe una función que encapsule el programa 17: Programa de resolución de ecuaciones lineales con condicionales. La función debe recibir los coeficientes de las ecuaciones (a, b, c, d, e y f). Puede recibirlos por separado o en un array. Para devolver el resultado puedes crear una estructura que simplemente tenga dos double.

```
1 #include <stdio.h>
2
3 struct solution_s {
```



```
4     double x;
5     double y;
6     int solved;
7 };
8
9 struct solution_s linear_system(int a, int b, int c, int d, int e, int
    f) {
10
11     double divisor;
12     struct solution_s res;
13     res.solved = 1;
14     if (a != 0 && d != 0) {
15         divisor = (a * e - d * b);
16         if (divisor == 0)
17         {
18             printf("El sistema es irresoluble .\n");
19             res.solved = 0;
20         }
21         else
22         {
23             res.y = (a * f - d * c) / divisor;
24             res.x = (f - e * res.y) / (d);
25         }
26     }
27     else if (b != 0 && e != 0) {
28         divisor = (b * d - e * a);
29         if (divisor == 0) {
30             printf("El sistema es irresoluble .\n");
31             res.solved = 0;
32         }
33         else {
34             res.x = (b * f - e * c) / divisor;
35             res.y = (c - a * res.x) / b;
36         }
37     }
38     else if ((a == 0 && b == 0) || (d == 0 && e == 0)) {
39         printf(" Esto no es un sistema \n");
40         res.solved = 0;
41     }
42     else {
43         if (a != 0) {
44             res.x = (double)c / a;
45             res.y = (double)f / e;
46         }
47         else {
48             res.x = (double)f / d;
49             res.y = (double)c / b;
50         }
51     }
52     return res;
53 }
54
55
```



```

56 int main(void)
57 {
58     struct solution_s sol = linear_system(1, 1, 1, 2, 2, 2);
59     printf(" %dx+ %dy= %d\n", 1, 2, 3);
60     printf(" %dx+ %dy= %d\n", 4, 5, 6);
61     if (sol.solved) {
62         printf("x = %f; y = %f\n", sol.x, sol.y);
63     }
64     else {
65         printf("El sistema no tiene solucion.\n");
66     }
67 }

```

Programa 175: Solución al ejercicio 8

Ej. 9: Escribe una función que normalice los elementos de un array de double.

```

1  #include <stdio.h>
2
3  void normalize(double array[], int array_size) {
4      double biggest = array[0];
5      for (int ii = 1; ii < array_size; ++ii) {
6          if (array[ii] > biggest) {
7              biggest = array[ii];
8          }
9      }
10     for (int ii = 0; ii < array_size; ++ii) {
11         array[ii] /= biggest;
12     }
13 }
14
15 int main(void)
16 {
17     double array[] = { 1,2,3,4,5,6,7,8,9,10 };
18     normalize(array, 10);
19     for(int ii = 0; ii < 10; ++ii){
20         printf("%f\n", array[ii]);
21     }
22     printf("\n");
23 }

```

Programa 176: Solución al ejercicio 9

Ej. 10: Completa esta tabla de números en diferentes bases numéricas:

Decimal	Binario	Hexadecimal
73	0100 1001	0x049
38	0010 0110	0x026
303	0001 0010 1111	0x12F
128	1000 0000	0x080

Ej. 11: Vuelve al ejercicio noveno y reproduce los contenidos de la pila en cada bloque de código del programa. Utiliza de referencia la solución que propongo yo.



1. Función main
 1. Array (10 elementos)
 2. Entramos en la función normalize
 1. Array (puntero a)
 2. array_size
 3. biggest
 4. Primer bucle for
 1. ii
 5. Segundo bucle for
 1. ii
 3. Bucle for
 1. ii

Ej. 12: Haz un programa que cree un puntero de tres niveles de tipo `int`, lo reserve correctamente, lo rellene con el valores correlativos **empezando en uno** y después lo imprima de una manera comprensible. Finalmente, libéralo también de tal modo que no quede memoria sin liberar al final del programa.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 #define DEPTH (10)
5 #define WIDTH (5)
6 #define HEIGHT (12)
7
8 int main(int argc, char const *argv[]) {
9     int ***cube = malloc(sizeof(*cube) * DEPTH);
10
11     for (int ii = 0; ii < DEPTH; ++ii) {
12         cube[ii] = malloc(sizeof(**cube) * HEIGHT);
13         for (int jj = 0; jj < HEIGHT; ++jj) {
14             cube[ii][jj] = malloc(sizeof(***cube) * WIDTH);
15             for (int kk = 0; kk < WIDTH; ++kk) {
16                 cube[ii][jj][kk] =
17                     kk + jj * WIDTH + ii * HEIGHT * WIDTH + 1;
18             }
19         }
20     }
21
22     for (int ii = 0; ii < DEPTH; ++ii) {
23         for (int jj = 0; jj < HEIGHT; ++jj) {
24             for (int kk = 0; kk < WIDTH; ++kk) {
25                 printf("%3d ", cube[ii][jj][kk]);
26             }
27             printf("\n");
28         }
29         printf("\n");
30     }
31
32     for (int ii = 0; ii < DEPTH; ++ii) {
33         for (int jj = 0; jj < HEIGHT; ++jj) {
34             free(cube[ii][jj]);
35         }
36         free(cube[ii]);
37     }
38     free(cube);
39
40     return 0;
41 }
```

Programa 177: Solución al ejercicio 12

Ej. 13: Basándote en el programa anterior, crea dos funciones, una para crear una matriz tridimensional con memoria dinámica dadas sus tres dimensiones y otra para liberarla.



```
1 int ***malloc_cube(size_t depth, size_t height, size_t width) {
2     int ***cube = malloc(sizeof(*cube) * depth);
3
4     for (int ii = 0; ii < depth; ++ii) {
5         cube[ii] = malloc(sizeof(**cube) * height);
6         for (int jj = 0; jj < height; ++jj) {
7             cube[ii][jj] = malloc(sizeof(**cube) * width);
8             for (int kk = 0; kk < width; ++kk) {
9                 cube[ii][jj][kk] =
10                     kk + jj * width + ii * height * width + 1;
11             }
12         }
13     }
14     return cube;
15 }
```

Programa 178: Solución al ejercicio 13 – reserva

```
1 void print_cube(int ***cube, size_t depth, size_t height,
2                 size_t width) {
3     for (int ii = 0; ii < depth; ++ii) {
4         for (int jj = 0; jj < height; ++jj) {
5             for (int kk = 0; kk < width; ++kk) {
6                 printf("%3d ", cube[ii][jj][kk]);
7             }
8             printf("\n");
9         }
10        printf("\n");
11    }
12 }
```

Programa 179: Solución al ejercicio 13 – impresión

```
1 void free_cube(int ***cube, size_t depth, size_t height,
2                size_t width) {
3     for (int ii = 0; ii < depth; ++ii) {
4         for (int jj = 0; jj < height; ++jj) {
5             free(cube[ii][jj]);
6         }
7         free(cube[ii]);
8     }
9     free(cube);
10 }
```

Programa 180: Solución al ejercicio 13 – liberación



```
1 int main(int argc, char const *argv[]) {
2     int ***cube = malloc_cube(DEPTH, HEIGHT, WIDTH);
3     print_cube(cube, DEPTH, HEIGHT, WIDTH);
4     free_cube(cube, DEPTH, HEIGHT, WIDTH);
5 }
```

Programa 181: Solución al ejercicio 13 – función main

Ej. 14: Escribe un programa que reciba un número variable de números como argumentos e imprima la descomposición en factores primos de todo ellos. Se recomienda hacer control de errores comprobando que los argumentos son números antes de utilizarlos, etc.

```
1 int main(int argc, char const *argv[]) {
2     int ***cube = malloc_cube(DEPTH, HEIGHT, WIDTH);
3     print_cube(cube, DEPTH, HEIGHT, WIDTH);
4     free_cube(cube, DEPTH, HEIGHT, WIDTH);
5 }
```

Programa 182: Solución al ejercicio 13 – función main

Ej. 15: Escribe un programa que lea **por consola** una serie de palabras y que sólo deje de leer cuando se introduzca «!!» como palabra. Después, debe imprimir dichas palabras en orden aleatorio. La función rand devuelve un número aleatorio entre cero y el máximo entero positivo. Si quieres que devuelva números aleatorios **distintos** cada vez debes ejecutar `srand(time(NULL))`; al inicio de la función main. Debes incluir la cabecera `time.h`.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <time.h>
5
6 #define STRING_SIZE (1024)
7 #define MAX_WORDS (1024)
8
9 int main(int argc, char const *argv[]) {
10     char *word_set[1024];
11     int word_length = 0;
12     do {
13         word_set[word_length] = malloc(STRING_SIZE);
14         scanf("%s", word_set[word_length]);
15         word_length++;
16     } while (strcmp(word_set[word_length - 1], "!!"));
17
18     srand(time(NULL));
19     for (int ii = 0; ii < word_length - 1; ++ii) {
20         char *aux = word_set[ii];
21         int rand_index = rand() % (word_length - 1);
22         word_set[ii] = word_set[rand_index];
23         word_set[rand_index] = aux;
24     }
25     for (int ii = 0; ii < word_length - 1; ++ii) {
26         printf("%s\n", word_set[ii]);
27         free(word_set[ii]);
28     }
29     free(word_set[word_length-1]);
30 }
```

Programa 183: Solución al ejercicio 15

Como nota, para «barajar» el vector de palabras lo que hago es recorrerlo intercambiando cada palabra con una posición aleatoria. Hay otros métodos que quizás hayas usado como generar una posición aleatoria del vector y copiarlo a otro, el problema de esto es que si lo que haces es generar un índice nuevo cuando encuentras que ya has copiado ese, el número de veces que ejecutas el aleatorio es, lógicamente, impredecible. Tal y como lo he escrito yo el algoritmo siempre tardará lo mismo generando resultados moderadamente aleatorios.

Ej. 16: Haz una función que lea dos archivos e **intercambie** su contenido, escribe dicho programa de tal modo que no sea necesario alojar ninguno de los dos archivos en memoria completamente.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(int argc, char const *argv[]) {
5     FILE *file_1 = NULL, *file_2 = NULL, *file_aux = NULL;
6     const int BLOCK_SIZE = 1024;
7     int read = 0;
8     char buffer[BLOCK_SIZE], aux_file_path[] = "/tmp/auxFile.txt";
9     if (argc < 3) {
10         printf("Uso: main.exe <archivo1> <archivo2>\n");
11         return EXIT_FAILURE;
12     }
13
14     file_1 = fopen(argv[1], "r+");
15     if (NULL == file_1) {
16         printf("ERROR: El primer archivo no existe.\n");
17         return EXIT_FAILURE;
18     }
19
20     file_2 = fopen(argv[2], "r+");
21     if (NULL == file_2) {
22         printf("ERROR: El segundo archivo no existe.\n");
23         fclose(file_1);
24         return EXIT_FAILURE;
25     }
26
27     file_aux = fopen(aux_file_path, "w+");
28     if (NULL == file_aux) {
29         fclose(file_1);
30         fclose(file_2);
31         return EXIT_FAILURE;
32     }
33
34     // copy file 1 to aux
35     while (read = fread(buffer, sizeof(char), BLOCK_SIZE, file_1)) {
36         fwrite(buffer, sizeof(char), read, file_aux);
37     }
38     fclose(file_1);
39     file_1 = fopen(argv[1], "w+");
40     if (NULL == file_1) {
41         printf("Error, el primer archivo no se ha podido reabrir\n");
42     }
43
44     // copy file 2 to 1
45     while (read = fread(buffer, sizeof(char), BLOCK_SIZE, file_2)) {
46         fwrite(buffer, sizeof(char), read, file_1);
47     }
48
49     fclose(file_2);
50     file_2 = fopen(argv[2], "w+");
51     if (NULL == file_2) {
52         printf("Error, el segundo archivo no se ha podido reabrir\n");
53     }
```



```
54
55 // copy aux file to file 2, we need to go back to begin of file aux
56 fseek(file_aux, 0, SEEK_SET);
57 while (read = fread(buffer, sizeof(char), BLOCK_SIZE, file_aux)) {
58     fwrite(buffer, sizeof(char), read, file_2);
59 }
60
61 fclose(file_1);
62 fclose(file_2);
63 fclose(file_aux);
64 remove(aux_file_path);
65 }
```

Programa 184: Solución al ejercicio 16

Ej. 17: Escribe una función que reciba una palabra como argumento e indique en qué posición (en bytes) dentro del archivo se encuentra la palabra. Sólo tienes que dar la primera ocurrencia, si la palabra no se encuentra, devuelve un número negativo. Haz un programa que, con esa función, reciba una ruta a un archivo y una palabra e imprima el resultado de buscar la palabra en el archivo.



```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int find_in_file(const char *path, const char *word) {
6      FILE *file = NULL;
7      char *buffer;
8      int word_length = 0, read = 0, pos = -1;
9
10     file = fopen(path, "r+");
11     if (NULL == file) {
12         printf("ERROR: El archivo no existe.\n");
13         return -1;
14     }
15     word_length = strlen(word);
16     buffer = malloc(sizeof(char) * word_length * 2);
17
18     while (read = fread(buffer, sizeof(char), word_length * 2, file)) {
19         fseek(file, word_length - read, SEEK_CUR);
20         for (int ii = 0; ii < word_length; ++ii) {
21             char local_word[word_length + 1];
22             memcpy(local_word, buffer + ii, word_length);
23             local_word[word_length] = 0;
24             if (!strcmp(local_word, word)) {
25                 pos = ftell(file) + ii - word_length;
26                 goto end;
27             }
28         }
29     }
30 end:
31     free(buffer);
32     fclose(file);
33     return pos;
34 }
35
36 int main(int argc, char const *argv[]) {
37
38     int pos = find_in_file(argv[1], argv[2]);
39     printf("La palabra %s está en la posición %d en el archivo %s\n",
40         argv[2], pos, argv[1]);
41 }

```

Programa 185: Solución al ejercicio 17

Aquí puedes ver un uso típico de la instrucción `goto`, como necesitamos hacer lo mismo encontremos la palabra o no, lo que hacemos es establecer una etiqueta y saltar allí para liberar recursos y devolver el resultado.

Ej. 18: Reescribe el ejercicio 15 prescindiendo del array estático de punteros a `char`. (Usa `realloc` y `strdup`).



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <time.h>
5
6 int main(int argc, char const *argv[]) {
7     char word[1024];
8     char **word_set = NULL;
9     int word_length = 0;
10    do {
11        word_set = realloc(word_set, ++word_length * sizeof(char *));
12        scanf("%s", word);
13        word_set[word_length - 1] = strdup(word);
14    } while (strcmp(word, "!!"));
15
16    srand(time(NULL));
17    for (int ii = 0; ii < word_length - 1; ++ii) {
18        char *aux = word_set[ii];
19        int rand_index = rand() % (word_length - 1);
20        word_set[ii] = word_set[rand_index];
21        word_set[rand_index] = aux;
22    }
23    for (int ii = 0; ii < word_length - 1; ++ii) {
24        printf("%s\n", word_set[ii]);
25    }
26    for (int ii = 0; ii < word_length; ++ii) {
27        free(word_set[ii]);
28    }
29    free(word_set);
30 }
```

Programa 186: Solución al ejercicio 18

Ej. 19: Escribe un programa que reciba un número indeterminado de palabras como argumentos y los ordene alfabéticamente y que, después, los imprima.



```
1 void generic_swap(void *one, void *other, size_t type_size) {
2 //...
3
4 void generic_bubble_sort(void *array, size_t array_size,
5 //...
6
7 int compare_strings(const void *one, const void *two) {
8 //...
9
10 int compare_strings(const void* one, const void* two) {
11 //...
12
13 int main(int argc, char const *argv[]) {
14
15     generic_bubble_sort(argv + 1, argc - 1, sizeof(char *),
16                         compare_strings);
17
18     for (int ii = 1; ii < argc; ++ii) {
19         printf("%s\n", argv[ii]);
20     }
21 }
```

Programa 187: Solución al ejercicio 19

He usado las funciones de ejemplo para ordenar, así que omito su contenido, simplemente tenemos que utilizar el comparador adecuado y tener en cuenta que el primer argumento es el nombre de programa, que no queremos ordenar. Además, puedes ver que podemos modificar el orden de los argumentos, pero no su contenido, al haber declarado `argv` como `char const*argv[]` que quiere decir un array (puntero) no constante a `char` constante. Es decir, como ya vimos en la figura 11.

Ej. 20: Haz un programa que reciba como argumento una palabra y un número. Si el número es cero, debe convertir la palabra a minúscula, si el número es distinto de cero, debe convertirla a mayúscula.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 char char_to_upper_case(char c) {
5     if (c < 123 && c > 96) {
6         return c - 32;
7     }
8     return c;
9 }
10
11 char char_to_lower_case(char c) {
12     if (c < 91 && c > 64) {
13         return c + 32;
14     }
15     return c;
16 }
17
18 char string_to_upper_case(char *message) {
19     int length = strlen(message);
20     for (int ii = 0; ii < length; ++ii) {
21         message[ii] = char_to_upper_case(message[ii]);
22     }
23 }
24
25 char string_to_lower_case(char *message) {
26     int length = strlen(message);
27     for (int ii = 0; ii < length; ++ii) {
28         message[ii] = char_to_lower_case(message[ii]);
29     }
30 }
31
32 int main(int argc, char const *argv[]) {
33
34     char *message = strdup(argv[1]);
35     int code = atoi(argv[2]);
36     if(code){
37         string_to_upper_case(message);
38     }else{
39         string_to_lower_case(message);
40     }
41     printf("%s\n", message);
42     free(message);
43 }
```

Programa 188: Solución al ejercicio 20

Aquí hemos utilizado dos funciones diferentes para poner a mayúscula y minúscula, otra opción sería utilizar un parámetro lógico (o incluso un enumerado) para indicar qué tipo de letras se quiere y llamar a una función que reciba ese parámetro y actúe en consecuencia. Puedes implementarlo así como ejercicio extra.

Ej. 21: Crea un programa que dado un número como argumento imprima una pirámide como esta de tantos pisos como el número indicado.



```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 char **make_pyramid(int steps) {
6     char **result = malloc(sizeof(*result) * steps);
7     for (int ii = 0; ii < steps; ++ii) {
8         result[ii] = malloc(sizeof(**result) * (steps * 2));
9         memset(result[ii], ' ', steps * 2 - 1);
10        memset(result[ii] + ii, '%', (steps * 2 - 1) - 2 * ii);
11        result[ii][(steps - 1) * 2 + 1] = 0;
12    }
13    return result;
14 }
15
16 void free_pyramid(char **pyramid, int steps) {
17     for (int ii = 0; ii < steps; ++ii) {
18         free(pyramid[ii]);
19     }
20     free(pyramid);
21 }
22
23 int main(int argc, char const *argv[]) {
24     int steps = atoi(argv[1]);
25     char **pyramid = make_pyramid(steps);
26     for (int ii = 0; ii < steps; ++ii) {
27         printf("%s\n", pyramid[ii]);
28     }
29     free_pyramid(pyramid, steps);
30 }
```

Programa 189: Solución al ejercicio 21

Ej. 22: Escribe un programa que reciba una serie de puntos y de nombres para cada uno y después los imprima en orden de su distancia al origen de menor a mayor.



```
1 #ifndef TAGGED_POINT_H
2 #define TAGGED_POINT_H
3 typedef struct tagged_point_s tagged_point_t;
4
5 tagged_point_t *tagged_point_create(const char *tag, double x,
6                                     double y);
7
8 void tagged_point_set_tag(const char *tag, tagged_point_t *tp);
9
10 void tagged_point_set_x(double x, tagged_point_t *tp);
11
12 void tagged_point_set_y(double y, tagged_point_t *tp);
13
14 const char *tagged_point_get_tag(const tagged_point_t *tp);
15
16 double tagged_point_get_x(const tagged_point_t *tp);
17
18 double tagged_point_get_y(const tagged_point_t *tp);
19
20 void tagged_point_destroy(tagged_point_t *tp);
21
22 double tagged_point_distance(const tagged_point_t *a,
23                             const tagged_point_t *b);
24 #endif
```

Programa 190: Solución al ejercicio 23 – tagged_point.h



```
1 #include "tagged_point.h"
2 #include <math.h>
3 #include <stdlib.h>
4 #include <string.h>
5
6 struct tagged_point_s {
7     char *tag;
8     double x, y;
9 };
10
11 tagged_point_t *tagged_point_create(const char *tag, double x,
12                                     double y) {
13     tagged_point_t *res = malloc(sizeof(tagged_point_t));
14     res->x = x;
15     res->y = y;
16     res->tag = strdup(tag);
17 }
18
19 void tagged_point_set_tag(const char *tag, tagged_point_t *tp) {
20     free(tp->tag);
21     tp->tag = strdup(tag);
22 }
23
24 void tagged_point_set_x(double x, tagged_point_t *tp) { tp->x = x; }
25
26 void tagged_point_set_y(double y, tagged_point_t *tp) { tp->y = y; }
27
28 const char *tagged_point_get_tag(const tagged_point_t *tp) {
29     return tp->tag;
30 }
31
32 double tagged_point_get_x(const tagged_point_t *tp) { return tp->x; }
33
34 double tagged_point_get_y(const tagged_point_t *tp) { return tp->y; }
35
36 void tagged_point_destroy(tagged_point_t *tp) {
37     free(tp->tag);
38     free(tp);
39 }
40
41 double tagged_point_distance(const tagged_point_t *a,
42                              const tagged_point_t *b) {
43     tagged_point_t origin = {"origin", 0.0, 0.0};
44     if (NULL == a) {
45         a = &origin;
46     }
47     if (NULL == b) {
48         b = &origin;
49     }
50     return sqrt((a->x - b->x) * (a->x - b->x) +
51                (a->y - b->y) * (a->y - b->y));
52 }
```

Programa 191: Solución al ejercicio 22 – tagged_point.c



```
1 #include "tagged_point.h"
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5
6 typedef int(comparator_t)(const void *, const void *);
7
8 void generic_swap(void *one, void *other, size_t type_size) {
9     // ...
10
11 void generic_bubble_sort(void *array, size_t array_size,
12 // ...
13
14 int compare_distance(const void *a, const void *b) {
15     tagged_point_t *p1 = *(tagged_point_t **)a;
16     tagged_point_t *p2 = *(tagged_point_t **)b;
17     return tagged_point_distance(NULL, p1) <
18         tagged_point_distance(NULL, p2);
19 }
20
21 int main(int argc, char const *argv[]) {
22
23     int point_lenght = 0;
24     if ((argc - 1) % 3 != 0) {
25         printf("Algo parece estar mal.");
26         return EXIT_FAILURE;
27     }
28     point_lenght = (argc - 1) / 3;
29     tagged_point_t *points[point_lenght];
30     for (int ii = 0; ii < point_lenght; ++ii) {
31         double x = atof(argv[1 + ii * 3 + 0]);
32         double y = atof(argv[1 + ii * 3 + 1]);
33         const char *tag = argv[1 + ii * 3 + 2];
34         points[ii] = tagged_point_create(tag, x, y);
35     }
36
37     generic_bubble_sort(points, point_lenght, sizeof(tagged_point_t *),
38         compare_distance);
39
40     for (int ii = 0; ii < point_lenght; ++ii) {
41         printf("%f %f %s\n", tagged_point_get_x(points[ii]),
42             tagged_point_get_y(points[ii]),
43             tagged_point_get_tag(points[ii]));
44         tagged_point_destroy(points[ii]);
45     }
46 }
```

Programa 192: Solución al ejercicio 22 – main.c