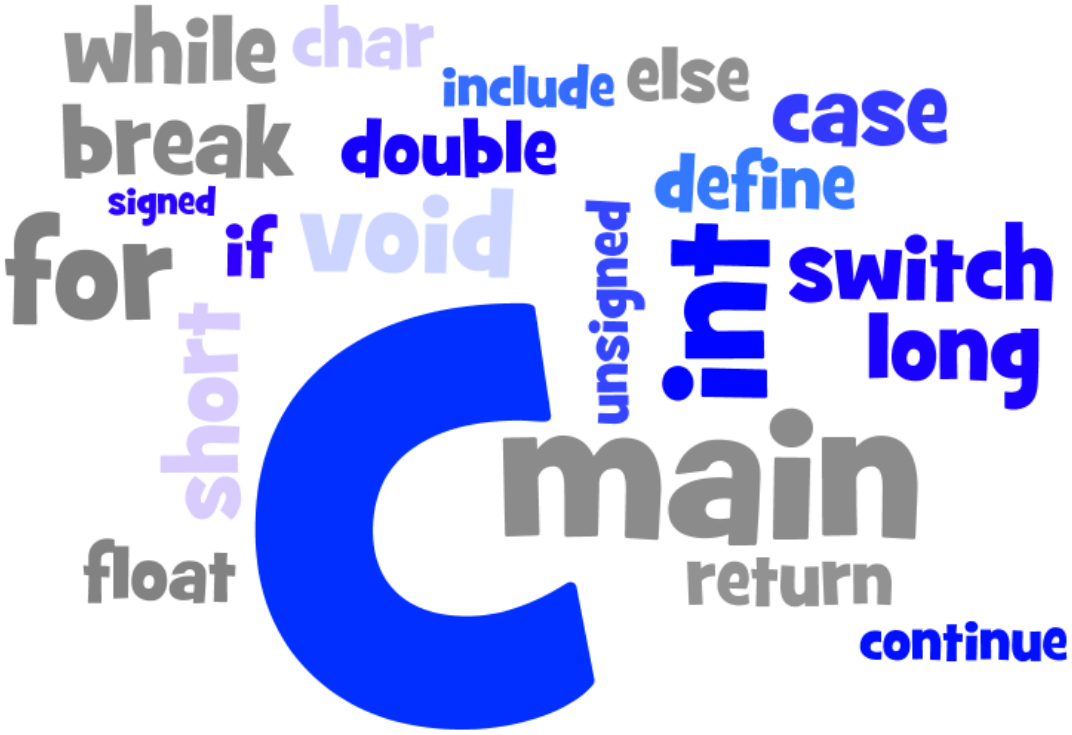
industriales
etsii

Escuela Técnica Superior de Ingeniería Industrial

Centro Universitario de la Defensa

Universidad Politécnica de Cartagena

INFORMÁTICA APLICADA PRÁCTICAS PARA APRENDER A PROGRAMAR EN LENGUAJE C



Pedro J. García Laencina
Pedro María Alcover Garau

INFORMÁTICA APLICADA
PRÁCTICAS PARA APRENDER
A PROGRAMAREN LENGUAJE C

Dr. D. Pedro José García Laencina

Centro Universitario de la Defensa
en la Academia General del Aire

Dr. D. Pedro María Alcover Garau

Universidad Politécnica de Cartagena

Septiembre 2014

Primera Edición, Agosto 2012.

(Revisado Agosto 2013)

(Revisado Septiembre 2014)

Publicado por:	Centro Universitario de la Defensa (CUD) de San Javier Base Aérea de San Javier C/ Coronel López Peña, s/n 30720 Santiago de la Ribera, Murcia (España).
Impresión:	Morpi S.L.
ISBN:	978-84-939010-7-3
DL:	MU 688-2012

IMPRESO EN ESPAÑA / PRINTED IN SPAIN

"The only way to learn a new programming language is by writing programs in it"

B. W. Kernighan[1]

[1]Brian Wilson Kernighan. Coautor, junto con Dennis Ritchie, de "El lenguaje de programación C.", 1978. Ese libro fue, durante años, la "Biblia" del C. Así lo cuenta Ritchie en "The Development of the C Language", April, 1993: "*In the middle of this second period (between 1977 and 1979), the first widely available description of the language appeared: The C Programming Language, often called the 'white book' or 'K&R' [Kernighan 78].*" Y como explica más adelante, "*In 1978, Kernighan and Ritchie wrote the book that became the language definition for several years.*"

El Lenguaje C fue diseñado e implementado por Ritchie; Kernighan fue el primero en escribir un tutorial para el lenguaje C. La génesis de ese libro la cuenta el propio Ritchie en el mismo artículo citado en el párrafo anterior: "*Although we worked closely together on this book, there was a clear division of labor: Kernighan wrote almost all the expository material, while I was responsible for the appendix containing the reference manual and the chapter on interfacing with the Unix system.*"

PRÓLOGO

Aprender el primer lenguaje de programación es una labor engorrosa y con frecuencia antipática. Requiere bastante estudio; pero especialmente exige, de quien emprende esta tarea de aprendizaje, muchas horas de trabajo delante de un ordenador. Y es que, aunque pueda parecer una obviedad, en realidad no hay otro camino: *para aprender a programar, hay que programar*.

Ésta es una recomendación clásica que se repite en el primer día de clase de cualquier asignatura relacionada con la programación y el desarrollo software. Ni mucho menos es suficiente con copiar y entender el código que el profesor explica en clase, el alumno debe llegar a ser capaz de crear su propio código. A algunos alumnos les ocurre que con relativa facilidad comienzan a comprender el código propuesto por otros; pero eso no implica ni de lejos que ya hayan aprendido a programar.

Hay que empezar desarrollando programas muy cortos y sencillos: nunca olvidará su primer programa llamado "Hola Mundo". Para implementar esos primeros programas únicamente deberá copiar literalmente el código y compilar. Aunque eso pueda parecer una tarea fácil, en sus primeros pasos cometerá una gran cantidad de errores sintácticos y surgen las típicas preguntas del tipo ¿por sólo un punto y coma ya no se puede compilar este código? Es el principio de nuestro camino en el mundo de la programación… *Para aprender a programar hay que picar código*.

Tras muchas horas de trabajo intentando desarrollar los programas que se plantean, donde incluso dan ganas de destrozar el ordenador, se comienza a ver la luz al final de túnel: el alumno consigue entender perfectamente esos programas de unas pocas líneas de código que parecían indescifrables hace unos días, e incluso puede proponer mejoras a dicho código. A partir de ese momento quizá comience ya a disfrutar programando. Este ciclo de aprendizaje lo hemos 'sufrido' todos y, de hecho, seguimos aprendiendo cada vez que creamos un nuevo código.

El objetivo principal que nos planteamos los autores al preparar este manual ha sido ofrecer al alumno novel una guía para el aprendizaje de los fundamentos de la programación en lenguaje C a través de prácticas. Al contrario que los clásicos boletines de prácticas formados por una simple colección de ejercicios, hemos querido desarrollar los diez capítulos que componen este libro desde un punto de vista muy didáctico: explicando detalladamente los conceptos teóricos a través de ejemplos prácticos resueltos, siguiendo cada uno de los pasos que el alumno da en su camino de iniciación en la programación.

Este manual es el resultado del trabajo conjunto que hemos realizado ambos profesores en el Centro Universitario para la Defensa (CUD) ubicado en la Academia General del Aire (AGA), centro adscrito a la Universidad Politécnica de Cartagena (UPCT), desde Septiembre de 2010. En particular,

es el primer libro docente publicado en el CUD de San Javier y complementa al libro “Informática Aplicada. Programación en Lenguaje C”.

Agradecemos especialmente a la profesora Nina Skorin-Kapov su esmerada revisión de este manual, y sus numerosas sugerencias que nos han ayudado tanto a mejorarlo en esta edición revisada.

Agradecemos recibir todas las sugerencias que puedan ayudar a mejorar las futuras ediciones de este manual. Nuestras direcciones de correo electrónico son pedroj.garcia@cud.upct.es y pedro.alcover@upct.es.

Por último, agradecemos poder trabajar al servicio de todos nuestros alumnos. A todos ellos va dedicado especialmente este libro.

Cartagena, 25 de julio de 2013

ÍNDICE GENERAL

Introducción al desarrollo de programas en lenguaje C

1

1.1. Introducción a los conceptos teóricos básicos de los ordenadores.

- **1.1.1. Arquitectura de Von Neumann.**
- **1.1.2. Microarquitectura y lenguaje máquina.**
- **1.1.3. Lenguajes de programación.**
- **1.1.4. Sistemas operativos.**

1.2. Fases de creación de un programa en C.

- **1.2.1. Definición del problema y diseño de la solución.**
- **1.2.2. Codificación.**
- **1.2.3. Compilación y ensamblaje del programa.**
- **1.2.4. Ejecución, prueba y depuración del programa.**
- **1.2.5. Recomendaciones finales.**

1.3. Primeros programas en lenguaje C.

- **1.3.1. El IDE: puesta en marcha.**
- **1.3.2. Primer programa.**
- **1.3.3. Algunas observaciones a nuestro primer programa.**
- **1.3.4. Segundo programa.**
- **1.3.5. Tercer programa.**

Esta primera práctica de carácter introductorio presenta muy brevemente los conceptos teóricos fundamentales de los ordenadores y del desarrollo de programas en lenguaje C. Estos conceptos iniciales suponen para el alumno un punto de partida que le permitirá comenzar a desarrollar sus primeros programas en C. El objetivo principal es que el alumno tenga un primer contacto con el entorno de programación que utilizará a lo largo de las distintas prácticas que se describen en este libro. Instalará el software de una aplicación para desarrollo de programas en C, y escribirá sus primeros programas.

1.1. INTRODUCCIÓN A LOS CONCEPTOS TEÓRICOS BÁSICOS DE LOS ORDENADORES

Un *ordenador* es un complejísimo y gigantesco conjunto de circuitos electrónicos y de multitud de elementos magistralmente ordenados que logran trabajar en total armonía, coordinación y sincronía, bajo el gobierno y la supervisión de lo que llamamos sistema operativo. Y es un sistema capaz de procesar información (datos) de forma automática, de acuerdo a unas pautas que se le indican previamente, dictadas mediante colecciones de sentencias o *instrucciones* que se llaman *programas*. Es un sistema que interactúa con el exterior, recibiendo los *datos* (*información*) a procesar, y mostrando también información. Un sistema que permite la inserción de nuevos programas.

Dos son, pues, los elementos fundamentales o integrantes del mundo de la programación: **datos** e **instrucciones**. Un programa puede interpretarse únicamente de acuerdo con la siguiente ecuación: **programa = datos + instrucciones**.

El objetivo de las páginas de este manual es lograr que el alumno aprenda cómo se introducen y se crean programas capaces de gestionar y procesar información. No es necesario para ello conocer a fondo el diseño y la estructura del ordenador sobre el que se desea programar. Pero sí es conveniente tener unas nociones básicas elementales de lo que llamamos la *arquitectura* y la *microarquitectura del ordenador*.

Hay diferentes arquitecturas. La más conocida y empleada en la práctica es la ***arquitectura de Von Neumann***. La característica fundamental de esta arquitectura es que el ordenador utiliza el mismo dispositivo de almacenamiento para los datos y para las instrucciones. Ese dispositivo es lo que conocemos como memoria del ordenador. Es importante conocer esta arquitectura: ayuda a comprender sobre qué estamos trabajando.

1.1.1. ARQUITECTURA DE VON NEUMANN

En la Figura 1.1 se muestra un esquema básico de la arquitectura de Von Neumann. Cinco son sus elementos básicos: (1) la Unidad de Control (UC); (2) la Unidad Aritmético Lógica (ALU, acrónimo en inglés de *Arithmetic Logic Unit*); (3) la Memoria Principal, donde se almacenan los datos y las instrucciones de los programas; (4) los dispositivos de entrada y de salida; y (5) los canales de intercomunicación entre los distintos elementos, también llamados buses: bus de datos, bus de instrucciones y bus de control, que permiten el flujo de información, de instrucciones o de señales de control a través de las partes del ordenador. Se llama Unidad Central de Proceso (CPU, acrónimo en inglés de *Central Processing Unit*) al conjunto de la UC y la ALU con sus buses de comunicación

necesarios. La CPU es un circuito integrado compuesto por miles de millones de componentes electrónicos integrados, y que se llama microprocesador.

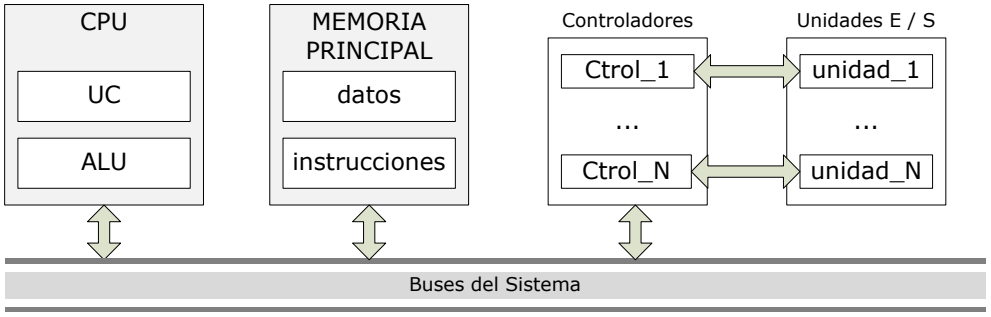


Figura 1.1. Arquitectura de Von Neumann de un ordenador

1.1.2. MICROARQUITECTURA Y LENGUAJE MÁQUINA

Cuál sea el diseño electrónico específico de cada elemento, o la tecnología concreta empleada para la fabricación de cada uno de ellos, es algo que configura el comportamiento del sistema en su conjunto. Hay diferentes tecnologías para la memoria, o para la ALU; o diferentes diseños para la CPU… Y cada diseño e implementación define la colección o repertorio de instrucciones (o microinstrucciones) que la CPU de ese sistema es capaz de entender y ejecutar. A la manera en que se implementa ese conjunto de microinstrucciones se la llama **microarquitectura**.

Cada una de las instrucciones de la colección de instrucciones queda codificada (¡como todo dentro de las tripas del ordenador!) mediante 0s y 1s: el famoso binario. Al conjunto de todas las secuencias binarias que expresan cada una de las instrucciones que la CPU es capaz de entender y ejecutar lo llamamos **lenguaje** o **código máquina**. Es posible conocer la colección completa de un ordenador y escribir programas en ese lenguaje. Pero no es una forma cómoda de programar, ni es intuitiva. Además, de la misma manera que cada modelo de ordenador está diseñado de acuerdo con su microarquitectura, y tiene por tanto su propio repertorio de instrucciones, así también, cada modelo tiene su propio lenguaje máquina. Así, un programa creado en un determinado lenguaje máquina, válido para un modelo concreto, es completamente ininteligible para otra máquina diseñada con otra microarquitectura.

1.1.3. LENGUAJES DE PROGRAMACIÓN

Con el fin de agilizar el trabajo de creación (redacción) de programas, poco a poco han surgido nuevos lenguajes, intuitivamente más sencillos de expresar y de recordar que el farragoso lenguaje máquina. Los primeros pasos en este camino de desarrollo de lenguajes fue la creación de los lenguajes ensamblador, que no eran más que una verbalización del código máquina: convertir cada cadena de ceros y unos que expresaba cada una de las microinstrucciones de la CPU en una cadena de texto, y crear luego una aplicación sencilla que, antes de mandar a ejecutar sobre el ordenador nuestro nuevo programa, traducía cada cadena de texto en su equivalente cadena binaria. Así, por ejemplo, aparecían las palabras ADD (para la instrucción de suma de números), ó SUB (para la instrucción de resta), ó MUL (para el producto), etc.

Actualmente existe gran diversidad de lenguajes y de paradigmas de programación. Nosotros vamos a centrarnos en el paradigma de la programación estructurada y en el lenguaje C. No vamos ahora a presentar el concepto de paradigma, ni el de programación estructurada: esto llega más adelante. Sí necesitamos explicar qué es un *lenguaje de programación*. Quédese por ahora con la idea intuitiva de que un programa es una secuencia ordenada de sentencias, donde ese orden de ejecución está establecido de acuerdo a unas pocas reglas fijas; y que el programa realiza un procedimiento destinado a dar solución a un problema inicial planteado.

Un **lenguaje de programación** es un lenguaje creado para expresar sentencias que puedan luego ser interpretadas y ejecutadas por una máquina. Son lenguajes artificiales, con una estructura semejante a la que tienen los lenguajes que utilizamos los humanos para nuestra comunicación. Tiene sus reglas sintácticas y semánticas, y su léxico. Y, desde luego, es un lenguaje millones de veces más sencillo que cualquiera que se emplee para la comunicación verbal entre seres humanos.

Cuando un ignorante en esos lenguajes lee un programa escrito en uno de esos lenguajes (por ejemplo, un programa escrito en C) logra comprender palabras: pero no alcanza a conocer la importancia de los elementos sintácticos, y tampoco logra comprender el significado del texto en su conjunto. Son llamados *lenguajes de alto nivel*: semejantes a los lenguajes humanos (frecuentemente al inglés) y ciertamente distantes del lenguaje de las máquinas.

Y esto es lo más extraño de todo: ¿para qué escribir programas en un lenguaje inteligible para los humanos —que somos los que los hemos de escribir— pero también ininteligible para cualquier máquina —que son las que, al fin y al cabo, deben ser capaces de comprender las instrucciones que los programas indican—? Evidentemente, esta pregunta tiene respuesta: existen aplicaciones que logran interpretar el código escrito en el lenguaje de alto nivel y ofrecer a la máquina, de forma ordenada, las microinstrucciones, expresadas de acuerdo al propio lenguaje máquina, que ésta debe ejecutar para lograr realizar lo que las sentencias del programa indican. A estas aplicaciones se las llama **Traductores**; y pueden ser **Compiladores** o **Intérpretes**. El lenguaje C no es un lenguaje interpretado, sino de compilación: es decir, la aplicación traductora crea un archivo con un conjunto ordenado de microinstrucciones que realizan exactamente aquello que el lenguaje de alto nivel expresa. Una vez el traductor ha generado el nuevo archivo, que está formado únicamente por microinstrucciones en código máquina, el código escrito en C puede guardarse para futuras revisiones: no es útil a la hora de la ejecución del programa creado, pero es imprescindible si queremos hacer modificaciones sobre el programa creado con ese código.

1.1.4. SISTEMAS OPERATIVOS

Habitualmente utilizamos la palabra **hardware** para referirnos a todos los elementos tangibles del ordenador: sus componentes eléctricos y electrónicos, el cableado, las cajas,… Pueden ser periféricos, como la impresora, o el joystick; o elementos esenciales como la placa base donde se encuentra el microprocesador, o las tarjetas y los chips de memoria; o dispositivos de uso casi obligado, como el ratón, el teclado y la pantalla.

Y cuando hablamos del **software** nos referimos a todo lo que en el ordenador es intangible: todo los elementos lógicos que dan funcionalidad al hardware: el software del sistema, como puede ser por ejemplo el sistema operativo; o las diferentes aplicaciones que se pueden ejecutar en el ordenador, como un editor de texto o un procesador gráfico o un reproductor multimedia, o un navegador.

El **sistema operativo** forma parte, por tanto, del software del sistema. Está formado por un conjunto de programas que administran los recursos del ordenador y controlan su funcionamiento. El sistema operativo administra las tareas y los usuarios, de forma que todos puedan trabajar de acuerdo a criterios previos, y de manera que todas las tareas pueden ejecutarse de forma ordenada compartiendo todas ellas y todos ellos los mismos recursos. De los sistemas operativos depende también la gestión de memoria principal del ordenador, la gestión de almacenamiento en los dispositivos masivos (discos), todo el sistema de archivos, de protección, de comunicaciones, etc.

Cuando un usuario ejecuta un programa, no lanza las instrucciones directamente hacia el hardware del sistema. Existe ese conjunto de programas intermedios, llamado sistema operativo, que hace de interfaz entre la máquina y el programa que se desea ejecutar. Y cuando un usuario lanza a imprimir varios documentos, éstos no salen amontonados e impresos en las mismas páginas. De nuevo el sistema operativo acude en nuestra ayuda y nos permite seleccionar una u otra impresora que esté conectada, establece la cola de impresión, y las prioridades de cada documento, etc.

Y cuando programemos, debemos tener presente que nuestro programa podrá acceder a los recursos del ordenador siempre a través del sistema operativo. No leeremos información del teclado directamente, sino del buffer que el sistema operativo tiene configurado para la entrada de pulsaciones de tecla. Y cuando imprimimos información, no la lanzamos contra la pantalla, sino contra el buffer que, para la gestión de esta operación de salida por pantalla, tiene creado el sistema operativo.

Hemos utilizado la palabra “buffer”, pero no hemos introducido el concepto. Un **buffer** es un espacio de memoria para almacenamiento temporal de información. Se emplea especialmente en la conexión de dispositivos, por ejemplo el ordenador y la impresora, o el disco duro o rígido, o el teclado. Ese almacenamiento de información tiene carácter temporal y transitorio: queda allí a la espera de que esa información pueda ser procesada. El sistema operativo se encarga de gestionar los diferentes buffers del ordenador. Los buffers mejoran el rendimiento del sistema en su conjunto y permiten ajustar las conexiones entre dispositivos que trabajan a distintas velocidades y con diferencias temporales. Gracias a ellos usted puede teclear aunque en ese momento el ordenador no esté “a la escucha” de sus entradas; o puede enviar varios documentos a la impresora y el sistema operativo tiene capacidad para gestionar una cola que despache los distintos documentos de forma ordenada, etc.

1.2. Fases de Creación de un Programa en C

Programar es algo más que redactar una serie de sentencias en un determinado lenguaje de programación. Es un trabajo que se desarrolla en diferentes fases: habitualmente siempre los mismos pasos, desde la definición del problema hasta la obtención del programa ejecutable correcto.

La Figura 1.2 representa esas fases en el desarrollo de un programa.

1.2.1. Definición del problema y diseño de la solución

Desarrollar un programa, incluso para cuando se trata de un programa muy sencillo como cualquiera de los que inicialmente se le van a plantear en estas páginas, es parte de un proceso que

comienza por (1) la definición del problema y (2) el posterior diseño de una solución. Estos dos pasos se abordan antes de comenzar a escribir una sola línea de código.

Si el problema es grande, o complicado, será necesario aplicar técnicas que permitan su descomposición en problemas más pequeños y sencillos y cuyas soluciones particulares puedan finalmente integrarse para alcanzar la solución del problema completo inicialmente planteado.

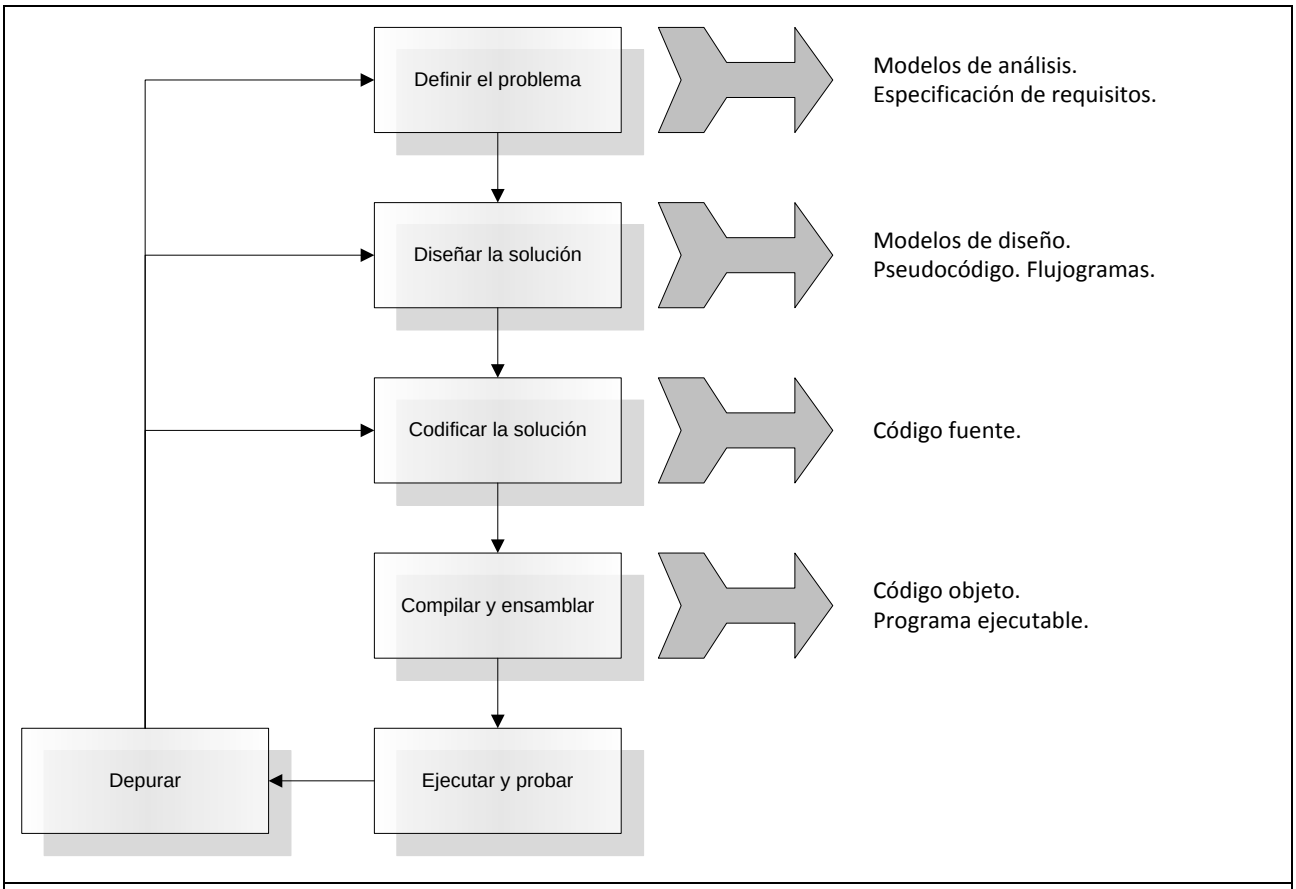


Figura 1.2. Fases del desarrollo de un programa.

Y si el problema es más sencillo, y aunque se reduzca a plantear un pequeño algoritmo, sigue siendo necesario inicialmente comprender el problema y saber cómo solventarlo antes de comenzar a redactar el programa en un lenguaje concreto.

Tenga en cuenta, sin embargo, que aunque la confección del programa es un paso posterior al de identificar una posible solución, la solución que planteemos deberá conocer los términos y las abstracciones que proporcione el lenguaje de programación en el que queremos programar. Aquí entra el juego el concepto de paradigma, que aún no explicamos.

Por ejemplo. Suponga que desea redactar un programa que tome dos números enteros y que calcule entonces su máximo común divisor. Lo primero que tendrá que hacer es conocer en qué consiste eso del máximo común divisor (mcd) y cómo se calcula. ***No sueñe con lograr programar aquello que usted mismo desconoce***. Quizá se acuerde de cómo aprendió a realizar ese cálculo del mcd, cuando era niña o niño: cómo se le enseñó a buscar la descomposición de los dos números en sus factores primos, y cómo debía entonces usted multiplicar, para el cálculo del mcd, aquellos factores comunes a ambos números, tomando de ellos las menores potencias. Por ejemplo, si debía calcular el mcd de 36 y 120, usted hacía la descomposición… $36 = 2^2 \times 3^2$ y $120 = 2^3 \times 3 \times 5$; y entonces tomaba los factores comunes a ambos números, y siempre en su menor potencia; y así calculaba y concluía: $mcd(36, 120) = 2^2 \times 3 = 12$.

Por ahora sabe cómo hacerlo. Pero no crea que eso basta para saber cómo explicárselo al ordenador en un programa. De hecho, nadie que quiera hacer un programa eficiente para el cálculo del mcd intentaría explicar tanta cosa al ordenador. Existen otras formas menos didácticas pero más eficientes. Y para el cálculo del mcd ya dejó el problema resuelto hace 25 siglos el matemático Euclides. En su momento aprenderá usted cómo implementar su algoritmo: y verá que es un algoritmo perfecto para poder convertirlo en un programa: porque cuadra perfectamente con las estructuras de programación de cualquier lenguaje: por ejemplo del lenguaje C.

Recuerde: una cosa es que usted logre saber cómo se resuelve un problema; y otra cosa distinta es que esa forma que usted conoce y comprende sea fácilmente expresable en un programa. Ante cada problema usted debe llegar a un procedimiento que, además de conducir a la solución del problema, también sea expresable en un programa atendiendo a las reglas y estructuras de un lenguaje determinado. Y eso no es trivial: requiere **oficio**: requiere, por tanto, **horas de programación**: las que usted deberá echarle para lograr alcanzar los objetivos de esta asignatura.

El modo en que se expresa una solución que luego haya que programar en un determinado lenguaje es mediante pseudocódigo o mediante flujogramas. Ya aprenderá a utilizar ambas herramientas. Puede consultar para ello el Capítulo 5 del manual de referencia de la asignatura.

1.2.2. CODIFICACIÓN

Una vez tenemos el problema resuelto, y la solución expresada en pseudocódigo o mediante un flujograma, el siguiente paso es crear el programa en un nuevo archivo o conjunto de archivos. Para ello se dispone de editores de programas, con un funcionamiento similar a un editor de texto, como por ejemplo el Word del paquete Office.

En realidad lo que se emplea son **entornos de programación**. Un entorno de programación es un conjunto de programas que son necesarios para construir, a su vez, otros programas. Un entorno de programación incluye editores de texto, compiladores, archivos con funciones, enlazadores y depuradores (ya se irá conociendo el significado de todos estos términos). Existen **entornos de programación integrados** (genéricamente llamados ***IDE***, acrónimo en inglés de *Integrated Development Environment*), de forma que en una sola aplicación quedan reunidos todos estos programas. Hay muchos IDEs disponibles para trabajar y programar con el lenguaje C, y cualquiera de ellos es, en principio, válido. Al final cada programador elige uno en el que se siente cómodo a la hora de programar. En Internet puede encontrar muchos gratuitos y de fácil instalación.

Los lenguajes ofrecen de partida grandes colecciones de funciones ya implementadas y disponibles para los programadores. También se comercializan o se distribuyen gratuitamente muchos archivos con funciones ya diseñadas. Y también el programador puede crear sus propias funciones y guardarlas en archivos para su posterior uso o para, a su vez, distribuir a terceros. La reutilización de código es un aspecto fundamental del desarrollo de software.

1.2.3. COMPILACIÓN Y ENSAMBLAJE DEL PROGRAMA

Al programa editado y que hemos escrito en C lo llamamos **código fuente**. Es muy importante saber custodiar ese código a buen recaudo: es el capital intelectual de cualquier empresa de software. Y es el valioso resultado final del trabajo de cualquier programador.

Pero, como ya hemos dicho, nuestras máquinas no son capaces de entender ese código fuente. Necesitamos traducirlo a otro lenguaje o código inteligible para la máquina. Y para ello necesitamos de un programa traductor. Ya sabe que los traductores pueden ser compiladores o intérpretes. El C no es un lenguaje interpretado; su código fuente debe ser compilado: así se genera un segundo archivo, que contiene lo que se llama código objeto, que expresa en código máquina lo que el código fuente expresaba en lenguaje C. La Figura 1.3 muestra el proceso completo de generación del programa ejecutable.

Habitualmente, el código fuente de un programa no se encuentra en un único archivo, sino que se construye de forma distribuida entre varios. Si deseamos generar un programa final ejecutable a partir de los distintos archivos fuente, deberemos primero compilar cada uno de esos archivos y generar así tantos archivos de código objeto como archivos iniciales tengamos en código fuente y, en un segundo paso, ensamblarlos para llegar al programa final ejecutable.

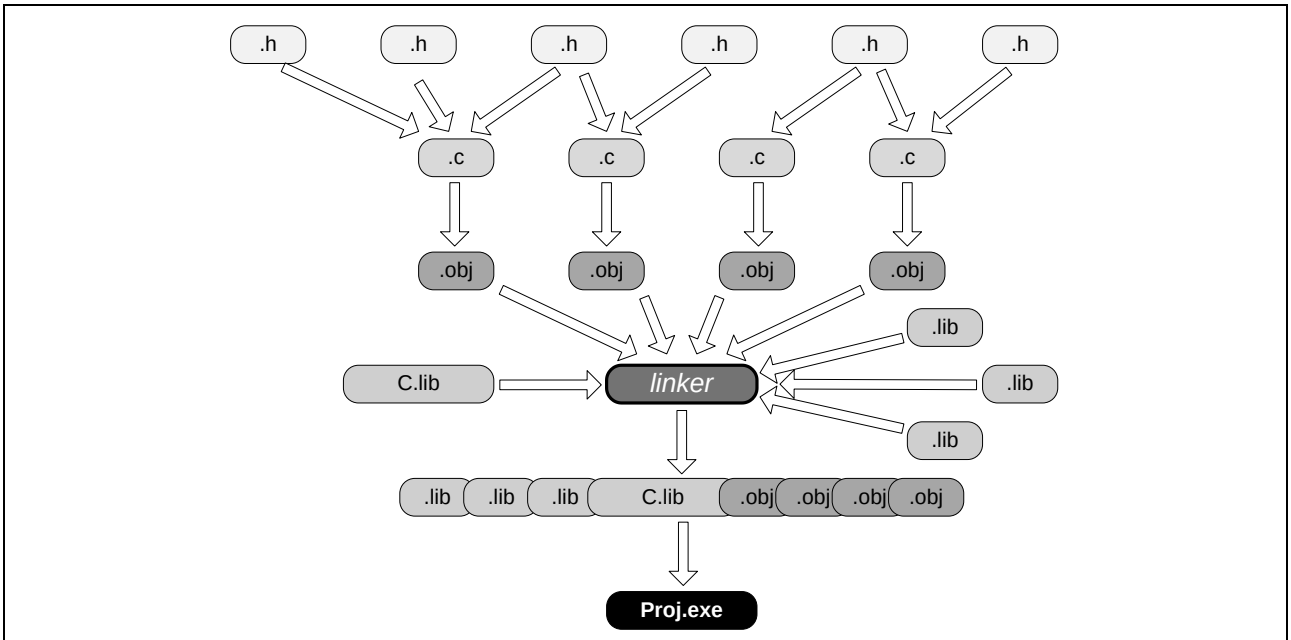


Figura 1.3. Generación de un programa ejecutable a partir del código C.

El proceso de generación de un ejecutable como el que se muestra representado en la Figura 1.3 comienza con la compilación de todos los **archivos fuente** de código C (ficheros con **extensión .c**). Cada archivo fuente puede a su vez hacer uso de otro código que ya ha sido escrito previamente por el mismo o por otro programador, o por el código de gran cantidad de funciones que el estándar de C ofrece para su uso directo. Como verá más adelante, ese código se estructura mediante ***funciones***. Este código compilado se encuentra en diferentes **archivos de librería** (**extensión .lib**): esos archivos recogen el código objeto o código máquina de distintas funciones ya creadas.

De alguna manera, cada uno de los archivos fuente que contienen el código de nuestro nuevo programa debe informar al compilador del uso que hacen de otro código ya compilado y disponible en esas librerías. Y debe además informar mínimamente del comportamiento del código que reutiliza. Para eso existen los **archivos de cabecera** (con **extensión .h**), que indican al compilador la información mínima necesaria para que logre crear el archivo con el código objeto, aunque el código fuente no disponga de todo el código que deberá ejecutar, porque ese código ya está compilado y disponible en algún archivo de librería.

Así pues, cuando un programador desea utilizar código ya creado y probado, disponible en algunas funciones, deberá indicar:

1. **Al programa**: la ubicación de los distintos archivos de cabecera donde el compilador podrá encontrar información mínima sobre las funciones que el nuevo código reutilizará. Esos archivos de cabecera recogen, para cada función, tres informaciones necesarias: el nombre de cada función disponible; la forma en que se invoca a esa función (en concreto, qué valores requerirá que se le faciliten para que logre ejecutarse correctamente); y el resultado que ofrecerá esa función (en concreto, qué valor devuelve como resultado de sus operaciones).
2. **Al compilador**: la ubicación de los distintos archivos de librería que contienen el código de todas las funciones que se van a utilizar.

Posiblemente usted no llegue —en este curso— a utilizar ninguna librería más allá de la estándar del C. Así pues, como esta librería el compilador sí la tiene bien localizada, no será necesario en ningún caso que haga indicación al compilador sobre la ubicación de las posibles librerías. Sí deberá indicar, en cada código fuente, el nombre de los archivos de biblioteca donde se encuentra la información mínima necesaria sobre las funciones que va a querer utilizar en ese código que va a escribir.

Retomemos el hilo del proceso de compilación. Tenemos nuestros archivos fuente, con sus referencias a los archivos de cabecera que sean necesarias. Al compilar, si no hay errores sintácticos en el código, se obtienen los distintos **archivos objeto** (con **extensión .obj**): tantos como archivos de código fuente tenga el proyecto que compilamos. Una vez obtenidos todos estos archivos, se procede al enlazado mediante un programa enlazador que toma todos los ficheros de código objeto y les añade el código de las funciones reutilizadas (archivo de extensión .lib, que ya está suministrado por el compilador: como ya se decía antes, al menos en sus primeros pasos, usted no utilizará otros archivos de librería distintos al estándar de C).

En resumen, y para volver a decir lo mismo de otra manera: el proceso de generación de un ejecutable a partir del código fuente consiste fundamentalmente en:

1. **Fase de COMPILACIÓN**, en la que se genera el código objeto a partir de cada uno de los archivos fuente, sin tener en cuenta el uso de las librerías externas.
2. **Fase de ENLAZAMIENTO**: donde se cogen todos los archivos de código objeto, se resuelven las dependencias con las librerías o bibliotecas externas y se genera finalmente el código ejecutable.

1.2.4. EJECUCIÓN, PRUEBA Y DEPURACIÓN DEL PROGRAMA

Dependiendo de la plataforma en la que trabajemos, para iniciar la ejecución de una aplicación que ya hemos programado y compilado basta con escribir su nombre como un comando en una consola, o con hacer doble clic en un icono.

Que nuestro programa sea capaz de ejecutarse no quiere decir que realice correctamente la tarea que tiene encomendada. En programación no es fácil acertar a la primera. Normalmente es necesario ejecutar el programa una y otra vez hasta conseguir detectar y localizar todos sus fallos, y modificar entonces el código fuente y volver a lanzar el proyecto a la fase de compilación.

Los errores de programación se clasifican en:

1. **Errores en tiempo de compilación** (errores **sintácticos**). Son detectados por el compilador cuando el código fuente no se ajusta a las reglas sintácticas del lenguaje C. Al principio usted los cometerá a miles. Le parecerá increíble que programas tan cortos como lo serán los suyos primeros acumulen tal gran cantidad de errores. Y quizá entender cada uno de ellos le resulte, al principio, arduo como descifrar la Piedra de Rosetta. Esos errores siempre se cometen pero, poco a poco, los irá reduciendo; y, desde luego, aprenderá a detectarlos con gran rapidez.
2. **Errores en tiempo de ejecución** (errores **semánticos**). Con errores de este tipo, el compilador sí compila, porque de hecho todo lo que el programa expresa cumple las reglas sintácticas. Pero no se logra que resuelva correctamente el problema, porque de hecho hay algo en el programa que está sintácticamente bien expresado pero que no realiza lo que debiera o lo que usted cree que debería hacer. Estos errores se detectan porque el programa tiene comportamientos inesperados: habitualmente, una de dos: o se llega a un resultado que no es el que se esperaba; o el programa se "cuelga". Estos errores son difíciles de localizar. El compilador no ayuda, y cuando el error da la cara no siempre somos capaces de identificar en qué parte del código está dando el fallo. En ese caso hay que recurrir al uso de técnicas de depuración, como la impresión de trazas dentro del programa, o el uso de programas depuradores. A veces la depuración de un error semántico puede ocasionar que el programador se lo replantee todo desde el principio.

1.2.5. RECOMENDACIONES FINALES

Cuide la calidad del software y su documentación. Se estima que el 60-70 % del tiempo del trabajo de un programador se ocupa en consultar y modificar código hecho por él mismo o por otra persona.

Este manual de prácticas presenta los fundamentos de programación del paradigma de programación imperativa y estructurada. Pero trabajamos únicamente los fundamentos más básicos: si debe hacer programas más extensos, con varios cientos de líneas de código, necesita aprender nuevas técnicas para el diseño de grandes programas, como especificación de requisitos, diseño arquitectónico, desarrollo de prototipos y diseño de pruebas de software. Es decir, es necesario realizar un diseño detallado del software, sus planos, antes de construirlo.

1.3. PRIMEROS PROGRAMAS EN LENGUAJE C

En la actualidad existen multitud de IDEs con grandes funcionalidades para programar en C. Casi todos son gratuitos y se encuentran fácilmente en Internet. . En la Tabla 1.1. destacamos algunos de ellos. Cada programador elige su entorno de desarrollo en función de sus necesidades y preferencias. Desde luego, deberá decidir sobre qué sistema operativo desea trabajar (Windows, Linux o Mac habitualmente) y con qué versión del IDE desea trabajar. Por su facilidad de instalación, por su sencillez de uso y porque resulta fácil hacer buen uso de sus herramientas de depuración, a los que aprenden a programar les recomendamos CodeLite o CodeBlocks.

Muchos de los IDEs ofrecen también la instalación del compilador. Tenga eso en cuenta si va a instalar más de un IDE, porque entonces debe procurar hacer una única instalación del compilador y aprender luego a configurar cada IDE para indicarle dónde se encuentra, dentro de su equipo, el programa compilador ya instalado. Tenga en cuenta que el compilador y sus archivos ocupa mucho espacio en el disco de su ordenador.

IDE	Página web	Lenguajes soportados	Sistemas operativos
Eclipse	`http://www.eclipse.org/`	C, C++, Java, JSP, Phyton, PHP	Windows, Linux, Mac OSX
Codelite	`http://www.codelite.org/`	C, C++	Windows, Linux, Mac OSX
NetBeans	`http://netbeans.org/index.html`	C, C++, Java, JSP, PHP, Groovy	Windows, Linux, Solaris, Mac OSX
XCode	`http://developer.apple.com/xcode/`	C, C++, Objective-C	Mac OSX
CodeBlocks	`http://www.codeblocks.org/`	C, C++	Windows, Linux, Mac OS X

Tabla 1.1. Relación de algunos IDEs de libre distribución, para programar en lenguaje C.

Cuando emprenda la tarea de instalar uno de esos IDEs no tenga prisa y no pinche en el primer hipervínculo que se le presente. Debe decidir bien qué programa instalador se bajará, y eso dependerá de su equipo, de lo que desee instalar realmente, de si posee permiso de superusuario dentro del equipo donde va a proceder a la instalación, etc.

No desarrollamos aquí una guía de instalación para ninguno de los IDEs. En las web de cada uno de ellos encontrará suficientes indicaciones para que, si usted las sigue detenidamente, logre terminar felizmente el proceso de instalación.

¡Ánimo! Ya puede comenzar a hacerlo… Instale una herramienta de programación. Mientras no la tenga en su equipo no merece la pena que siga leyendo este manual de prácticas…

1.3.1. EL IDE: PUESTA EN MARCHA

No sabemos qué IDE ha instalado usted finalmente. No podemos hacer aquí una guía para cada IDE. Los sugeridos en la Tabla 1.1. ofrecen en su web suficiente documentación para que usted se habitúe a moverse por la aplicación. Verá en cualquiera de ellos un programa del aspecto de un editor de texto pero con algunos botones y algunos menús de opciones desconocidos. Y es que no se trata sólo de un editor de texto: se trata de un editor de programas… Y esos programas escritos en C, luego hay que compilarlos, y probarlos, y depurarlos, y ejecutarlos.

Lo primero que debe hacer es ejecutar el programa de edición de código que acaba de instalar. Seguramente, durante la instalación se le ha sugerido la creación de un acceso directo en su escritorio y de un icono de acceso rápido en la barra de tareas. Además, se habrá creado un acceso directo en el menú inicio de Windows. Cualquiera de esas vías le sirve para arrancar el entorno de programación IDE que usted haya instalado y poder, así, comenzar a trabajar.

Una vez arrancado su IDE, le recomendamos que se arregle un poco la ventana de la aplicación. Tenga localizada la vista del espacio de trabajo (*Workspace View*, *Management*, etc.: cada IDE tendrá su propio nombre). Los IDEs suelen tener un menú View, donde puede indicar que muestre la ventana del Workspace.

Un *espacio de trabajo* (workspace) es el lugar donde se guardará su trabajo. Algunos IDEs le permitirán crear tantos workspace como usted desee, aunque sólo podrá tener uno activo cada vez. En ese caso, podrá decidir en qué directorio del disco desea crear cada uno de esos espacios. Es el caso, entre otros, de Codelite, o de Eclipse. Otros IDEs le ofrecerán un único workspace, donde usted creará todos sus proyectos. Pero usted no podrá decidir dónde se crea éste. En ese caso, deberá decidir la ubicación de cada uno de sus proyectos. Ése es el caso, por ejemplo, de CodeBlocks.

Importante: El nombre de la carpeta del Workspace debe ser sencillo, sin caracteres que sean distintos a los caracteres alfabéticos, los numéricos, o el carácter subrayado; tampoco acentúe ninguna letra. Esto no es una colección de restricciones que exige CodeLite, o CodeBlocks, u otro IDE: es conveniente que todas sus carpetas verifiquen esas mismas condiciones. También debe seguir las mismas restricciones a la hora de crear un nuevo proyecto. No haga nombres que no cumplan esas restricciones. Los espacios en blanco, los puntos, los guiones,… ¡no los use!: posiblemente no pase nada si no es cuidadoso con esto… ¡hasta que pasa!... Que pasa.

IMPORTANTE: Sí: lo acabamos de decir. Pero es necesario destacarlo: es asombroso cómo los alumnos caen en esto…

Los nombres de los directorios (las carpetas) donde usted vaya a crear sus espacios de trabajo y sus proyectos no deben tener espacios en blanco, ni caracteres acentuados, ni la letra 'Ñ'. No guarde su código en una carpeta dentro de "***Mis Documentos***", porque su nombre tiene un espacio en blanco. No lo guarde en la carpeta "***Informática***", porque su nombre tiene un carácter acentuado. Tampoco lo guarde en la carpeta "***peñazoInformatica***" (nombres mucho peores hemos visto entre nuestros alumnos) porque la letra 'ñ' también da problemas.

Deberá dedicar un tiempo a moverse por la nueva herramienta. Deberá aprender a abrir un workspace, a cerrarlo, a pasar de uno a otro… Deberá aprender a crear, dentro de un workspace, un nuevo proyecto, o eliminarlo. Al menos debe aprender a manejar esas dos cosas.

1.3.2. PRIMER PROGRAMA

En su *Workspace* debe crear ahora un proyecto. Al indicar al IDE que desea crear un nuevo proyecto, éste le ofrecerá distintas posibilidades, según el tipo de proyecto que quiera crear. Usted debe elegir uno de tipo Aplicación para consola. La herramienta de programación le solicitará alguna información adicional, como el nombre y la ubicación en disco donde desea almacenar el proyecto. También deberá indicar que su proyecto es para un programa en C (no C++).

Le aparecerá, anidado en la carpeta del espacio de trabajo, otra carpeta con el nombre de su proyecto. Y dentro de ella una nueva carpeta donde se guarda el código fuente creado por defecto, en un archivo posiblemente llamado ***main.c*** (todas esas especificaciones pueden variar de un IDE a otro). Si abre el archivo de extensión .c (doble click sobre él) podrá ver el código que se ha creado y que será igual o algo parecido a lo que se muestra en el Código 1.1. (la numeración a la izquierda de algunas de las líneas de código y mostrada entre los caracteres "/*" y "*/" han sido introducidos

por los autores y no forma parte del código útil del programa; no le deben aparecer a usted en el documento *main.c* que le ha generado el editor de programas).

Código 1.1.

```
/*01*/    #include <stdio.h>
/*01*/    #include <stdlib.h>

/*02*/    int main(int argc, char **argv)
/*03*/    {
/*04*/          printf("hello world\n");
/*05*/          return 0;
/*06*/    }
```

Es un programa muy sencillo. Imprime una cadena de texto por pantalla (`"hello world"`: vea la línea `/*04*/`) y termina. Desde luego, puede cambiar el texto de esa cadena. Hágalo: por ejemplo, ponga `"Este es mi primer programa en C... "`. El código modificado quedara cómo el recogido en el Código 1.2.

Código 1.2.

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

int main(int argc, char **argv)
{
      printf("Este es mi primer programa en C...\n");
      return 0;
}
```

Así, habrá creado su primer y propio programa. Después de esto, sólo le queda seguir aprendiendo a programar…

Si abre, a través del explorador de archivos de su ordenador, la carpeta del espacio de trabajo, verá que dentro de ella aparece una nueva carpeta: con el nombre que usted le haya querido dar al proyecto. Y dentro de esa nueva carpeta aparecerá el documento con nombre main de extensión .c, y un archivo de tipo Project. El primero de los dos es el archivo de texto donde queda recogido el código fuente. Puede, si quiere, abrirlo con el bloc de notas o con el WordPad y verá el texto introducido. Pero si lo hace ahora, verá el código del programa primero, antes de la modificación: y es que aún no ha guardado el nuevo código.

Vuelva a la ventana de su aplicación IDE. Ahora vamos a realizar tres operaciones:

1. Primera, GUARDAR el nuevo código. Para ello tiene múltiples caminos, como en cualquier aplicación de ventanas.

2. Segunda, COMPILAR el programa escrito y ENLAZAR para construir el ejecutable. Ya ha quedado explicado qué es eso de compilar el programa. Para hacer esa operación, de nuevo tiene varias vías dentro de las opciones de su IDE.

   En cuanto haya ejecutado el comando de construcción del proyecto podrá consultar, en el panel de mensajes de salida, el resultado de su compilación. Allí verá usted algún mensaje indicando que el proceso se ha finalizado correctamente, o señalando las líneas del código donde usted tiene algún error sintáctico.

   Si ha conseguido compilar y construir correctamente, habrá creado entonces un archivo ejecutable. Lo puede encontrar en la carpeta del espacio de trabajo, en la carpeta del proyecto creado. Si lo ejecuta desde esa carpeta es posible que vea apenas nada. Porque la aplicación se abrirá, ejecutará la línea de texto y se cerrará al instante. Puede arreglar eso insertando una nueva línea, tal y como se muestra en el Código 1.3. (línea marcada con /*01*/).

Código 1.3.

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

     int main(int argc, char **argv)
     {
           printf("Este es mi primer programa en C...\n");
/*01*/     system("pause");       // Válido sólo para Windows
           return 0;
     }
```

   Guarde. Compile… Y ahora ejecute el programa directamente haciendo doble click sobre el icono del ejecutable que ha creado al compilar. Cuando quiera terminar la ejecución del programa pulse una tecla: la ventana de ejecución se cerrará.

3. Tercera, EJECUTAR la aplicación desde el propio IDE. Para ello debe buscar el botón, o la opción de menú, que indique algo parecido a “Run”. También existe en algunos IDEs la opción combinada de “Build + Run”: esta opción compila el programa y si logra terminar el proceso, entonces lanza la ejecución del programa recién creado.

   Al ejecutar la aplicación se abrirá una ventana de consola, y en ella aparecerá el texto que venía recogido dentro de la función `printf()` que habíamos escrito en nuestro código. Ahora aparece también un texto que le sugiere que pulse cualquier tecla antes de terminar la ejecución de la aplicación. Si así lo hace, se cerrará la ventana de comandos. Y tenga en cuenta que mientras no cierre esa ventana, no podrá lanzar a ejecutar desde el IDE ningún otro programa.

1.3.3. ALGUNAS OBSERVACIONES A NUESTRO PRIMER PROGRAMA

Vamos a ir viendo el código de este primer programa. No es indescifrable, y con un poco de buena voluntad se puede lograr comprender todo o casi todo. Las referencias a las líneas están tomadas del Código 1.1.

1. Lo primero que encontramos en nuestro programa es la línea **`#include`** `<stdio.h>` (vea línea `/*01*/`). Esta palabra, **`include`**, no es una palabra de C. Todas las veces que vea una palabra precedida del carácter almohadilla (#) estará ante lo que se llama una **directiva de preprocesador**. No hay muchas, sino algo más de una docena. Y usted en este curso introductorio solo va a conocer dos: la que ahora nos ocupa, **`include`**, y otra. Las directivas envían mensajes al compilador, y el compilador las ejecuta en cuanto se las encuentra. La directiva **`include`** informa al compilador que, al código escrito en nuestro archivo .c hay que insertarle el del archivo de cabecera que se indica entre ángulos.

   El archivo `stdio.h` es un **archivo de cabecera**. Recoge la información mínima necesaria para poder usar en nuestro programa diferentes funciones de entrada y salida estándar por consola (ratón, teclado, pantalla) u otras vías: archivos, la red, etc.

   Con esta directiva, por tanto, informamos al compilador que nuestro programa hará uso de algunas funciones, de entrada y salida, que ya están creadas y compiladas. Usted puede considerar —por ahora— que una función es como un trozo de código ya compilado que le resuelve tareas concretas. Ese código no está en nuestro documento programa *main.c*. Pero ahora, con la directivas **`include`**, que implican la inclusión de un determinado archivo de cabecera, nuestro programa ofrece la información mínima necesaria para que el compilador pueda verificar si, en nuestro código, hemos hecho un uso correcto de esas funciones que ya están compiladas y que, más tarde, el linkador (o enlazador) enlazará con nuestro código para crear, por fin, el programa ejecutable.

   También puede aparecer (eso depende, de nuevo, del IDE con el que usted está trabajando) una línea con la directiva **`#include`** `<stdlib.h>`. El archivo `stdlib.h` es otro archivo de cabecera. Ya lo ira conociendo…

2. La siguiente línea (marcada con el número `/*02*/`) es la cabecera de la función principal: **`int`** `main(`**`int`** `argc, `**`char`** `**argv)`. No ha de entender ahora mismo todo lo que allí se pone. De hecho, por ahora bastaría que la cabecera tuviera la siguiente apariencia: **`int`** `main(`**`void`**`)`. Con el tiempo se dará cuenta que todo programa ejecutable debe tener su propia función main; que todos sus proyectos tendrán, de hecho, una función principal; y que no puede crear un proyecto con más de una función con ese nombre.

   El código que recoge todas las sentencias que debe ejecutar la función principal va delimitado por dos caracteres de apertura y cierre de llave (líneas de código `/*03*/` y `/*06*/`): {}. Desde luego, el código de esta primera función `main()` es muy simple: una línea (línea `/*04*/`) en la que se indica que el programa debe mostrar, a través de la consola de ejecución un determinado texto que viene recogido entre comillas dobles. Es sencillo entender que eso es lo que manda hacer la función `printf()`. El uso de esta función está ampliamente desarrollado en el Capítulo 8 del manual de referencia de la asignatura y aprenderá a utilizarla correctamente en la práctica del Capítulo 2 de este libro.

3. En la última línea (`/*05*/`) de la función principal encontramos una sentencia con la palabra reservada **`return`**. Esta línea indica el final de la ejecución de la función main y devuelve el control al sistema operativo. No es ahora el momento de comprender del todo su significado. Por ahora, simplemente inserte esta línea al final de cada una de sus funciones `main()`. Tiempo habrá de llegar a todo.

1.3.4. SEGUNDO PROGRAMA

No se trata de que ahora comprenda todo lo que va a hacer: ya llegará a aprender y comprender cada detalle de la programación. Ahora cree un nuevo proyecto (llámelo como quiera) y elimine el código que aparecerá en el archivo *main.c* que se creará por defecto. Escriba en su lugar lo sugerido en el Código 1.4.

Código 1.4.

```
#include <stdio.h>

int main(void)
{
	short a = 0x7FFA;
	do
	{
		a++;
		printf("a --> %+6hd (%04hX)", a , a);
		getchar();
	}while(a != (signed short)0x8005);
	return 0;
}
// FIN.
```

No se trata, repetimos, de que lo entienda ahora. Se trata de que escriba código. Es la única vez que le recomendamos copiar sin entender (los números del margen izquierdo, introducidos entre los caracteres "/*" y "*/": no forman parte del código, y son sólo guía para poder, luego, hacer referencia en este manual a algunas líneas de código). Pero siempre hay una primera vez en la que hay que andar completamente guiados de la mano.

Guarde… Compile… Ejecute… Tendrá en pantalla la siguiente salida:

```
a --> +32763 (7FFB)
```

Si va pulsando la tecla *Intro* irán apareciendo más valores…

```
a --> +32763 (7FFB)
a --> +32764 (7FFC)
a --> +32765 (7FFD)
a --> +32766 (7FFE)
a --> +32767 (7FFF)
```

Como puede ver, cada vez que pulsa la tecla de Intro se incrementa en 1 el valor de a que se muestra en pantalla. La a es una variable (ya entenderá más adelante qué es una variable…) de 16 bits (2 bytes) que codifica los enteros con signo. Su dominio va, por tanto, desde el $-2^{-15} = -32.768$ hasta el $+2^{15} - 1 = +32.767$.

¿Cuál es el siguiente valor que aparecerá en pantalla? Pulse la tecla *Intro* y lo verá. En el Capítulo 3 del manual de referencia tiene una explicación del comportamiento de esta variable.

1.3.5. TERCER PROGRAMA

Ya le hemos dicho que no se pretende ahora que lo entienda todo. Tenemos por delante la tarea de ayudarle a aprender un lenguaje de programación. Y partimos del supuesto de que usted no sabe ni de lenguajes ni de programación. Y es complicado ir poco a poco mostrándole las numerosas piezas de este puzle, porque por más que vea no es fácil contextualizarlas. En esta primera práctica no se trata de que usted entienda todo lo que se le muestra, sino simplemente que vea y haga lo mismo que se le muestra hecho. Un paseo por casi todo aún sabiendo que no se entiende casi nada. Paciencia.

Todo programa ejecutable tiene una función principal. Esta función se llama *main*, y usted ya ha creado varias. Como ya le hemos dicho, un programa (identifique usted por ahora programa con proyecto) no puede tener dos funciones principales: si lo intenta, el compilador le informará del error y no logrará crear el archivo con el código objeto compilado.

Si se desean crear varios programas, entonces deberemos crear varios proyectos. Un espacio de trabajo puede tener tantos proyectos como desee. Pero un proyecto sólo tiene una función principal.

Hagamos otro programa. Creamos un nuevo proyecto y editamos el archivo *main.c*. Y escribimos el código del Código 1.5.

Código 1.5.

```
            #include <stdio.h>
/*01*/      #include <math.h>

            /* Programa que recibe del usuario un valor
             * en coma flotante y calcula y muestra por
             * pantalla el valor de su raíz cuadrada.
            */

            int main(void)
            {
/*02*/            double a;   // Variable para valor de entrada.
/*03*/            double r;   // Variable par el valor de la raíz.

            // Primer paso: entrada del valor de a...
                  printf("Valor de a ... ");
                  scanf("%lf", &a);

            // Segundo paso: calculo de la raíz de a...
/*04*/            r = sqrt(a);

            // Tercer paso: mostrar por pantalla el valor de la raiz...
                  printf("La raiz cuadrada de %lf es %lf\n", a, r);
                  return 0;
            }
```

Este programa solicita del usuario un valor en coma flotante (un valor, diríamos, Real o Racional) y devuelve la raíz cuadrada de ese valor introducido que luego muestra por pantalla.

En este programa hay varias novedades entre las que conviene señalar las dos siguientes:

1. **Comentarios**. Un comentario es cualquier cadena de texto que el compilador ignora. Son muy útiles y convenientes, sobre todo cuando el código redactado va tomando cierta extensión. Tenga por cierto que el código que usted redacte en una intensa y fructífera jornada de trabajo le resultará claro y nítido a la media hora de haber terminado la redacción… y oscuro y espeso, hasta casi decir indescifrable, a los pocos días de esa buena jornada de trabajo. No escriba programas pensando únicamente en el compilador: piense en usted, programador, y en quienes quizá más tarde deberán trabajar sobre su código.

IMPORTANTE: Es muy mala práctica de programación no insertar comentarios. Es el camino más corto para lograr que su código sea inútil. Con frecuencia resulta más cómodo reinventar de nuevo un código que intentar descifrar un código ya olvidado y no comentado.

   Hay dos formas de introducir comentarios en un programa redactado en C: la primera es insertar la cadena de texto barra asterisco: `"/*"`. A partir de ese momento, todo lo que el compilador encuentre en el archivo será ignorado… hasta que aparezca la cadena asterisco barra: `"*/"` (puede ver varias líneas de comentario antes del encabezado de la función `main`). La segunda forma es insertar dos barras seguidas: `'//'` (por ejemplo, líneas `/*02*/` y `/*03*/` del Código 1.5.). A partir de ellas, y hasta que no haya un salto de línea, el texto que siga será considerado comentario, y el compilador lo ignorará. En estas dos líneas indicadas, el código anterior a las dos barras sí es válido para el compilador; el comentario no comienza sino más allá, a la derecha de las dos barras.

2. El uso de la **función** `sqrt` (línea `/*04*/` del Código 1.5.). Esta función realiza el cálculo de la raíz cuadrada de un número que recibe entre paréntesis. Para poder hacer uso de ella, se debe incluir un nuevo archivo de cabecera con nombre de cabecera `math.h` (línea `/*01*/`).

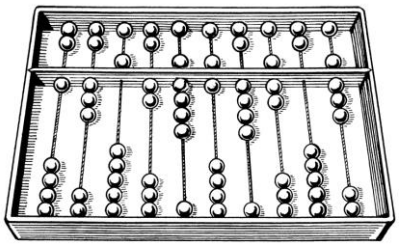
   Uno puede preguntarse cuántas funciones y cuántos archivos de cabecera están disponibles. La respuesta es que muchísimas y muchísimos. Pero no desespere. Poco a poco las irá conociendo. Es mejor conocer las que vaya a necesitar, y no aprenderlas toda para luego usar unas pocas.

Conceptos básicos para la entrada y salida por consola

2

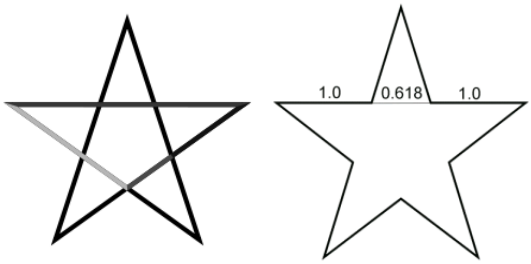
El objetivo de esta práctica es ofrecer al alumno un primer contacto con el lenguaje de programación en C aprendiendo a utilizar las funciones disponibles para entrada/salida por pantalla y conocer los distintos tipos de datos. La práctica comienza con dos ejemplos introductorios. A continuación, el alumno deberá ser capaz de realizar una serie de proyectos[1]:

1. Una calculadora, denominada **ABACO**, para realizar operaciones básicas (suma, resta, producto y división) con dos números introducidos por teclado.



2. Un display digital muestra la temperatura interior de un almacén de explosivos en grados Fahrenheit. El programa del proyecto denominado **TEMPERATURA** ha de permitir realizar la conversión de grados Fahrenheit a Celsius.

3. Análisis del número **ÁUREO**. La relación que define a este número irracional se encuentra en muchas figuras geométricas, infinidad de elementos de la naturaleza y múltiples diseños del hombre. Así por ejemplo, la proporción áurea aparece en la estrella de cinco puntas, que es una figura geométrica muy utilizada en las divisas y distintivos de las Fuerzas Armadas. El proyecto AUREO calculará el valor de este número y lo mostrará por pantalla en diferentes formatos.

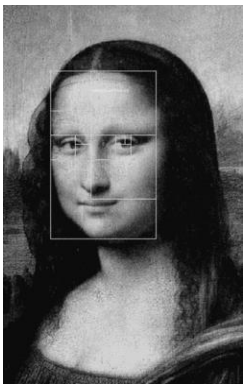


Estrella de cinco puntas

Estado Mayor del Ejército del Aire

1 En cada proyecto se sugieren sucesivas mejoras que también aumentan progresivamente la complejidad del código a implementar.

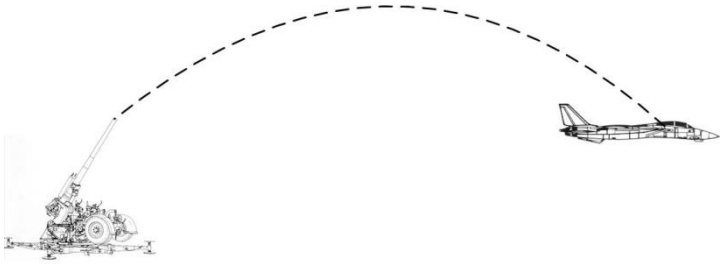
Como decíamos, continuamente se puede encontrar esta relación en la naturaleza. Quizá por eso ya los arquitectos antiguos utilizaban el número áureo como proporción de armonía. Leonardo Da Vinci lo utilizó para el rostro de la Gioconda. Y los griegos para el diseño del Partenón.



4. La aplicación desarrollada con el proyecto **OCULTO** permitirá realizar las operaciones binarias necesarias para extraer el número secreto oculto entre algunos de los bits de un determinado número entero de 4 bytes.

Después de estos cuatro proyectos propuestos, se incluye además un ejercicio complementario.

5. Por último, diseñaremos, en el proyecto **TIRO**, un programa que permita estudiar el fenómeno físico asociado al tiro parabólico y su aplicación a un *cañón antiaéreo*.



2.1. EJEMPLOS SENCILLOS RESUELTOS

Antes de intentar resolver los ejercicios y problemas propuestos en esta práctica, parece conveniente y casi necesario proponer algunos ejercicios guiados que permitan al alumno introducirse en el uso de las funciones de entrada y salida de datos por consola.

Ejemplo 2.1. USOPRINTF

Creamos un nuevo proyecto denominado **USOPRINTF**. Tal y como se ha analizado en la práctica anterior, al crear un nuevo proyecto el IDE (Codelite, CodeBlocks, …) genera, automáticamente, un código similar al mostrado en Código 2.1:

Código 2.1.

```
#include <stdio.h>

int main(int argc, char **argv)
{
      printf("hello world\n");
      return 0;
}
```

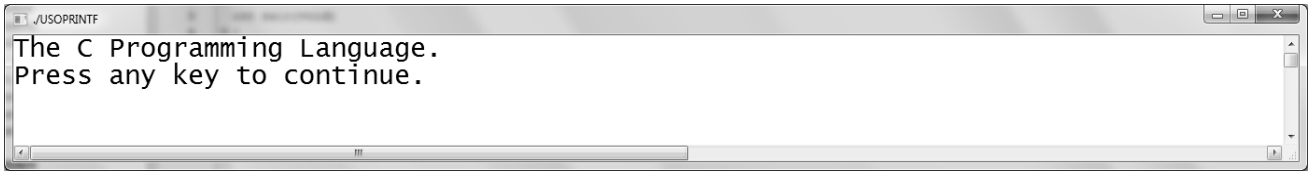
Como ya hicimos en la práctica anterior, modificamos el código del programa cambiando el texto a mostrar por pantalla, como se puede ver en Código 2.2.

Código 2.2.

```
#include <stdio.h>

int main(int argc, char **argv)
{
      printf("The C Programming Language.");
      return 0;
}
```

y el resultado del mismo será mostrar por pantalla el texto recogido entre comillas dobles y entre paréntesis en la llamada a la función `printf()`.



Vemos, pues, que la función `printf()` nos permite mostrar información por pantalla. Como toda función, `printf` lleva después de su nombre unos paréntesis, `()`. Entre comillas dobles, y dentro de los paréntesis, `printf()` recibe el texto que se desea mostrar por pantalla.

Supongamos ahora que queremos, además, escribir el texto `"Edition 1"`. En este caso, lo primero que pensamos es algo parecido a lo recogido en Código 2.3.

Código 2.3.

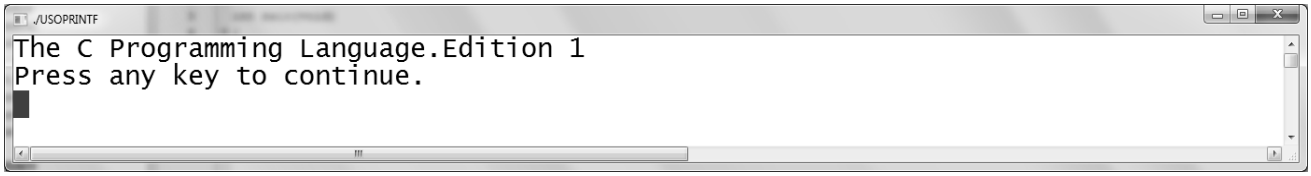
```
#include <stdio.h>
int main(void)
{
      printf("The C Programming Language.Edition 1");
      return 0;
}
```

Aunque también podemos hacerlo como se recoge en Código 2.4.

Código 2.4.

```
#include <stdio.h>
int main(void)
{
      printf("The C Programming Language.");
      printf("Edition 1");
      return 0;
}
```

Y en ambos casos obtenemos un resultado idéntico por pantalla:



Esto es debido a que en la segunda llamada a la función `printf()`, ésta continúa su ejecución "escribiendo" allí donde el cursor se quedó la última vez que se ejecutó la función, es decir, en la siguiente posición de la ventana de ejecución más allá del `'.'` después de `"Language."`.

Si se desea introducir un salto de línea (como cuando, en un editor de texto, pulsamos la tecla *Enter*) debemos hacer uso de los *caracteres de control.* Todos ellos comienzan con la barra invertida (`'\'`) seguida de una letra: una letra distinta específica para cada carácter de control. Así, por ejemplo, `'\n'` representa el carácter de control que significa salto de línea o nueva línea. Cada vez que el ordenador encuentre `'\n'`, en la cadena de texto a imprimir mediante `printf()`, se va a introducir un nuevo salto de línea. De esta forma si se ejecuta el código recogido en Código 2.5, donde en la línea `/*01*/` hemos introducido el carácter de control nueva línea, la salida por pantalla ocupará ahora 2 líneas de texto:

Código 2.5.

```
     #include <stdio.h>
     int main(void)
     {
/*01*/     printf("The C Programming Language.\n");
           printf("Edition 1");
           return 0;
     }
```

Salto de línea

Una vez que sabemos imprimir un texto predeterminado, vamos a ver el procedimiento para mostrar por pantalla el valor de una variable de nuestro programa. Supongamos que queremos mostrar por pantalla el valor de una determinada variable entera, `x`, de tipo **`int`**. Para ello, debemos hacer uso del carácter reservado `'%'`. Este carácter, junto con otros que le siguen, indican que en esa posición de la cadena de texto que se ha de mostrar por pantalla mediante la función `printf()` se quiere insertar, en un formato de representación concreto y determinado, el valor de una variable. Por esta razón, al carácter `'%'` junto con los caracteres que le siguen, se les denomina *especificadores de formato*. El especificador de formato utilizado depende de dos factores: (1) del tipo de dato de la variable cuyo valor se desea insertar en una determinada posición de la cadena de texto; y (2) de la forma en que deba ir impreso ese valor. En el primer caso se pueden incluir modificadores de tipo; en el segundo caso, se incluyen los llamados anchos de campo y precisión y las banderas de alineamiento de texto. Para más detalles se recomienda al lector consultar el capítulo 8 del manual de teoría de referencia (Apartado "Salida de datos. La función `printf()`") Por ejemplo, en nuestro caso queremos imprimir en formato decimal (base 10) el valor entero almacenado en la variable `x`, por lo que se debe utilizar la cadena de especificación de formato `"%d"`. Deberemos situar los caracteres `%d` en la posición del texto donde queramos que se muestre el valor de `x`. Después de las comillas dobles de cierre se escribe una coma y a continuación el

nombre de la variable. Así, el código C necesario es el recogido en Código 2.6. Así puede verse en la línea /*01*/ de ese código. El resultado que por pantalla ofrece la ejecución de Código 2.6. es:

Código 2.6.

```
     #include <stdio.h>
     int main(void)
     {
          int x=1;

          printf("The C Programming Language.\n");
/*01*/     printf("Edition %d",x);

          return 0;
     }
```

Variable a imprimir su valor

Especificador de formato

En el Código 2.7. se muestra un ejemplo con dos variables. En la línea marcada (/*01*/) vemos que en la cadena de texto a imprimir que recibe la función printf() se han incluido dos especificaciones de formato, ambos para variables enteras y para ser mostradas en formato decimal (base 10); la primera variable (x), que es la primera que queda indicada después de la cadena de texto y después de la coma, se imprimirá de acuerdo al primer especificador de formato; la segunda (y) se mostrará donde se encuentra el segundo especificador de formato.

Código 2.7.

```
     #include <stdio.h>
     int main(void)
     {
          int x = 1, y = 1978;

          printf("The C Programming Language.\n");
/*01*/     printf("Edition %d - Year %d", x, y);

          return 0;
     }
```

Esta será la salida que, por pantalla, tendremos cuando ejecutemos el programa de Código 2.7.

Las Tablas 1 y 2 resumen, respectivamente, el conjunto de caracteres de control y especificadores de formato disponibles en el lenguaje C.

`\a`	Carácter sonido. Emite un pitido breve.
`\v`	Tabulador vertical.
`\0`	Carácter nulo.
`\n`	Nueva línea.
`\t`	Tabulador horizontal.
`\b`	Retroceder un carácter.
`\r`	Retorno de carro.
`\f`	Salto de página.
`\'`	Imprime la comilla simple.
`\"`	Imprime la comilla doble.
`\\`	Imprime la barra invertida `'\'`.
`\xdd`	dd es el código ASCII, en hexadecimal, del carácter que se desea imprimir.

Tabla 1. Caracteres de control para la función `printf()`.

`%d`	Entero con signo, en base decimal.
`%i`	Entero con signo, en base decimal.
`%o`	Entero (con o sin signo) codificado en base octal.
`%u`	Entero sin signo, en base decimal.
`%x`	Entero (con o sin signo) codificado en base hexadecimal, usando letras minúsculas. Codificación interna de los enteros.
`%X`	Entero (con o sin signo) codificado en base hexadecimal, usando letras mayúsculas. Codificación interna de los enteros.
`%f`	Número real con signo.
`%e`	Número real con signo en formato científico, con el exponente ‘e’ en minúscula.
`%E`	Número real con signo en formato científico, con el exponente ‘e’ en mayúscula.
`%g`	Número real con signo, a elegir entre formato e ó f según cuál sea más corto.
`%G`	Número real con signo, a elegir entre formato E ó f según cuál sea más corto.
`%c`	Un carácter. El carácter cuyo ASCII corresponda con el valor a imprimir.
`%s`	Cadena de caracteres.
`%p`	Dirección de memoria.
`%n`	No lo explicamos aquí ahora.
`%%`	Si el carácter % no va seguido de nada, entonces se imprime el carácter sin más.

Tabla 2. Especificadores de tipo de dato en la función `printf()`.

Los especificadores de formato presentan múltiples opciones para la representación de información. A modo de resumen, se recoge en Código 2.8. un ejemplo auto-explicativo. Para más detalles, consultar el manual de referencia de la asignatura.

Escriba y compile en su editor de programas el código propuesto en Código 2.8. En él se muestran distintos ejemplos de uso de la función `printf()`. Analice la salida que, por pantalla, ofrecerá el programa cuando lo ejecute.

Código 2.8.

```
#include <stdio.h>
int main(void) {
     int x = 45;
     double y = 23.354;
     char z = 'h';
/*
*    utilizamos barras inclinadas (/) para ver claramente
*    la anchura del campo de caracteres
*/
     printf("Voy a escribir /45/ con el formato %%d: /%d/\n", x);
     printf("Voy a escribir /45/ con el formato %%1d: /%1d/\n", x);
     printf("Voy a escribir /45/ con el formato %%10d: /%10d/\n", x);
     printf("\n");
     printf("Voy a escribir /23.354/ con el formato %%f:/%f/\n", y);
     printf("Voy a escribir /23.354/ con el formato %%.3f: /%.3f/\n", y);
     printf("Voy a escribir /23.354/ con el formato %% .2f: /% .2f/\n", y);
     printf("Voy a escribir /-23.354/ con el formato %% .2f: /% .2f/\n", -y);
     printf("Voy a escribir /23.354/ con el formato %%5.1f: /%5.1f/\n", y);
     printf("Voy a escribir /23.354/ con el formato %%10.3f:/%10.3f/\n", y);
     printf("Voy a escribir /23.354/ con el formato %%-10.3f:/%-10.3f/\n", y);
     printf("Voy a escribir /23.354/ con el formato %%08.2f: /%08.2f/\n", y);
     printf("\n");
     printf("Voy a escribir /h/ con el formato %%c: /%c/\n", z);
     printf("Voy a escribir el valor ASCII de /h/ con %%d: /%d/\n", z);
     printf("%s /%d/\n", "Voy a escribir el valor ASCII de /h/ %%d:", z);

     return 0;
}
```

***NOTA*:** Para imprimir el carácter `'%'` es necesario utilizar `'%%'`.

Ejemplo 2.2. USOSCANF

Una vez hemos aprendido a mostrar información por pantalla, necesitamos conocer también cómo leer la información introducida por el usuario a través del teclado. Para ello, el lenguaje de programación C dispone, entre otras, de la función `scanf()`.

Cree un nuevo proyecto denominado **USOSCANF** con el código recogido en Código 2.9., que permite solicitar al usuario la introducción de un valor (entero en el ejemplo) que se guardará o almacenará en (suele decirse que ese valor se asignará a) una variable (variable x en el ejemplo), y después lo muestra por pantalla.

Código 2.9.

```
     #include <stdio.h>
     int main(void){
           int x;

           printf("Introduzca un caracter....");
/*01*/     scanf("%d", &x);
           printf("Entero: %d", x);

           return 0;
     }
```

En el anterior código, si analizamos la sentencia donde aparece `scanf()` (línea `/*01*/`) podemos ver su sintaxis básica:

`scanf(tipo, &var);`

- `tipo`: Tipo de dato de la variable que almacena el valor introducido por el usuario desde el teclado. Primero – antes de la coma – aparece, entre comillas dobles, el especificador de formato del dato introducido por teclado. En este caso es un número entero – en formato decimal – y se emplea `%d`. Los especificadores de formato siguen las mismas pautas que hemos mostrado para la función `printf()`.
- Ampersand (&). Es el operador dirección. Cuando en una expresión dentro de un programa en C aparece el nombre de una variable, esa expresión se evalúa tomando el valor que esa variable tiene en ese momento. Si el nombre de esa variable va precedido por este operador dirección (&) entonces en esa expresión ya no se toma el valor de la variable, sino su dirección en memoria: el código de su ubicación, que para muchos compiladores tiene formato de entero de 4 bytes en ordenadores con arquitectura de 32 bits, aunque pueden ser también de 8 bytes (para arquitecturas de 64 bits). En la función `scanf()` debemos indicar la ubicación o dirección de memoria de la variable donde se almacenará el dato. Al contrario que `printf()`, la función `scanf()` espera como segundo parámetro, pues, el lugar donde se aloja la variable y no el nombre (=valor) de la misma.
- `var`: variable donde queremos que quede asignado el valor introducido por teclado.

En el Código 2.9., la ejecución de la sentencia marcada con `/*01*/` correspondiente a la función `scanf()` provoca que la ventana de la consola quede a la espera de que el usuario introduzca el valor de un dato entero en formato decimal. Para introducirlo, el usuario deberá teclear ese valor y terminar pulsando la tecla *Intro*. Este valor quedará almacenado en la variable `x`.

A continuación se muestra el resultado obtenido al introducir el número 3 por teclado:

NOTA: La lectura de caracteres con la función `scanf()`, utilizando el especificador `%c`, puede ser problemática. Una forma recomendada de uso para este tipo de dato **`char`** es la siguiente: dejar un espacio en blanco después de las primeras comillas dobles y antes del carácter reservado `%`. Esto es:

```
char  y;
scanf(" %c", &y); //Hay un espacio antes de %
```

Con ello le indicamos a la función `scanf()` que ignore como entrada los caracteres en blanco, los tabuladores o los caracteres de nueva línea. Sin escribir el espacio se obtiene, generalmente, un correcto funcionamiento con valores numéricos y los problemas surgen al leer caracteres. De todas formas, recomendamos insertar siempre ese espacio en blanco entre las comillas de apertura y el carácter tanto por ciento.

Otra forma de tomar entrada de caracteres por teclado es utilizando la función `getchar()`.

```
y = getchar();
```

No obstante, en general es muy recomendable, cuando se leen caracteres por teclado, hacer antes uso de `fflush(stdin)` que permite vaciar (limpiar) el buffer del teclado.

```
fflush(stdin); scanf(" %c", &y);
fflush(stdin); y = getchar();
```

No es objeto de la práctica profundizar en las particularidades y dificultades de la función `scanf()`. Para más detalles consultar el manual de la asignatura.

Para finalizar este ejemplo, escriba el código sugerido en Código 2.10. y compile el proyecto. Tenga en cuenta, como ya ha quedado explicado más arriba, que al invocar a la función scanf() (vea `/*01*/`, `/*02*/` y `/*03*/`) es muy recomendable dejar siempre un espacio en blanco entre las comillas dobles de inicio de la cadena de texto del primer parámetro de la función y el tanto por ciento del especificador de formato.

¿Comprende todo lo que se muestra por pantalla? En cualquier caso, no se preocupe: vamos aprendiendo paso a paso. Si tiene alguna duda, consulte con el profesor.

Código 2.10.

```
     #include <stdio.h>

     int main(void)
     {
           int x;
           char y;
           float z;

     //Lectura de enteros
           printf("Introduzca un entero ....");
/*01*/     scanf(" %d", &x);
           printf("\nEntero: %d", x);
           printf("\n\n");

     //Lectura de caracteres
     //1. Con scanf
           printf("Introduzca un caracter ....");
           fflush(stdin); //evita problemas en la lectura de caracteres
/*02*/     scanf(" %c", &y);
           printf("\nCaracter: %c", y);
           printf("\n");
     //2. Con getchar
           printf("Introduzca un caracter ....");
           fflush(stdin); //evita problemas en la lectura de caracteres
           y = getchar();
           printf("\nCaracter: %c", y);
           printf("\n\n");

     //Lectura de numeros reales
           printf("Introduzca ahora un numero real ...");
/*03*/     scanf(" %f", &z);
           printf("\nReal: %f", z);

           return 0;
     }
```

2.2. PROYECTOS PROPUESTOS

Proyecto 2.1. ABACO

➔ *P2.1. Ejercicio 1.*

Cree un nuevo proyecto llamado **ABACO** con una función `main()` que al ejecutar muestre el texto "Calculadora ABACO" (vea Código 2.11.).

Código 2.11. Correspondiente al ejercicio propuesto en P2.1. Ejercicio 1.

```
#include <stdio.h>

int main(void)
{
	printf("Calculadora ABACO");
	return 0;
}
```

➔ *P2.1. Ejercicio 2.*

Amplíe el programa anterior para mostrar el siguiente texto utilizando saltos de línea, tabulaciones horizontales, etc. (Vea Código 2.12.),

```
-----------------------
--Calculadora "ABACO"--
-----------------------
```

Código 2.12. Correspondiente al ejercicio propuesto en P2.1. Ejercicio 2.

```
#include <stdio.h>

int main(void)
{
	printf("-----------------------\n");
	printf("--Calculadora \"ABACO\"--\n");
	printf("-----------------------\n");

	return 0;
}
```

¿Qué ocurre si no se escribe el carácter de control `'\n'`? ¿Qué ocurre al intentar introducir una tilde en la cadena de texto ÁBACO?

NOTA: Para mostrar letras con acentos en C debemos usar los códigos ASCII de dichos caracteres:

Á → `printf("%c",181);`
É → `printf("%c",144);`
Í → `printf("%c",214);`
Ó → `printf("%c",224);`
Ú → `printf("%c",233);`

Por lo que el código debería ser:

```
#include <stdio.h>
int main(void){
      printf("----------------------\n");
      printf("--Calculadora \"%cBACO\"--\n", 181);
      printf("----------------------\n");
      return 0;
}
```

➔ *P2.1. Ejercicio 3.*

¿Cuál es el resultado mostrado por pantalla correspondiente al código propuesto en Código 2.13.? (Note que en las líneas marcadas con `/*01*/` y `/*02*/` se ha cambiado el carácter de control `'\n'` por el carácter `'\r'`.)

Código 2.13. Correspondiente al ejercicio propuesto en P2.1. Ejercicio 3.

```
     #include <stdio.h>

     int main(void)
     {
/*01*/     printf("----------------------\r");
          printf("--Calculadora \"ABACO\"--\n");
/*02*/     printf("----------------------\n");

          return 0;
     }
```

¿Cuál es la sentencia que debería escribir para hacer sonar 5 veces la campana del sistema? Bastaría con la llamada a la función `printf()` con el carácter de control correspondiente repetido cinco veces: `printf("\a\a\a\a\a");`

➔ *P2.1. Ejercicio 4.*

Considere que queremos que nuestra primera versión de **ABACO** realice las operaciones aritméticas con números enteros cortos (16 bits) con signo. Será conveniente avisar inicialmente al usuario del

rango posible de valores. Hacemos, para eso, uso de la librería <stdint.h>. Puede verlo en Código 2.14.

Código 2.14. Correspondiente al ejercicio propuesto en P2.1. Ejercicio 4.

```
#include <stdio.h>
#include <stdint.h> // libreria con limites y rangos de tipos de datos

int main(void)
{
	printf("----------------------\n");
	printf("--Calculadora \"ABACO\"--\n");
	printf("----------------------\n");

	printf("NOTA. El rango permitido es");
	printf("\n[%hd,%hd]\n", INT16_MIN, INT16_MAX);
	return 0;
}
```

➔ *P2.1. Ejercicio 5.*

A continuación, se deben incorporar las sentencias necesarias para solicitar por teclado dos números enteros cortos con signo, que serán codificados y almacenados en las variables a y b (Vea Código 2.15.).

Código 2.15. Correspondiente al ejercicio propuesto en P2.1. Ejercicio 5.

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

int main(void)
{
	short int a, b; //a, b: Primer y segundo dato de entrada

	printf("----------------------\n");
	printf("--Calculadora \"ABACO\"--\n");
	printf("----------------------\n");
	printf("NOTA. El rango permitido es");
	printf("\n[%hd, %hd]\n", INT16_MIN, INT16_MAX);

	printf("\nIntroduzca el primer dato (a): ");
	scanf(" %hd", &a);	// Recuerde el espacio antes del %.
	printf("\nIntroduzca el segundo dato (b): ");
	scanf(" %hd", &b);	// Recuerde el espacio antes del %.
```

Código 2.15. (Cont.)

```
        printf("Los datos introducidos son: %hd y %hd", a, b);

        return 0;
}
```

Al ejecutar el programa resultante del anterior código,

- ¿Qué ocurre si el usuario introduce por teclado el número 32768? ¿Y para -32771?
- ¿Qué ocurre si introduzco una letra cuando solicita un número entero?

NOTA: Para comprender mejor el comportamiento de nuestro programa puede servir de ayuda el establecimiento de un *breakpoint* y realizar un *debug*.

NOTA: Puesto que estamos haciendo uso de la librería `<stdint.h>` podemos declarar las variables a y b como de tipo `int16_t`.

➔ *P2.1. Ejercicio 6.*

Realice los cambios necesarios en el anterior código para trabajar con números enteros con signo y codificados con 4 bytes (32 bits). A continuación añada el código correspondiente al cálculo de la suma, resta, multiplicación, cociente y resto de las variables a y b. Debe visualizar por pantalla el resultado de cada una de las operaciones realizadas.

Ejecute el programa en modo de depuración (haga un *debug*), introduzca un punto de ruptura (*breakpoint)*, ejecute paso a paso desde la sentencia asociada a este *breakpoint* y, por último, analice los resultados obtenidos con:

- `a = 65  b = 3`
- `a = 65  b = 0`

Si tiene dudas con el modo depuración (*debug*), consulte con el profesor. Los menús, las teclas y las opciones correspondientes al modo de depuración pueden ser distintos dependiendo del IDE que utilice. No obstante, el proceso de ejecución de un programa en modo depuración es el mismo en cualquier IDE.

➔ *P2.1. Ejercicio 7.*

Nuestra calculadora trabaja con enteros en formato decimal. Amplíe la funcionalidad de **ABACO** para que, además de las operaciones en base decimal, muestre el resultado en octal y hexadecimal.

➔ *P2.1. Ejercicio 8.*

Una vez que hemos desarrollado la calculadora **ABACO**, necesitamos generar una nueva versión **ABACO2** que trabaje con datos numéricos de coma flotante con 8 bytes y realice las operaciones de suma, resta, producto y cociente. (Vea Código 2.16.).

> ***NOTA:*** En este caso NO podemos realizar la operación de cálculo de resto, ya que estamos trabajando con valores reales. Recuerde que un tipo de dato es la especificación de un conjunto de valores (que llamamos DOMINIO) sobre el que se define una colección de operadores que se pueden usar para crear expresiones con variables o literales de ese tipo. Y recuerde que el operador módulo o resto sólo está definido para variables de tipo entero. Si intenta hacer una operación de resto con variables o literales de tipo **`float`** o **`double`** tendrá un error de compilación y el programa ejecutable no se creara.

Código 2.16. Correspondiente al ejercicio propuesto en P2.1. Ejercicio 8.

```
#include <stdio.h>
int main(void)
{
	double a, b; //a, b: Primer y segundo dato de entrada

	printf("------------------------\n");
	printf("--Calculadora \"ABACO2\"--\n");
	printf("------------------------\n");

	printf("\nIntroduzca el primer dato (a): ");
	scanf("%lf", &a);
	printf("\nIntroduzca el segundo dato (b): ");
	scanf("%lf", &b);

	printf("\nLa suma de %lf y %lf es %lf", a, b, a+b); //suma
	printf("\nLa resta de %lf y %lf es %lf", a, b, a-b); //resta
	printf("\nEl producto de %lf y %lf es %lf", a, b, a*b); //producto
	printf("\nEl cociente de %lf y %lf es %lf", a, b, a/b); //cociente

	return 0;
}
```

➔ *P2.1. Ejercicio 9.*

Realice las modificaciones necesarias para que:

- Los valores de a y b se muestren siempre con 3 decimales
- Los resultados por pantalla deben ser mostrados en formato científico con 2 decimales en la mantisa.

Compruebe el resultado obtenido para los siguientes datos:

- a = -1.45781
- b = 2.1e-3

Seguidamente, modifique el código para mostrar resultados con 3 decimales en la mantisa. ¿Qué diferencias aprecia con respecto a la anterior ejecución?

Proyecto 2.2. TEMPERATURA

➔ *P2.2. Ejercicio 1.*

El siguiente programa solicita el valor de la temperatura en grados Fahrenheit y muestra por pantalla el equivalente en grados Celsius. Esta transformación viene dada por la siguiente expresión:

$$\text{Celsius} = \frac{5}{9} \times (\text{Fahrenheit} - 32)$$

Código 2.17. Correspondiente al ejercicio propuesto en P2.2. Ejercicio 1.

```
     #include <stdio.h>

     int main(void)
     {
           double f; // Temperatura en Fahrenheit.
           double c; // Temperatura en Celsius.

           printf("\nTemperatura en Fahrenheit ... ");
/*01*/     scanf(" %lf", &f);

/*02*/     c = (5/9) * (f-32);

           printf("\nLa temperatura en grados Celsius ");
           printf("es ... %lf. ", c);

           return 0; // el programa ha acabo correctamente
     }
```

Compruebe el funcionamiento del programa propuesto en Código 2.17. para los siguientes valores de temperatura: 0 F y 10 F. ¿Qué ocurre en el segundo caso? ¿Cuál es la razón? ¿Es posible solucionarlo modificando el anterior código? (Recuerde el espacio en blanco insertado en la función `scanf()`, entre las comillas dobles de inicio de la cadena de formato y el tanto por ciento del especificador de formato: vea la línea marcada con `/*01*/`.)

➔ *P2.2. Ejercicio 2.*

Una vez que haya detectado el error (si le sirve de ayuda, el error está localizado en la línea marcado con `/*02*/` en Código 2.17.), sustituya la sentencia de conversión de temperatura por una de las siguientes sentencias y verifique el correcto funcionamiento con cada una de ellas.

```
c = (5 / 9.0) * (f - 32);
c = ((double)5 / 9) * (f - 32); //casting
c = 5 * (f - 32) / 9;
```

O una directiva de preprocesador: `#define _K (5.0 / 9.0)`

¿Qué ocurre al emplear la siguiente sentencia?

```
c = 5 * f - 32 / 9;
```

➔ *P2.2. Ejercicio 3.*

Modifique el código convenientemente para mostrar por pantalla valores de temperatura con dos decimales.

Proyecto 2.3. AUREO

➔ *P2.3. Ejercicio 1.*

Un **número áureo** es aquel que verifica la propiedad de que *al elevarlo al cuadrado se obtiene el mismo valor o resultado que al sumarle 1*. Cree un nuevo proyecto llamado **AUREO**, y escriba un programa que calcule y muestre por pantalla el número áureo con 4 decimales, considerando datos de coma flotante con 4 bytes (**`float`**). El programa debe, además, mostrar por pantalla que el número obtenido verifica la propiedad mencionada en este enunciado.

> ***NOTA***: Primero, determine el proceso para calcular el número áureo a partir de la información aportada en el enunciado de este ejercicio. Lógicamente si no sabe usted calcular ese número, será difícil que sepa qué sentencias debe ejecutar el ordenador para obtener su valor.
>
> ¿Necesita realizar alguna operación matemática distinta a las operaciones básicas (suma, resta, producto, etc.)? La gran mayoría de funciones matemáticas que necesite a lo largo del curso van a estar disponibles en la librería `<math.h>`, como por ejemplo las funciones `pow()` y `sqrt()`, cuyos prototipos son **`double`** `pow(`**`double`**`)` y **`double`** `sqrt(`**`double`**`)`, para realizar potencias y raíces cuadradas, respectivamente.
>
> Por último, tenga en cuenta que el número de decimales debe ser igual a 4. Eso requerirá de usted que indique a la función `printf()` que no trabaje con los 6 que, por defecto, muestra por pantalla cuando trabaja con variables de coma flotante (**`float`** o **`double`**): deberá hacer uso, en los especificadores de formato, del ancho de campo y ancho de precisión.

Código 2.18. Correspondiente al ejercicio propuesto en P2.3. Ejercicio 1.

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

int main(void)
{
      float aureo_f;

      printf("EL NUMERO AUREO\n");
      printf("-------------------------\n");

      aureo_f = (1.0 + sqrt(5.0)) / 2.0;

      printf("\n El numero aureo como float (%d bytes) ",sizeof(float));
      printf(" y 4 decimales es igual a %.4f", aureo_f);

      return 0;
}
```

¿Comprende la expresión que se ha utilizado para el cálculo del número áureo?

➔ *P2.3. Ejercicio 2.*

Ahora debemos realizar una sencilla ampliación: considerar además un formato de datos en coma flotante de 8 bytes. Se debe mostrar por pantalla el número áureo obtenido en formatos **`float`** y **`double`** con distintos decimales: desde 1 hasta 9.

➔ *P2.3. Ejercicio 3.*

Seguidamente, se debe mostrar el error entre los formatos de representación **`float`** y **`double`** para el número áureo, es decir, si `aureo_f` es el número áureo en formato **`float`** y `aureo_d` es el número áureo en formato **`double`** se debe mostrar por pantalla el error cometido al utilizar 32 bits (**`float`**) en lugar de 64 bits (**`double`**): `e = aureo_f - aureo_d`.

> ***NOTA*:** Tenga en cuenta los tipos de datos en cada caso y recuerde el concepto de *casting*.

Proyecto 2.4. OCULTO

➔ *P2.4. Ejercicio 1.*

Necesitamos desarrollar un sencillo programa que permita extraer un número oculto que se encuentra codificado en los bits 10, 9, 8 y 7 de una variable de tipo **`short int`**. El resultado se mostrará por pantalla en formato decimal. Lo primero que haremos será crear un nuevo proyecto, que llamaremos **OCULTO**.

Para usted, un alumno novel, este ejercicio tiene una complejidad media-alta. Por esta razón, es conveniente analizarlo en profundidad.

SOLUCIÓN: Dado que tenemos que trabajar con los bits correspondientes a una determinada variable (en este caso una variable **`short int`** con 16 bits), deberemos hacer uso de los operadores lógicos para poder extraer la información contenida en los bits 10, 9, 8 y 7. En concreto, como veremos ahora con un sencillo ejemplo ilustrativo, deberemos hacer uso del operador AND a nivel de bit (es decir, &) y del operador desplazamiento a la derecha (es decir, >>).

Como ejemplo, vamos a suponer que tenemos el número 1560 (0x0618, en hexadecimal). Su expresión binaria es: 11000011000, tal y como se muestra en la siguiente tabla.

BIT	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
ENTRADA	0	0	0	0	0	1	1	0	0	0	0	1	1	0	0	0

Como se indica en el enunciado, debemos obtener el número oculto que está codificado en los bits 10, 9, 8 y 7. En este caso (para el número 1560) estos 4 bits codifican el valor 1100, que expresado en decimal es el valor 12. Ya sabemos cuál debe ser el resultado final, pero ¿cómo implementamos un programa que nos dé la respuesta correcta?

Para resolverlo, debemos recordar que la operación AND de un determinado bit X con el valor binario 1 siempre nos devuelve el valor de dicho bit (ya sea 0 ó 1):

X AND 1
X=0 → X AND 1 = 0 AND 1 = 0
X=1 → X AND 1 = 1 AND 1 = 1

Por tanto, lo primero de todo es realizar la operación AND a nivel de bit con el valor de entrada que codifica nuestro valor secreto y un segundo valor, también de 16 bits, que tiene a cero todos sus bits excepto los correspondientes a la ubicación de nuestro número secreto, es decir, excepto los bits 7, 8, 9 y 10. Ese valor suele llamarse **máscara**.

En nuestro caso, la máscara deberá ser 0000 0111 1000 0000 (0x0780 en hexadecimal). Con él, extraeremos los valores binarios correspondientes a dichos bits en el número de entrada.

BIT	*15*	*14*	*13*	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*
MÁSCARA	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0

El resultado de `0x0618 & 0x0780` es igual a 0000 0110 0000 0000 (suponiendo que codificamos con 16 bits). Una vez que tenemos los valores para dichos bits, es necesario realizar un desplazamiento de los mismos hacia la izquierda: sólo así podremos obtener la expresión decimal del valor de esos cuatro bits aislados, obteniendo el valor 0000 0000 0000 1100, donde el que era el bit 7 ha quedado ahora en la posición más a la derecha. ¿Cuántas posiciones debemos desplazar? Un total de 7 posiciones. Con ello tenemos el número binario 1100 (en decimal, 12).

Una vez que se conoce el procedimiento para extraer el número oculto, podemos realizar su implementación en lenguaje C.

Código 2.19. Correspondiente al ejercicio propuesto en P2.4. Ejercicio 1.

```
#include <stdio.h>
int main(void)
{
      short int numero;
      short int oculto;

      printf("Introduzca el numero... ");      scanf("%hd",&numero);

      oculto = numero & 0x780;
      oculto = oculto >> 7;
      printf("\n El numero oculto es ... %hd", oculto);
      return 0;
}
```

NOTA: Quizá le llame la atención que digamos ahora que el operador & sea el operador AND a nivel de bit, cuando unas páginas más arriba le hemos explicado que ese operador era el operador dirección.

Los operadores también adquieren su significado o identidad en el contexto. Y así, cuando el operador & se aplica sobre una sola variable, como prefijo a ella, se trata efectivamente del operador dirección; y si encontramos el operador infijo en una expresión, entre dos variables [enteras], entonces se trata del operador AND a nivel de bit.

Ya lo irá viendo poco a poco. Bastantes caracteres significan uno u otro operador, según cómo esté ubicado en la expresión donde los encuentra. Y un mismo operador también podrá cambiar su comportamiento en función de esa ubicación.

➔ *P2.4. Ejercicio 2.*

A partir del programa anterior, debemos ser capaces de desarrollar una nueva aplicación en el proyecto **OCULTO2**, capaz de ocultar, en las ocho posiciones menos significativas de una variable llamada `oculto` de tipo **`unsigned short int`**, un valor formado a partir de los cuatro bits menos significativos de dos variables, llamadas `numeroL` y `numeroH`, de tipo **`unsigned short int.`** Los

cuatro bits más significativos del valor oculto de 8 bits se toman de los 4 bits menos significativos de la variable `numeroH`. Los cuatro bits menos significativos del valor oculto de 8 bits se toman de los 4 bits menos significativos de la variable `numeroL`. Al final del programa, se deberá mostrar el resultado por pantalla en formato decimal.

Proyecto 2.5. TIRO [EJERCICIO ADICIONAL]

➔ *P2.5. Ejercicio 1.*

El **tiro parabólico** es un ejemplo de movimiento realizado por un cuerpo en dos dimensiones o sobre un plano. El tiro parabólico es la *resultante de la suma vectorial del movimiento horizontal uniforme y de un movimiento vertical rectilíneo uniformemente acelerado.* Algunos ejemplos de cuerpos cuya trayectoria corresponde a un tiro parabólico son: proyectiles lanzados desde la superficie de la Tierra o desde un avión, el de una pelota de fútbol al ser despejada por el portero, el de una pelota de golf al ser lanzada con cierto ángulo respecto al eje horizontal. A continuación mostramos las conocidas ecuaciones que definen este tipo de trayectorias:

- $v_x = v_0 \cdot \cos\alpha$,
- $v_y = v_0 \cdot \operatorname{sen}\alpha - g \cdot t$,
- $x = v_0 \cdot t \cdot \cos\alpha$,
- $y = v_0 \cdot t \cdot \operatorname{sen}\alpha - \frac{1}{2} \cdot g \cdot t^2$,

donde v_0 es la velocidad inicial del objeto lanzado; α es el ángulo de lanzamiento respecto al eje horizontal, y que debe ser un ángulo entre 10 y 90 grados; g es la constante de aceleración de los cuerpos sometidos a la fuerza de atracción gravitatoria sobre la Tierra; y t es el tiempo transcurrido desde el instante del lanzamiento.

Con todo lo anterior, desarrolle un proyecto, denominado **TIRO**, que:

1. Solicite por teclado la velocidad inicial (v_0) y el ángulo de salida (α) medido en grados.
2. A continuación, solicite el tiempo que ha transcurrido desde el lanzamiento del proyectil.
3. A partir de estos datos, calcule y muestre por pantalla la posición (x, y) y la velocidad (v_x, v_y).

Pruebe ahora con los siguientes valores: (a) $v_0 = 150\ m/s$, $\alpha = 45^\circ$ y $t = 10\ s$; (b) $v_0 = 10\ m/s$, $\alpha = 20^\circ$ y $t = 10$.

***NOTA*:** Es necesario utilizar las funciones trigonométricas disponibles en la librería `<math.h>`, como son `sin`, `cos`, `tan`, `asin`, `acos` y `atan`. Tenga en cuenta que estas funciones trabajan con radianes, por lo que hay que convertir el dato del ángulo de grados a radianes: *α (radianes)= α (grados)* π/180.* Para el valor de π defina una constante `_PI = 3.14` (o use el valor definido en `<math.h>` con el nombre `M_PI`) y considere 9.8 m/s para la constante de la gravedad.

➔ *P2.5. Ejercicio 2.*

Amplíe la funcionalidad del anterior programa para que muestre ahora por pantalla la altura máxima que alcanzará el proyectil.

➔ *P2.5. Ejercicio 3.*

Suponga que un avión de combate vuela hacia una posición donde puede ponerse al alcance de un cañón antiaéreo. A partir del programa desarrollado, y tras introducir la altura de vuelo del avión, necesitamos implementar un nuevo proyecto denominado **PROYECTIL** que muestre por pantalla "IMPACTO" o "NO IMPACTO" en función de que el proyectil pueda abatir el avión, según los datos introducidos.

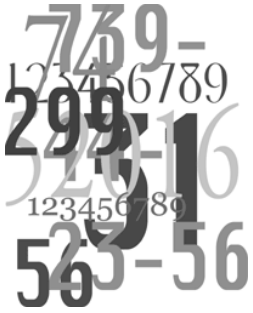
> ***NOTA***: Por ahora debe intentar evitar el uso de estructuras de control condicionales. Por lo tanto, si para la implementación de una solución para este ejercicio desea condicionar sentencias, deberá intentar hacer uso, únicamente, del operador interrogante dos puntos.

Estructuras de control condicionales

3

Una vez que el alumno ha adquirido, mediante la practica anterior (n. 2), los conceptos básicos de programación en C acerca de entrada/salida por pantalla, y ha comenzado a crear variables y expresiones de diferentes tipos de datos, el objetivo de esta práctica es aprender a utilizar las tres estructuras de control condicionales disponibles en el lenguaje C, es decir, **if**, **if/else** y **switch** (en algunos casos las soluciones propuestas con estas estructuras también podrán programarse utilizando el operador "interrogante dos puntos": consulte para conocer mejor ese operador el Capítulo 7 del manual de referencia). Para ello, en esta práctica presentaremos una serie de ejercicios cuya solución requiere el uso de este tipo de estructuras. Inicialmente, comenzaremos con unos ejercicios introductorios básicos. Al final, el alumno deberá ser capaz de realizar los siguientes ejercicios:

1. Un sencillo programa que permita determinar algunas posibles relaciones existentes entre tres números introducidos por teclado, así como ordenarlos de menor a mayor.



2. Un sencillo programa para resolver ecuaciones de primer y segundo grado con números reales y complejos.

$$x = \frac{-1 \pm \sqrt{1 - 4(2)(25)}}{4} = \frac{-1 \pm \sqrt{-79}}{4}$$
$$= \frac{-1 \pm \sqrt{79}\,i}{4}$$

3. Un display digital que monitorice la temperatura exterior de un bunker. Se necesita un programa que permita al usuario cambiar grados Fahrenheit a Celsius y viceversa a través de un menú de opciones. Además el programa deberá proporcionar información adicional acerca del rango de temperatura donde se encuentra la medida realizada.

Después de estos proyectos, se incluye además este ejercicio complementario.

4. Ampliar la funcionalidad del programa que permite estudiar el fenómeno físico asociado al **TIRO** parabólico y su aplicación a un *cañón antiaéreo*.

3.1. EJEMPLOS SENCILLOS RESUELTOS

Antes de comenzar con los problemas que componen la práctica, vamos a realizar unos pocos ejemplos iniciales que permitan al alumno introducirse en el uso de las estructuras de control condicionales.

Ejemplo 3.1. PARIMPAR

Creamos un nuevo proyecto denominado **PARIMPAR** donde deberemos escribir un programa que determine si un determinado entero es par o impar.

Antes de plantearnos el código de este programa necesitamos conocer de qué manera podríamos resolverlo. Proponemos un posible flujograma que exprese el algoritmo (sencillo, desde luego) que nos permite determinar si un entero positivo es par o impar. Se considera que el usuario introduce un número mayor o igual que cero. Por suerte disponemos, para las variables de tipo entero, del operador módulo (%) que devuelve el resto de la división de dos enteros.

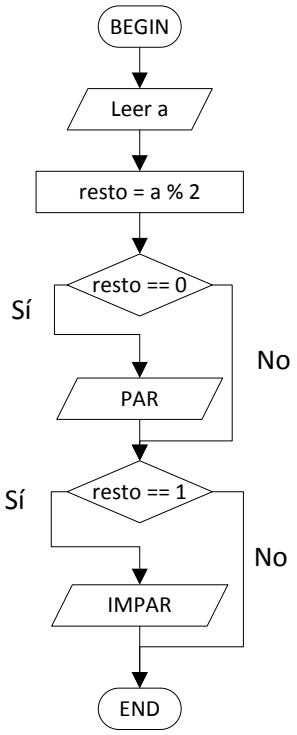


Figura 3.1. Primera propuesta de flujograma para el Ejemplo 3.1.

Una primera propuesta de flujograma podría ser la recogida en la Figura 3.1, que expresado en C, podría quedar de la forma que se muestra en Código 3.1. Este programa permite introducir un entero por teclado (almacenado en una variable de tipo **`short`**) y visualiza si es par o impar.

Código 3.1. Implementación del programa representado en el Flujograma de la Figura 3.1.

```
     #include <stdio.h>

     int main(void) {
           short int a, resto;

           printf("Introduzca un numero positivo....");
           scanf(" %hd",&a); //almaceno el valor leido en a

           resto = a % 2; //calcular el resto de dividir a y 2

/*01*/     if(resto == 0)    { //si el resto es 0 --> PAR
                 printf("El numero %hd es PAR\n",a);
           }
/*02*/     if(resto == 1)    { //si el resto es 1 --> IMPAR
                 printf("El numero %hd es IMPAR\n",a);
           }
           return 0;
     }
```

Como se puede comprobar, la solución propuesta emplea la estructura **if** dos veces con dos condiciones distintas (líneas marcadas con /*01*/ y /*02*/). En el primer **if**, se evalúa si el resto es 0 con objeto de mostrar por pantalla que el número introducido es par; mientras que en el segundo **if**, se evalúa si el resto es 1 para mostrar en ese caso que el número introducido es impar.

Si la condición recogida entre los paréntesis que siguen a la palabra reservada **if** es verdadera se ejecuta la sentencia (simple o compuesta: conjunto de sentencias simples agrupadas mediante llaves) condicionada mediante la estructura **if** y que se encuentran inmediatamente a continuación de la estructura condicional. Si la condición es falsa (es decir, igual a 0 en lenguaje C), no se ejecuta la sentencia. Por tanto, también sería válida la implementación propuesta en Código 3.2., donde se ha reemplazado la segunda condición `resto == 1` por `resto != 0`, utilizando el operador negación (!). Como se puede comprobar, la segunda condición se ha expresado a través de la negación de la primera.

Código 3.2.

```
           if(resto == 0)    { //si el resto es 0 --> PAR
                 printf("El numero %hd es PAR\n",a);
           }
/*01*/
           if(resto != 0)    { //si el resto es distinto de 0 --> IMPAR
                 printf("El numero %hd es IMPAR\n",a);
           }
```

El lenguaje C nos ofrece la estructura **`if-else`** para este tipo de construcciones, por lo que las sentencias de Código 3.2. podrían de forma equivalente expresarse como recoge Código 3.3.

***NOTA*:** En los primeros estándares de C no estaba definido un tipo de dato, muy común en muchos lenguajes, llamado tipo de dato Booleano: su dominio se limita a dos posibles valores: TRUE (Verdadero) o FALSE (Falso). El estándar C99 ya lo introduce.

Durante muchos años, los programadores en C han trabajado sin necesitar este tipo de dato. Y seguimos haciéndolo. En C, cuando se debe evaluar una expresión como verdadera o falsa se hace la siguiente asociación de valores: **todo lo que es distinto de cero se considera verdadero; y todo lo que es cero es o se considera falso**. Y cuando una expresión se evalúa como verdadera se le asigna el valor 1; y si es evaluada como falsa se le asigna el valor 0.

Y así, la expresión `resto != 0` será evaluada como verdadera si el valor de la variable `resto` es distinto de cero: es decir, cuando la variable `resto` sea verdadera (será verdadera si es distinto de cero). Y por lo tanto, las siguientes tres expresiones son equivalentes:

```
if(resto != 0)    sentencia;  // Verdadero si resto es distinto de cero
if(resto)         sentencia;  // Verdadero si resto es verdadero
if(!(resto == 0)) sentencia;  // Verdadero si es falso que resto sea cero
```

Y así, la expresión `resto == 0` será evaluada como verdadera si el valor de la variable `resto` es igual a cero: es decir, cuando la variable `resto` sea falsa (será falsa si es igual a cero). Y por lo tanto, las siguientes dos expresiones son equivalentes:

```
if(resto == 0)    sentencia;  // Verdadero si resto es igual a cero.
if(!resto)        sentencia;  // Verdadero si resto es falso.
```

Código 3.3.

```
            if(resto == 0)
            { //si el resto es 0 --> PAR
                  printf("El numero %hd es PAR\n",a);
            }
 /*02*/
            else
            { //si el resto es distinto de 0 --> IMPAR
                  printf("El numero %hd es IMPAR\n",a);
            }
```

Como puede observar de los distintos códigos mostrados anteriormente, se puede utilizar una estructura **`if-else`** porque las condiciones que finalmente hemos construido con la primera y la segunda estructura condicional **`if`** son mutuamente excluyentes: o es verdad el resultado de la operación módulo es igual a cero o no es verdad (`resto == 0` ó `resto != 0`).

El flujograma de la implementación de Código 3.3. se muestra en la Figura 3.2. Si compara los flujogramas propuestos en las dos figuras 3.1. y 3.2. comprobará que en la primera existen dos estructuras de control (dos diamantes con su condición de evaluación como TRUE o FALSE),

mientras que en la segunda sólo existe una estructura (un solo diamante con su condición de evaluación). En el primer caso tenemos 2 bifurcaciones abiertas; en el segundo tenemos una sola bifurcación cerrada. Es importante que distinga las sentencias propuestos en Código 3.2. y Código 3.3. En ambos listados de código hemos insertado las marcas `/*01*/` y `/*02*/`, respectivamente. Allí donde hemos ubicado la marca `/*01*/` podríamos insertar cualquier línea de código: las dos sentencias condicionadas son independientes y desde el punto de vista sintáctico el resultado de la evaluación del primer **`if`** no afecta a la posterior evaluación de la siguiente condición en el segundo **`if`**. En cambio, allí donde hemos ubicado la marca `/*02*/` no cabe escribir ninguna sentencia: si lo hace (puede escribir un simple punto y coma para hacer la prueba y verlo en su ordenador) el compilador denunciará un error sintáctico y no creará el ejecutable. Téngalo en cuenta, porque es ése un error habitual entre principiantes: una bifurcación cerrada emplea dos palabras clave para expresarse (**`if – else`**), pero entre ellas sólo debe haber una sentencia (simple o compuesta): la que se ejecutará si la condición del **`if`** se evalúa como verdadera.

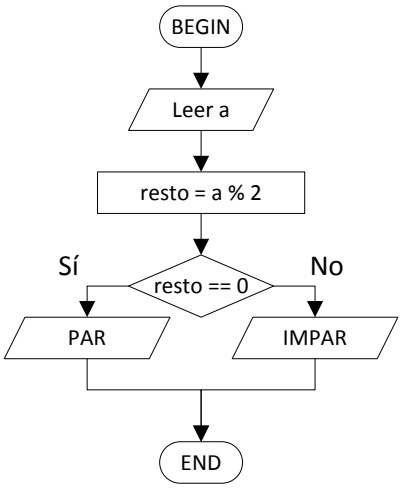


Figura 3.2. Segunda propuesta de flujograma para el Ejemplo 3.1.

Código 3.4. Par o impar, excluyendo el valor 0 (Solución 1 de 3)

```
#include <stdio.h>

int main(void)
{
      short int a, resto;

      printf("Introduzca un numero positivo....");
      scanf(" %hd", &a); // Observe espacio entre la comilla y el %.

      if(a == 0)
      { // comprobar si a es igual a 0
            printf("No se puede aplicar el concepto de par/impar al 0");
      }
```

Código 3.4. (Cont.)

```
        else
        { //en caso contrario (a distinto de 0)
              resto = a % 2; //calcular el resto de dividir a y 2
              if(resto == 0)
              { //si el resto es igual a 0
                    printf("El numero %hd es PAR\n", a);
              }
              else
              { //si el resto no es 0
                    printf("El numero %hd es IMPAR\n", a);
              }
        }
        return 0;
  }
```

En el pasado, algunos matemáticos consideraban que no se podía aplicar el concepto de par/impar al número 0. Partiendo de esta consideración (teóricamente incorrecta), suponga ahora que queremos ampliar la funcionalidad del programa para indicar esta particularidad. Para ello, se pueden presentar varias soluciones a través de estructura **`if-else`**.

- La primera podría ser usando estructuras anidadas o concatenadas (cfr. Código 3.4., cuyo flujograma puede ver en la Figura 3.3.)
- Otra posible solución sería la de emplear la estructura **`if-else if-else`**, que permite una concatenación de condicionales (cfr. Código 3.5.).
- Y una tercera solución podría ser utilizar condiciones compuestas (cfr. Código 3.6.). Vea las tres implementaciones y advierta las diferencias entre ellas, aunque evidentemente en los tres casos se expresa lo mismo.

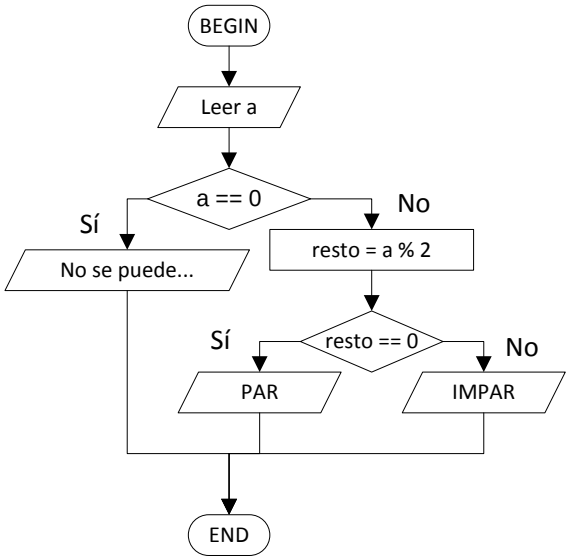


Figura 3.3. Tercera propuesta de flujograma para el Ejemplo 3.1. (cfr. Código 3.4.)

Código 3.5. Par o impar, excluyendo el valor 0 (Solución 2 de 3)

```
      #include <stdio.h>
      int main(void)
      {
            short int a, resto;

            printf("Introduzca un numero positivo....");
            scanf(" %hd", &a);

            resto = a % 2; //calcular el resto de dividir a y 2

            if(a == 0)
            { // comprobar si a es igual a 0
                  printf("No se puede aplicar el concepto de par/impar al 0");
            }

            else if (resto == 0)
            { //si a distinto de 0 y ademas el resto = 0
                  printf("El numero %hd es PAR\n",a);
            }

            else
            { //si el resto no es 0
                  printf("El numero %hd es IMPAR\n",a);
            }

            return 0;
}
```

Código 3.6. Par o impar, excluyendo el valor 0 (Solución 3 de 3)

```
#include <stdio.h>

      int main(void)
      {
            short int a, resto;

            printf("Introduzca un numero (entre -32768 y 32767)....");
            scanf(" %hd", &a);

            resto = a % 2; //calcular el resto de dividir a y 2

            if(resto == 0 && a != 0)
            { //si el resto es igual a 0 y ademas a es distinto de 0
                  printf("El numero %hd es PAR\n",a);
            }
```

Código 3.6. (Cont.)

```
/*01*/     else if(resto != 0)
           {
                 printf("El numero %hd es IMPAR\n",a);
           }

           else
           { //si a es 0
                 printf("No se puede aplicar el concepto de par/impar al 0");
           }
           return 0;
     }
```

En la segunda condición (/*01*/): `resto != 0`, no es necesario que `a != 0` : si el resto del cociente `a / 2` es distinto de `0`, a deberá ser necesariamente distinto de `0`.

Ejemplo 3.2. NOTAS

Creamos ahora un nuevo proyecto denominado **NOTAS** que debe mostrar por pantalla la calificación en texto (Suspenso, Aprobado, Notable, Sobresaliente) en función de una nota numérica introducida por teclado y almacenada en una variable n de tipo entero corto sin signo. En este caso, se ofrece primero (Código 3.7.) una solución mediante estructuras **if-else** (puede encontrar el flujograma correspondiente al siguiente código en el manual de referencia de la asignatura).

Código 3.7. Solución (Ejemplo 3.2.) con una concatenación de sentencias **if – else if – else**.

```
#include <stdio.h>
int main(void)
{
      short int n;

      printf("Introduzca una nota (numero entero positivo)...");
      scanf(" %hd", &n);

      if (n < 0)              {
            printf("Numero incorrecto (menor que 0).");
      }
      else if (n < 5)         {
            printf("SUSPENSO.");
      }
      else if(n < 7)          {
            printf("APROBADO.");
      }
```

Código 3.7. (Cont.)

```
        else if(n < 9)          {
              printf("NOTABLE.");
        }
        else if(n <= 10)        {
              printf("SOBRESALIENTE.");
        }
        else                    {
              printf("Numero incorrecto (mayor que 10).");
        }

        return 0;
  }
```

Sin embargo, también es posible utilizar una estructura **switch** (en el capítulo de algoritmia del manual de referencia hemos presentado ésta entre las estructuras derivadas, y la hemos llamado estructura CASE), que es una estructura de control con decisión múltiple, donde el compilador busca el valor de la expresión de control de la estructura en una lista de constantes de tipo entero o carácter. Si el programa encuentra en esa lista de constantes el valor de la expresión de control, entonces inicia la ejecución de las sentencias del bloque **switch** a partir de la línea marcada con esa constante. Opcionalmente el programador puede recoger una colección de sentencias a ejecutar por defecto para el caso de que el valor de la expresión de control no esté recogido en la lista de variables. La lista de sentencias a ejecutar dentro de una estructura **switch** se puede interrumpir en cualquier momento mediante una sentencia de ruptura.

***IMPORTANTE*:** La estructura **switch** sólo se puede implementar con expresiones evaluadas como enteras. Si la expresión de control del **switch** es de tipo **float**, o **double**, entonces el compilador delatará un erro sintáctico y no construirá el programa ejecutable.

Esta estructura, un poco arcaica en el ámbito de la programación pero aún útil, requiere de 4 de las 32 palabras reservadas de C: (1) **switch** para iniciar la estructura y junto a la que se indica la expresión de control; (2) **case** para indicar cada uno de los valores de la lista entre los que se buscará el valor de la expresión de control; (3) **default**, para indicar la entrada de código en caso de que no se dé ningún **case** con el valor de la expresión de control; y (4) **break**, para romper la línea de ejecución de las sentencias del bloque de la estructura **switch** y continuar con la siguiente sentencia posterior al cierre del bloque de sentencias controladas por el **switch**.

En Código 3.8. se muestra la solución al ejercicio propuesto y ya resuelto en Código 3.7., usando ahora una estructura **switch**.

Código 3.8. Solución (Ejemplo 3.2.) con una estructura **switch**.

```
#include <stdio.h>

int main(void) {
     unsigned short int n;

     printf("Introduzca una nota (numero entero positivo)...");
     scanf(" %hu", &n);

     switch(n) { // n es la expresión de control
     case 0:
           printf("SUSPENSO.");
           break;      // interrumpe la línea de ejecución.
     case 1:
           printf("SUSPENSO.");
           break;      // interrumpe la línea de ejecución.
     case 2:
           printf("SUSPENSO.");
           break;      // interrumpe la línea de ejecución.
     case 3:
           printf("SUSPENSO.");
           break;      // interrumpe la línea de ejecución.
     case 4:
           printf("SUSPENSO.");
           break;      // interrumpe la línea de ejecución.
     case 5:
           printf("APROBADO.");
           break;      // interrumpe la línea de ejecución.
     case 6:
           printf("APROBADO.");
           break;      // interrumpe la línea de ejecución.
     case 7:
           printf("NOTABLE.");
           break;      // interrumpe la línea de ejecución.
     case 8:
           printf("NOTABLE.");
           break;      // interrumpe la línea de ejecución.
     case 9:
           printf("SOBRESALIENTE.");
           break;      // interrumpe la línea de ejecución.
     case 10:
           printf("SOBRESALIENTE.");
           break;      // interrumpe la línea de ejecución.
     default:
           printf("Numero incorrecto (mayor que 10 o negativo).");
     }
     return 0;
}
```

Estas cuatro palabras trabajan juntas para expresar una sola estructura de control: no forman cada una de ellas una nueva estructura. Una estructura de control sirve para decidir la secuencialidad de

la ejecución de las sentencias de un programa en tiempo de ejecución, de acuerdo con el valor de las variables del programa o de acuerdo al valor de alguna expresión creada con algunas de ellas. La palabra que encabeza esta estructura es la palabra **switch**. Las sentencias correspondientes a cada **case** no requieren ir entre llaves. (Una excepción: que en un deteminado **case** desee crear una variable, local dentro del código de ese **case**. En tal caso deberá crear mediante llaves el ámbito de esa variable. Pero eso no tiene que entenderlo todavía: sí es conveniente decirlo para que pueda entenderlo más adelante, cuando vuelva a este capítulo del manual y ya haya comprendido qué es una variable de ámbito local.)

No es necesario siempre asociar alguna sentencia a cada etiqueta **case**. Las líneas recogidas en Código 3.9. son equivalentes a las anteriormente mostradas; ahora se ha omitido la sentencia **break** para los casos con notas iguales a 0, 1, 2, 3, 5, 7 y 9.

Código 3.9. Una segunda solución (Ejemplo 3.2.) con una estructura **switch**.

```
#include <stdio.h>

int main(void) {
      unsigned short int n;

      printf("Introduzca una nota (numero entero positivo)...");
      scanf(" %hu",&n);

      switch(n) {
      case 0:    case 1:    case 2:    case 3:    case 4:
            printf("SUSPENSO.");
            break;
      case 5:
      case 6:
            printf("APROBADO.");
            break;
      case 7:
      case 8:
            printf("NOTABLE.");
            break;
      case 9:
      case 10:
            printf("SOBRESALIENTE.");
            break;
      default:
            printf("Numero incorrecto (mayor que 10).");
      }
      return 0;
}
```

Las estructuras de tipo **switch** se suelen usar generalmente para construir programas de selección de menús, donde al usuario se le plantean varias opciones distintas y el propio usuario selecciona cuál de ellas se ejecuta. Ahora, con la programación conducida por eventos, y el uso del ratón, y los pulsadores, y las operaciones de click o de doble click, este tipo de menús de ejecución han

quedado más en desuso. Pero la estructura **`switch`** se sigue usando en muchas estructuras de programación y sigue siendo una herramienta muy útil y necesaria para la programación.

***NOTA*:** Es necesario aprender a expresar, haciendo uso de operadores relaciones y lógicos, aquellas condiciones que se evalúan como verdaderas o como falsas y que dan paso a la ejecución de una u otra parte del código del programa implementado.

Como ya habrá estudiado y sabrá, los operadores relaciones son 6: mayor que (>); menor que (<); mayor o igual que (>=); menor o igual que (<=); iguales (==); y distintos (!=). Los operadores lógicos son 3: and (&&); or (||); y not (!). Todos estos son los únicos operadores que evalúan una expresión como verdadera o falsa.

- ¿Cómo expresar la condición que se evalúe como verdadera si el valor de una variable llamada x verifica que es múltiplo de 3 positivo y menor que 1000?
- ¿Cómo será la expresión que se evalúe como verdadera si una variable llamada z es múltiplo de alguno de los cuatro primeros primos: 2, 3, 5 ó 7?
- ¿Y si queremos que la expresión se evalúe como verdadera si la variable z es múltiplo de alguno de esos cuatro primeros primos, pero sólo de uno de ellos?
- ¿Y cómo escribir la condición que se evalúe como verdadera si el valor de las variables llamadas a y b verifican que ambas son distintas de cero, ambas de signo contrario y tales que el valor absoluto de la diferencia de sus valores absolutos (mayor menos menor valor absoluto) no es mayor que 100?

No piense que hemos propuesto aquí cuatro condiciones de una complejidad irreal, de interés sólo académico: en realidad no son expresiones complejas, y es habitual encontrarse con condiciones de este estilo o más largas en su enunciado.

Piense en estas expresiones propuestas, y si no acierta a descubrir cómo presentarlas en C, entonces consulte en la última página de esta práctica.

3.2. PROYECTOS PROPUESTOS

Proyecto 3.1. TRESNUMEROS

➔ *P3.1. Ejercicio 1.*

Vamos a crear un nuevo proyecto llamado **TRESNUMEROS** que nos permita introducir tres números enteros (a, b y c) y visualice el texto "SUMA" si el tercer número (c) es el resultado de la suma de los dos primeros (a y b). En caso contrario se debe mostrar "NO SUMA". Lo tiene resuelto en Código 3.10.

Código 3.10. Solución a P3.1. Ejercicio 1.

```
#include <stdio.h>

int main(void) {
      int a, b, c;

      printf("Introducir el primer numero ...");     scanf(" %d", &a);
      printf("Introducir el segundo numero ... ");   scanf(" %d", &b);
      printf("Introducir el tercer numero ... ");    scanf(" %d", &c);

      if(a + b == c)    {
            printf("SUMA.");
      }
      else              {
            printf("NO SUMA.");
      }

      return 0;
}
```

➔ *P3.1. Ejercicio 2.*

A continuación, se debe visualizar el texto "SUMA" si la suma del valor de cualesquiera dos variables es igual al valor de la tercera. En caso contrario mostrar "NO SUMA".

Es muy posible que a usted se le ocurra una solución con tres estructuras de bifurcación cerrada **if – else**. Desde luego, esa será una solución correcta, pero le sugerimos que intente proponer una solución con una única estructura **if-else.**

➔ *P3.1. Ejercicio 3.*

Creamos ahora un nuevo proyecto, que llamaremos **TRESNUMEROSORDENADOS**, que nos permita introducir por teclado tres números enteros (guardados en las variables a, b y c) y que realice los necesarios intercambios de valores entre ellas para lograr que, al finalizar del programa, la variable

a contenga el valor menor, la variable b el valor intermedio y la variable c almacene el valor mayor. Además de estas tres variables, quizá le puede ayudar utilizar una variable auxiliar (d). Hay que procurar utilizar el menor número de estructuras condicionales.

***NOTA*:** Una vez que haya obtenido una solución correcta, en la sección "Intercambio de valores de dos variables" del Capítulo 7 del manual puede ver el resultado correspondiente a realizar el intercambio de variables utilizando el operador OR exclusivo, sin recurrir a variables auxiliares. Esta solución es menos evidente que la primera, aunque puede resultar más eficiente al no requerir una variable adicional. Desde luego, para intercambiar valores entre dos variables con el operador a nivel de bit OR exclusivo es necesario que esas variables sean de tipo entero. No se pueden aplicar operadores a nivel de bit para variables **`float`**, o **`double`** o **`long double`**.

Proyecto 3.2. ECUACION

➔ *P3.2. Ejercicio 1.*

Cree un nuevo proyecto llamado **ECUACION** cuyo programa permita resolver una ecuación de primer grado: $a \cdot x + b = 0$. Debe pedir al usuario los valores para los coeficientes a y b (almacenarlos en variables de tipo double) y mostrará por pantalla los valores de x que resuelven la ecuación. En la Figura 3.4 se muestra el diagrama de flujo correspondiente a este ejercicio.

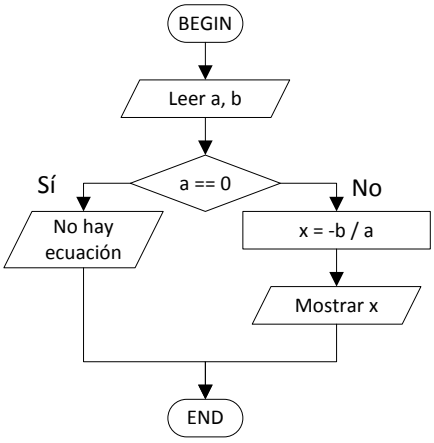


Figura 3.4. Flujograma para la primera parte del Proyecto 3.2.

➔ *P3.2. Ejercicio 2.*

Cree un nuevo proyecto llamado **ECUACION2** para resolver una ecuación de segundo grado: $ax^2 + bx + c = 0$. Debe pedir al usuario los valores para los coeficientes a, b y c (almacenarlos en variables de tipo **`double`**) y mostrará por pantalla los valores de x que resuelven la ecuación.

Considerar todos los posibles casos:

- a y b son iguales a 0 → Mostrar por pantalla "No hay ecuación"

- a es igual a 0 → Resolver la ecuación de primer grado y mostrar la solución
- El discriminante (d) es mayor o igual que 0 → Resolver la ecuación de segundo grado y mostrar las dos soluciones
- El discriminante (d) es menor que 0 → Mostrar por pantalla "No hay solución real".

En Código 3.11. se le ofrece, como ayuda, un posible código que resuelva este programa, donde unas partes del mismo se han sombreado. Intente completarlo.

Código 3.11. Solución a P3.2. Ejercicio 2.

```
#include <stdio.h>
#include <math.h>
int main(void)
{
      double a, b, c;
      double d; //discrimante
      // introduccion de parámetros...
      printf("****************************************\n");
      printf("**\t Ecuacion de Segundo Grado \t**\n");
      printf("****************************************\n");
      printf("Introduzca los coeficientes...\n\n");
      printf("a --> ");            scanf("    ",&a);
      printf("b --> ");            scanf("    ",&b);
      printf("c --> ");            scanf("    ",&c);
      // Ecuacion de primer grado...
      if(      )
      { // No hay ecuacion ...
            if(      )  {
                  printf("No hay ecuacion.\n");
            }
            else // Sí hay ecuacion de primer grado
                  {
                        printf("Ec. de primer grado.\n");
                        printf("Una unica solucion.\n");
                        printf("x1 -->    \n",      );
                  }
      }
                                                  
      {// Ecuacion de segundo grado. Soluciones imaginarias.
            printf("ERROR: Ecuacion sin soluciones reales.\n");
      }
            
      {// Ecuacion de segundo grado. Soluciones reales.
            printf("Las soluciones reales son:\n");
                  
            printf("\tx1 -->    \n",                    ));
            printf("\tx2 -->    \n",                    ));
      }
      return 0; //el programa ha acabado correctamente
}
```

➔ *P3.2. Ejercicio 3.*

Amplíe el anterior programa para que muestre por pantalla las soluciones complejas, en lugar de mostrar el mensaje de error.

> ***NOTA*:** Un número complejo Z viene dado por la suma de un número real X y un número imaginario Y (que es un múltiplo real de la unidad imaginaria, $\sqrt{-1}$, que se indica con la letra i): $Z = X + iY$

Proyecto 3.3. CONVTEMP

➔ *P3.3. Ejercicio 1.*

Cree un nuevo proyecto denominado **CONVTEMP** para implementar un programa que convierta el valor de de temperatura introducido por pantalla de grados Fahrenheit a Celsius o viceversa. Para ello, a través de una estructura **switch** debe solicitar al usuario que tipo de conversión desea "(1) Fahrenheit a Celsius" o "(2) Celsius a Fahrenheit". En caso de introducir un valor incorrecto (distinto de 1 y 2), se debe mostrar por pantalla "Opcion incorrecta". Si no recuerda la expresión de transformación, consulte la práctica del capítulo anterior. En Código 3.12. se le propone una solución a este ejercicio.

Código 3.11. Solución a P3.2. Ejercicio 2.

```
#include <stdio.h>
int main(void)
{
      double tempF, tempC; // Temperatura en Fahrenheit y en Celsius
      short int opcion;    // (1) Fahrenheit a Celsius; (2) al contrario

      printf("---Conversor de temperatura---\n");
      printf("(1) Fahrenheit --> Celsius\n");
      printf("(2) Celsius --> Fahrenheit\n");
      scanf(" %hd",&opcion);

      switch(opcion)
      {
      case 1:
            printf("\nTemperatura en Fahrenheit ... ");
            scanf(" %lf",&tempF);
            tempC = (5.0 / 9) * (tempF - 32);
            printf("\nLa temp. en grados Celsius es %.2lf", tempC);
            break;
```

Código 3.11. (Cont.)

```
            case 2:
                  printf("\nTemperatura en Celsius ... ");
                  scanf(" %lf",&tempC);
                  tempF = (9.0 / 5) * tempC + 32;
                  printf("\nLa temp. en grados Fahrenheit es %.2lf", tempF);
                  break;
            default:
                  printf("\nOpcion incorrecta.");
                  break;
            }
            return 0; // el programa ha acabado correctamente
      }
```

➔ *P3.3. Ejercicio 2.*

Modifique el anterior programa para que la variable opcion sea un carácter (`'a'` ó `'b'`) y limitar el rango de posibles valores de temperatura (en grados Celsius) al intervalo $[-150, +150]$. Utilice la escala equivalente para grados Fahrenheit. En caso de introducir un valor fuera del intervalo, se debe mostrar por pantalla el mensaje "Temperatura fuera de rango".

***NOTA*:** Se recomienda, aunque no es completamente necesario, declarar variables constantes para definir el rango: `const double tempRangoMax=150.0,tempRangoMin=-150.0;`
También se podría realizar utilizando la directiva de preprocesador **`#define`**.

➔ *P3.3. Ejercicio 3.*

Modifique el anterior programa para que indique por pantalla las siguientes situaciones:

- Si la temperatura es inferior a -89.5ºC, el programa debe indicar que se ha introducido un valor inferior a la menor temperatura registrada en la superficie de la Tierra.
- Si la temperatura es inferior a 0ºC, el programa debe indicar que el agua se encuentra en estado sólido.
- Si la temperatura está entre 0ºC y 100ºC, el programa debe indicar que el agua se encuentra en estado liquido.
- Si la temperatura es superior a 100ºC, el programa debe indicar que el agua se encuentra en estado gaseoso.
- Si la temperatura es superior 121ºC, el programa debe indicar que se ha superado la mayor temperatura conocida que soporta vida.

Se deben mostrar estos mensajes para los dos conversores implementados (Celsius a Fahrenheit y Fahrenheit a Celsius).

***NOTA*:** Se recomienda, aunque no es completamente necesario, declarar variables constantes para definir los valores anteriores.

```
const double tempRangoMax = 150.0, tempRangoMin = -150.0;
const double tempAguaSolido = 0.0, tempAguaGas = 100.0;
const double tempMinTierra = -89.5, tempMaxVida = 121.0;
```

Proyecto 3.4. PROYECTIL2. [Ejercicio Adicional]

➔ *P3.4. Ejercicio 1.*

En el último ejercicio de la anterior práctica, implementó usted un programa denominado **PROYECTIL** que permitía determinar si el disparo de cañón antiaéreo iba a impactar contra un avión que volaba hacia dicho cañón. Una posible solución podría ser la que se recoge en Código 3.12. Como puede ver en el listado, la sentencia marcada con `/*03*/` muestra el texto `"IMPACTO"` o `"NO IMPACTO"` por pantalla (`havion` es la altura actual del avión y `hmax` es la altura máxima alcanzada por el proyectil: cfr. líneas `/*01*/` y `/*02*/`). Partiendo de esta implementación que se le muestra, genere un nuevo proyecto **PROYECTIL2** que haga uso de la estructura **`if – else`** (en lugar de la anterior sentencia que usaba el operador interrogante – dos puntos) para indicar por pantalla si el proyectil impacta contra el avión.

***NOTA*:** Va a necesitar las funciones trigonométricas, incluidas en `math.h`, que ofrecen el cálculo del valor del seno y del coseno de un ángulo:

```
double sin(double);
double cos(double);
```

Código 3.12. Solución a P3.4. Ejercicio 1.

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

int main(void)
{
      const float PI=3.14; //PI: Constante PI
      const float g=9.8;     //g: Constante gravedad

      float v0; //v0: velocidad inicial
      float alpha; //alpha: angulo de salida (grados)
      float alpha_rad; //alpha_rad: angulo de salida (radianes)
      float t; //t: tiempo en segundos
      float vx, vy; //componentes horizontal y vertical de la velocidad
      float x,y; //posicion horizontal y vertical
```

Código 3.12. (Cont.)

```
/*01*/     float hmax; //altura maxima del proyectil
/*02*/     float havion; //altura del avion

           printf("-\tTIRO PARABOLICO\t\t-\n");
           printf("\nAngulo de salida: ");
           scanf(" %f", &alpha);
           alpha_rad = alpha * PI / 180; //conversion a radianes

           printf("\nVelocidad inicial: ");
           scanf(" %f", &v0);
           printf("\nTiempo transcurrido tras el lanzamiento: ");
           scanf(" %f", &t);

           /*Calculo de x,y,vx,vy*/
           x = v0 * t * cos(alpha_rad); // No hay aceleración horizontal.
           y = v0 * t * sin(alpha_rad) – 0.5 * g * pow(t, 2);
           vx = v0 * cos(alpha_rad);
           vy = v0 * sin(alpha_rad) - g * t;

           printf("\nPosicion x: %.3f m.", x);
           printf("\nPosicion y: %.3f m.", y);
           printf("\nVelocidad Vx: %.3f m/s.", vx);
           printf("\nVelocidad Vy: %.3f m/s.", vy);

           //Altura maxima
           hmax = pow(v0 * sin(alpha_rad), 2) / (2 * g);
           printf("\nAltura Maxima hmax: %.3f m.", hmax);

           //Suponer un avion volando hacia el c. antiareo
           printf("\n\nIntroduzca la altura actual de vuelo: ");
           scanf(" %f", &havion);

/*03*/     printf("%s", havion < hmax ? "IMPACTO": "NO IMPACTO");

           return 0;
     }
```

➔ *P3.4. Ejercicio 2.*

Ahora se le pide que modifique el proyecto **PROYECTIL2** de tal forma que:

- Pregunte al usuario si el ángulo de salida lo va a introducir en grados o radianes. Para ello el programa debe preguntar:

  `"Angulo en grados (0) o radianes (1)"`

  y en función del valor introducido (almacenado en un entero corto) se realiza la correspondiente conversión a radianes: únicamente en el caso de introducir el ángulo en grados se debe realizar dicha conversión. Si el usuario introduce un valor distinto de 0 ó 1, el programa debe finalizar con el mensaje `"No se ha elegido grados (0) o radianes (1)"`.

➔ *P3.4. Ejercicio 3.*

Seguidamente, ahora debe ampliar el programa **PROYECTIL2**:

- Incluya la siguiente restricción: el programa sólo se puede ejecutar cuando los valores introducidos para el ángulo de salida sea menor que 80° (o su equivalente en radianes: `80 * PI / 180`). En caso contrario el programa debe finalizar con el mensaje `"Angulo no permitido. Debe estar entre 0 y 80"` para el caso de grados y con el mensaje equivalente para el caso de ángulos en radianes

➔ *P3.4. Ejercicio 4.*

Ahora se le pide que incluya la siguiente restricción adicional:

- El programa sólo se puede ejecutar cuando los valores introducidos para la velocidad inicial sea mayor que 0. En caso contrario, finaliza con `"Velocidad debe ser mayor que 0"`.

➔ *P3.4. Ejercicio 5.*

Por último, modifique el programa de forma que:

- Ofrezca al usuario un menú con las siguientes opciones:

  `"(a) Calcular posicion y velocidad del proyectil en funcion del tiempo"`

  `"(b) Calcular altura maxima"`

  `"(c) Determinar impacto en objetivo en funcion de su altura"`

  y al escoger una de ellas (al introducir un carácter, `'a'` ó `'b'` ó `'c'`) se ejecuten las correspondientes sentencias. En caso contrario el programa debe finalizar con el mensaje `"Opcion incorrecta".`

- Para la opción (a), primero se debe solicitar al usuario que introduzca el tiempo transcurrido. Una vez introducido por teclado, se debe calcular el tiempo que tarda el proyectil en alcanzar el suelo: se alcanza t_{max} cuando tenemos que $y = 0$ (es decir, el proyectil ha terminado su recorrido y ha alcanzado el suelo). En ese caso, $t_{max} = 2 \cdot v_0 \cdot \sin(\propto) / g$. Deberá comprobar entonces que el valor introducido (t) verifica que $0 < t < t_{max}$. En caso afirmativo, se debe mostrar por pantalla los valores para la posición y la velocidad (componentes horizontales y verticales). En caso contrario, el programa finaliza con `"Instante de tiempo mayor que el tiempo de alcance"` o `"Instante de tiempo menor que 0"`, según corresponda.

- Para la opción (b), primero deberá calcular la altura máxima alcanzada por el proyectil y a continuación mostrarla por pantalla: `"La altura maxima alcanzada por el proyectil es …".`

- Para la opción (c), primero debe solicitar al usuario la altura actual del objetivo (en nuestro ejemplo el objetivo es un avión de combate volando hacia el cañón); segundo, calcular la altura máxima alcanzada por el proyectil y, a continuación, mostrar `"IMPACTO"` o `"NO IMPACTO"` por pantalla en función de la altura del avión y la altura máxima del proyectil.

***NOTA*:** Queda pendiente resolver las expresiones que se proponían unas páginas arriba, en una nota previa. Esas expresiones eran las siguientes:

- ¿Cómo expresar la condición que se evalúe como verdadera si el valor de una variable llamada x verifica que es múltiplo de 3 positivo y menor que 1000?

  ```
  (x % 3 == 0) && (x > 0) && (x < 1000)
  ```

  Son 3 las condiciones que debe cumplir: las 3 deben ser verdaderas. Por eso las concatenamos con el operador lógico AND. Se recomienda, aunque no es necesario, cuando se expresan condiciones compuestas de varias expresiones concatenadas con operadores lógicos, poner entre paréntesis cada unas de las condiciones.

- ¿Cómo será la expresión que se evalúe como verdadera si una variable llamada z es múltiplo de alguno de los cuatro primeros primos: 2, 3, 5 ó 7?

  ```
  (z % 2 == 0) || (z % 3 == 0) || (z % 5 == 0) || (z % 7 == 0)
  ```

  Ahora no es necesario que se evalúen como verdaderas todas las condiciones: basta que una de ellas sea verdadera para que podamos afirmar que se cumple la exigencia de la condición que hemos querido expresar.

- ¿Y si queremos que la expresión se evalúe como verdadera si la variable z es múltiplo de alguno de esos cuatro primeros primos, pero sólo de uno de ellos?

  ```
  ((z % 2 == 0) && (z % 3) && (z % 5) && (z % 7)) ||
  ((z % 2) && (z % 3 == 0) && (z % 5) && (z % 7)) ||
  ((z % 2) && (z % 3) && (z % 5 == 0) && (z % 7)) ||
  ((z % 2) && (z % 3) && (z % 5) && (z % 7 == 0))
  ```

  Ahora debe cumplirse a la vez que uno cualquiera de los primos divida exactamente a z, y que los otros 3 no lo hagan. Como puede ser cualquiera de los 4 primos el que sí lo haga, hay que testear las 4 combinaciones posibles. Son excluyentes entre sí, y a lo sumo sólo una de las 4 podrá evaluarse como TRUE.

- ¿Y cómo escribir la condición que se evalúe como verdadera si el valor de las variables llamadas a y b verifican que ambas son distintas de cero, ambas de signo contrario y tales que el valor absoluto de la diferencia de sus valores absolutos (mayor menos menor valor absoluto) no es mayor que 100?

  ```
  ((a > 0 && b < 0) || (a < 0 && b > 0)) &&
  ((a + b) > -100) && (a + b) < 100)
  ```

  La primera de las dos condiciones exige que ambas variables sean distintas de cero y ambas de signos contrarios. La segunda es una de las muchas formas en las que se puede construir una expresión que será evaluada como verdadera si la diferencia de los valores absolutos de dos enteros de signos opuestos tiene un valor absoluto menor que 100. Es posible que a usted no se le haya ocurrido expresarlo así, y que, sin embargo, sí tenga en mente otra formulación. No pretenda aprender ésta que aquí proponemos: lo importante es que logre expresar lo que se ha propuesto.

Estructuras de repetición (I)

4

4.1. Ejemplos sencillos resueltos

Ejemplo 4.1. *POTENCIA_W. Estructura* ***while****.*

Ejemplo 4.2. *LECTURACONDICIONADA. Estructura* ***do*****-*****while*** *(I).*

Ejemplo 4.3. *POTENCIA_DW. Estructura* ***do*****-*****while*** *(II).*

Ejemplo 4.4. *POTENCIA_F. Estructura* ***for****.*

4.2. Proyectos propuestos

Proyecto 4.1. *IMPRIMIRNUMEROS*.

Proyecto 4.2. *IMPRIMIRPARES*.

Proyecto 4.3. *PARESIMPARES*.

Proyecto 4.4. *PRESIONMEDIA*.

Proyecto 4.5. *CIFRAS*.

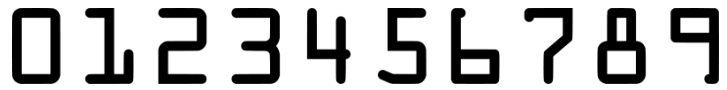
Proyecto 4.6. *TECLADO*. *Ejercicio adicional*.

Tras las primeras prácticas, donde el alumno ha aprendido a desarrollar programas secuenciales básicos y el uso de las estructuras de control condicionales, el siguiente objetivo de este manual de prácticas es ayudar a que el alumno aprenda a utilizar las estructuras de repetición o iteración disponibles en el lenguaje C: **`for`**, **`while`** y **`do-while`**. Dedicaremos dos sesiones de prácticas para profundizar en estos contenidos. Así, este capítulo comienza con unos ejemplos sencillos que permiten al alumno conocer los fundamentos de las tres estructuras de repetición. Al final de esta práctica, el alumno deberá ser capaz de realizar los siguientes ejercicios:

1. Una serie de sencillos programas que permitan mostrar el conjunto de *N* números naturales consecutivos ordenados de menor a mayor utilizando las tres estructuras de control.
2. Desarrollar unos programas para imprimir por pantalla el conjunto de números pares e impares en un determinado intervalo y realizar operaciones matemáticas con esos números.

3. La bomba de trasiego de combustible de un helicóptero dispone de un sensor de monitorización de presión. Desarrollar un programa que permita calcular el valor promedio medido por el sensor y determinar si las medidas están dentro del rango de funcionamiento del dispositivo.

4. Implementar un sencillo programa que permita extraer las cifras de un número entero decimal.



Después de estos cuatro proyectos propuestos, se incluye además este ejercicio complementario.

5. Desarrollar un programa que permita leer dígitos y/o letras por teclado. Aplicando la codificación estándar de caracteres ASCII se debe determinar la cantidad de dígitos y letras introducida en cada caso.

4.1. EJEMPLOS SENCILLOS RESUELTOS

Antes de comenzar con los problemas que componen esta sesión, es conveniente realizar una serie de ejemplos iniciales que permita al alumno introducirse en el uso de las estructuras de repetición.

Ejemplo 4.1. POTENCIA_W

Creamos un nuevo proyecto denominado **POTENCIA_W**. En él vamos a crear un programa que va a permitir al usuario introducir por teclado un valor entero y almacenarlo en una variable que vamos a llamar `base`, de tipo **`short int`**, y que deberá calcular el resultado de multiplicar sucesivamente ese número por sí mismo. El número de veces que se realiza la multiplicación es también introducido por teclado y almacenado en una variable —denominada `exponente`— de tipo **`short int`**. En el algoritmo de nuestro programa vamos a suponer que el exponente es mayor o igual que 0. El resultado es almacenado en la variable `solucion` de tipo **`long int`**.

CÁLCULO DE POTENCIA basada en estructura de control WHILE

Una posible solución, utilizando una estructura **`while`** podría ser, por ejemplo, la presentada en el flujograma de la Figura 4.1.

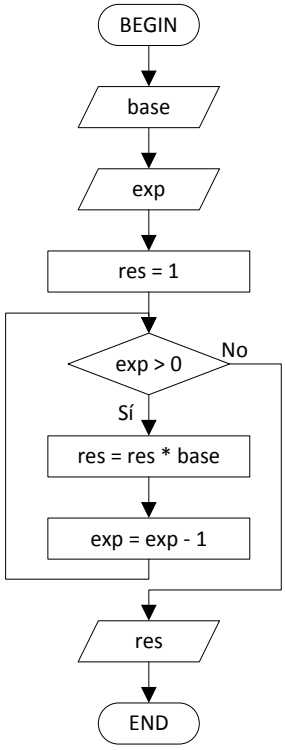


Figura 4.1. Algoritmo, expresado mediante un flujograma y utilizando una estructura básica **`while`**, para el cálculo de la potencia donde la base y el exponente son enteros positivos.

Este mismo algoritmo, expresado ahora con pseudocódigo, podría quedar de la siguiente manera:

```
Entrada de datos: base, exp.
WHILE exp > 0 IS TRUE
      res = res * base
      exp = exp – 1
END WHILE
Mostrar resultado: res.
```

Es importante que usted comprenda que ambos algoritmos, el expresado mediante el flujograma, y el expresado mediante pseudocódigo, son exactamente el mismo y que ambos recogen exactamente la misma secuencia de órdenes, o instrucciones, o sentencias. A continuación, en Código 4.1., se muestra el código en C de este algoritmo. Como puede comprobar, la sintaxis propuesta como pseudocódigo es muy similar a la que utiliza el lenguaje C. Habitualmente, en las prácticas, no vamos a presentar cada solución mediante flujogramas y pseudocódigo; pero es necesario que quien desea aprender a programar se plantee, antes de redactar las líneas de código en C, cuáles han de ser las sentencias y su orden que, ejecutadas, conformarán el algoritmo final que solventa el problema planteado.

Código 4.1. Cálculo de la potencia con base y exponente enteros positivos.

```
     #include <stdio.h>
     int main(void)
     {
           short int base, exponente;
           long int solucion = 1;

           printf("Introduzca la base:\n");
           scanf(" %hd", &base);
           printf("Introduzca el exponente:\n");
           scanf(" %hd", &exponente); //suponer exponente>0

           //calculo de la potencia mediante while
/*01*/     while(exponente > 0) {
                 solucion = solucion * base;
                 exponente = exponente – 1;
/*02*/     }
           printf("El resultado es: %ld", solucion); //mostrar solucion
           return 0; //el programa ha acabado correctamente
     }
```

Como se puede comprobar, la solución propuesta consta de una estructura de control **while** (entre las líneas marcadas con /*01*/ y /*02*/) gobernada por la condición exponente > 0: las sentencias incluidas dentro de esta estructura **while** se ejecutarán una y otra vez mientras que dicha condición sea verdadera. El valor de la variable exponente se decrementa una unidad en cada iteración hasta que se cumple la condición. Por tanto, las sentencias internas del bucle se ejecutan repetidamente un número de iteraciones igual al valor inicial del exponente. Además de decrementar el valor de exponente, en el interior del bucle **while** se actualiza el valor de la solución

multiplicando el valor actual de la solución (correspondiente a la iteración anterior) por la base. A continuación se muestra el resultado para un exponente igual a 2 y la base igual a 8.

Fíjese que, si el valor inicial del exponente es cero, el valor de la variable solucion será 1, que es el valor con el que ha quedado inicializado, y que además es el valor del elemento neutro del producto. Además, las sentencias interiores del bucle no se ejecutan *nunca* si el valor inicial de la variable exponente es menor o igual que 0.

Dentro de cualquier iteración es necesario que alguna o algunas de las variables que intervienen en la condición de permanencia alteren su valor. Sólo así la condición que gobierna la estructura de control de iteración, y que inicialmente, antes de entrar en el ámbito de las sentencias iteradas, era verdadera, podrá llegar, en algún momento, a ser evaluada como falsa y el programa podrá, por tanto, continuar con la ejecución secuencial de sentencias programadas más allá de la estructura de control. Por ejemplo: en nuestro ejemplo la condición de permanencia depende de la variable exponente, cuyo valor se decrementa de uno en uno cada vez que se vuelve a ejecutar la sentencia.

Ejemplo 4.2. LECTURACONDICIONADA

Creamos ahora un nuevo proyecto denominado **LECTURACONDICIONADA**. En él implementaremos un programa (recogido en Código 4.2.) que permita al usuario introducir sucesivamente valores enteros por teclado hasta que introduzca uno que sea negativo. Inicialmente, no exigimos al programa que realice operación alguna con esos valores introducidos: simplemente que siga solicitando nuevos valores mientras que el usuario siga introduciendo enteros positivos o el cero.

Código 4.2. Lectura condicionada basada en estructura de control **DO-WHILE**.

```
#include <stdio.h>
int main(void)
{
      short int x;
      do {//lectura indefinida condicionada mediante do-while
            printf("Introduzca un entero ... \n");
            scanf(" %hd",&x);
      }
      while(x >= 0);
      printf("Ha introducido un entero negativo\n");
      return 0;
}
```

Tal y como se puede comprobar en el código anterior, tras declarar la variable `x` se procede a ejecutar las sentencias contenidas en el bucle **`do...while`**, que en este caso son: primero, mostrar por pantalla el mensaje `"Introduzca un entero ... "` y, segundo, almacenar el número introducido por teclado en la variable `x`. Estas dos sentencias se van a ejecutar **siempre una primera vez** y, después, se evalúa la condición `x >= 0`. En el caso de que esta condición sea VERDADERA (el número introducido es mayor o igual que 0) se vuelve a repetir estas dos sentencias. Sin embargo, si la condición es FALSA (el número introducido es negativo) se termina la ejecución de las sentencias incluidas en el bucle. Por último, tras no cumplirse la condición, se muestra por pantalla el mensaje `"Ha introducido un entero negativo".`

Ya sabe que, en C (no ocurre así en otros lenguajes), cuando se crea una variable, ésta no tiene un valor conocido: toma uno cualquiera (podríamos decir que de forma aleatoria) de los valores definidos en el dominio (tipo de dato) en el que la variable ha sido declarada. Sí se puede, al declarar la variable, asignarle un valor inicial. En el código recién propuesto, la variable x tiene un valor inicial aleatorio cualquiera dentro del dominio de los enteros cortos (de tipo **`short int`**); y ese valor inicial no afecta a la ejecución del bucle. Sin embargo si hubiésemos propuesto la solución basada en **`while`** (presentada en Código 4.3. y en Figura 4.2. b), quizá pudiera ocurrir que las sentencias iteradas jamás se ejecutasen: en cualquier situación en la que la variable x hubiera tomado un valor inicial negativo.

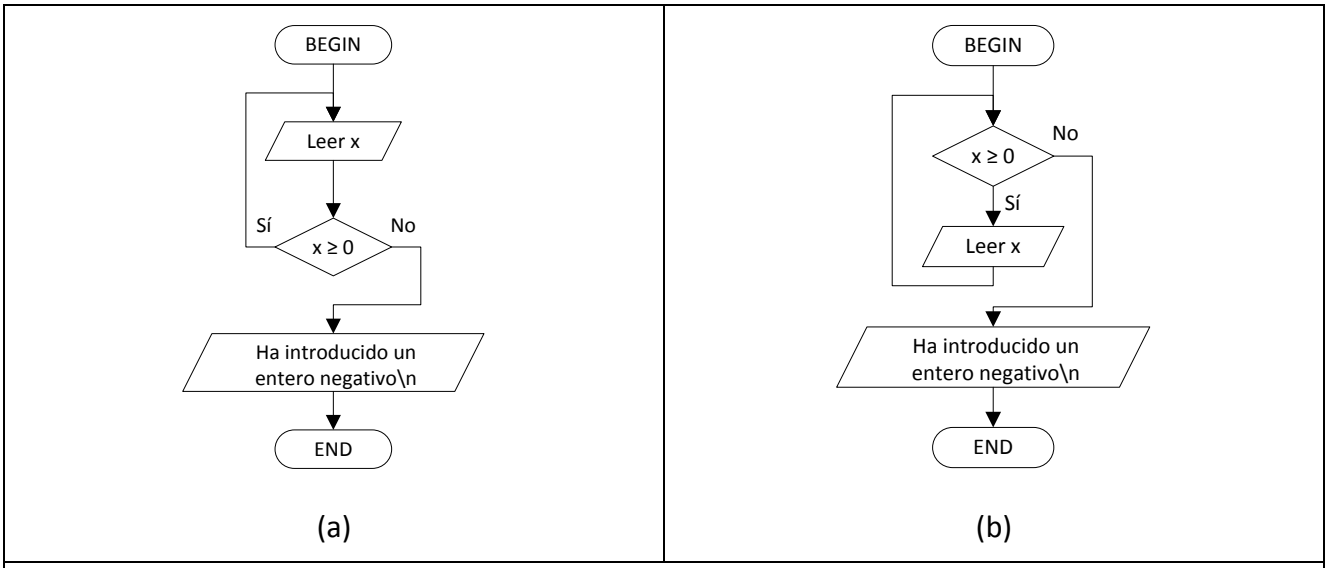


Figura 4.2. Flujogramas para resolver el problema enunciado en el Ejemplo 4.2. (a) Solución basada en una estructura **DO-WHILE**. (b) Solución basada en una estructura **WHILE**.

NOTA: Recuerde que la estructura básica de control WHILE primero pregunta y luego ejecuta. Y así, las sentencias controladas por esa estructura podrán ejecutarse de forma iterativa muchas veces… o tan sólo una vez… ¡o incluso ninguna!
Y recuerde que la estructura derivada de control DO-WHILE primera ejecuta y luego pregunta. Y así, las sentencias controladas por esa estructura se ejecutarán en cualquier caso, siempre al menos una vez.

Código 4.3. Lectura condicionada basada en estructura de control **WHILE**.

```
#include <stdio.h>
int main(void)
{
      short int x;
/* Hubiera sido conveniente haberle asignado un valor que
*  hiciera inicialmente cierta la condición del while
*/
      //lectura indefinida condicionada mediante while
      while(x >= 0){
            printf("Introduzca un entero ... \n");   scanf(" %hd",&x);
      }
      printf("Ha introducido un entero negativo\n");
      return 0;
}
```

Para finalizar esta breve introducción de la estructura **do-while**, el siguiente programa (**POTENCIA_DW**) presenta una posible solución (aunque ***errónea***) para el Ejemplo 4.1 empleando esta vez el bucle **do-while**.

Ejemplo 4.3. POTENCIA_DW

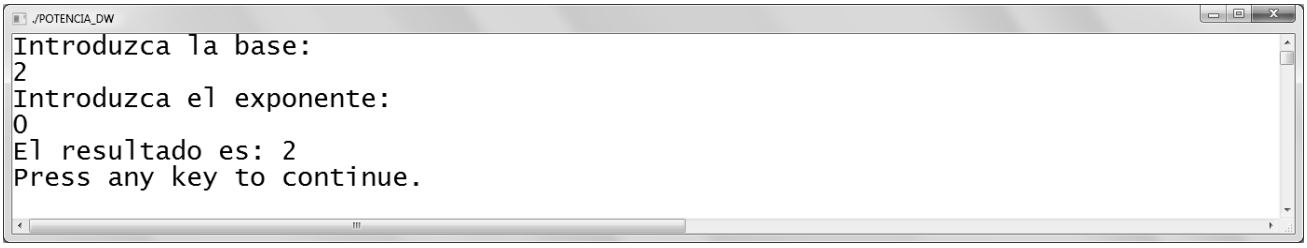
Volviendo al ejemplo 4.1 para el cálculo de potencia, se muestra ahora, en Código 4.4., una ***solución errónea*** a este ejemplo.

Código 4.4. Solución errónea para el cálculo de la potencia propuesta en Ejemplo 1.1.

```
#include <stdio.h>
int main(void) {
      short int base, exponente;
      long int solucion = 1;

      printf("Introduzca la base:\n");         scanf(" %hd",&base);
      printf("Introduzca el exponente:\n");    scanf(" %hd",&exponente);
                                               //suponer exponente > 0
      //calculo de la potencia mediante do-while
      do{
            solucion = solucion * base;
            exponente = exponente - 1;
      }while(exponente > 0);
      printf("El resultado es: %d",solucion); //mostrar solucion
      return 0; //el programa ha acabado correctamente
}
```

En este caso la solución es sintácticamente muy similar a la anterior. Sin embargo presenta una diferencia importante: las sentencias interiores del bucle **do-while** se ejecutan *al menos una vez*. Si después de haberse ejecutado, la condición se cumple, entonces vuelve a ejecutarse, así hasta que la condición establecida no se cumpla. Por ejemplo este sería el resultado de introducir 2 como base y 0 como exponente:



Tal y como hemos descrito anteriormente, después de introducir los valores de `exponente` (0) y `base` (2), directamente se actualiza el valor de la variable `solución` multiplicando su valor inicial (1) por el valor de la variable `base`, y seguidamente se decrementa el valor de `exponente` en una unidad. Tras ejecutar las dos sentencias interiores una primera vez, se evalúa la condición como falsa (`exponente` es igual a `-1` y no cumple con `exponente > 0`). Finalmente muestra el valor de la solución errónea (2 elevado a 0 no es 2) por pantalla.

Ahora mismo no resulta muy evidente la manera de solucionar el mal funcionamiento manteniendo la estructura **do-while**. Comprobará que con los conocimientos adquiridos al final de esta primera sesión será capaz de ofrecer una solución correcta al anterior código.

Ejemplo 4.4. POTENCIA_F

Por último, se presenta en Código 4.5. una solución para el cálculo de la potencia basada en la estructura **for**

Como se puede comprobar en este código, se ha introducido una nueva variable `i` cuya función es controlar el número de veces que son ejecutadas iterativamente la sentencia interior del bucle **for**. En concreto, en los paréntesis del bucle se incluyen dos sentencias y una expresión

```
for(sentencia1; expresion; sentencia2) {
     sentencia3;
}
```

En nuestro caso, sentencia1 es la asignación `i = 1`. El valor de sentencia2 es el incremento `i++`. Y la sentencia 3, que se encuentra ya fuera de los paréntesis del **for**, es `solucion = solucion * base;` La condición de permanencia recogida en lo que ha quedado llamado expresión en nuestro caso es `i <= exponente`. La sentencia3, seguida luego de la sentencia2 forman la porción de código que se repite iterativamente.

Código 4.5. Cálculo de la potencia (base y exponente enteros) con una estructura **for**.

```
     #include <stdio.h>
     int main(void)
     {
           short int base, exponente;
           long int solucion = 1;
           short int i; //variable de control para bucle for

           printf("Introduzca la base:\n");          scanf(" %hd", &base);
           printf("Introduzca el exponente:\n");     scanf(" %hd", &exponente);
                                                     //suponer exponente > 0.
           //calculo de la potencia mediante for
/*01*/     for(i = 1; i <= exponente ; i++)
           {
/*02*/           solucion = solucion * base;
           }

/*03*/     printf("El resultado es: %d", solucion); //mostrar solucion

// Otra solución mediante for, con el mismo proceso descrito en Código 4.4.:

           for(solucion = 1 ; exponente > 0 ; exponente--)
           {
                 solucion = solucion * base;
           }

           return 0; //el programa ha acabado correctamente
     }
```

El orden de ejecución del código recogido en Código 4.5. es el siguiente:

Paso 1. (línea `/*01*/`) El valor de la variable `i` es inicializado a 1 (tal y como se indica en la `sentencia1` del **for**).

Paso 2. (línea `/*01*/`) Se evalúa la condición establecida: `i <= exponente`. Si la expresión es verdadera, se continúa con la ejecución del bucle. En caso contrario (la expresión es falsa) no se continúa y se pasa al Paso 6.

Paso 3. (línea `/*02*/`) Se ejecuta todo el código interior del bucle (`sentencia3`): puede ser una sentencia simple o una sentencia compuesta.

Paso 4. (línea `/*01*/`) Se ejecuta la `sentencia2`, en nuestro caso esto implica incrementar una unidad el valor de la variable `i`.

Paso 5. Regreso al Paso 2. Es decir, la ejecución de sentencia1 sólo se ejecuta la primera vez y siempre antes de la evaluación de la condición de permanencia.

Paso 6. (línea `/*03*/`) Fin del bucle. En nuestro ejemplo, se pasa a imprimir por pantalla el valor de la variable `solucion`.

Por último, se muestra en Código 4.6. la solución equivalente a la anterior (basada en bucle o iteración **for**) ahora mediante bucle **while**.

- En la línea /*01*/ podemos ver la sentencia indicada antes en el Paso 1. Aquí se ve esta sentencia fuera de iteración y de condición. Se ejecuta una vez y sólo una, al margen de cualquier condición.
- En la línea /*02*/ tenemos la condición de permanencia (antes referenciada como Paso 2); mientras que ésta se evalúe como verdadera el código de la iteración (líneas /*03*/ y /*04*/) de irá ejecutando una y otra vez.
- La línea /*03*/ es la que en el código con el for formaba en cuerpo de la iteración.
- En la línea /*04*/ queda la sentencia que antes venía en el tercer espacio creado con los paréntesis y los puntos y comas de la sentencia **for**. Es el antes referenciado como Paso 4.

Código 4.6. La misma solución presentada en Código 4.5., ahora con una estructura **while**.

```
     #include <stdio.h>
     int main(void) {
           short int base, exponente;
           long int solucion = 1;
           short int i; //variable de control para bucle for

           printf("Introduzca la base:\n");         scanf(" %hd", &base);
           printf("Introduzca el exponente:\n");    scanf(" %hd", &exponente);
           //calculo de la potencia mediante for
/*01*/     i = 1;
/*02*/     while(i <= exponente) {
/*03*/           solucion = solucion * base;
/*04*/           i++;
           }
           printf("El resultado es: %d", solucion); //mostrar solucion
           return 0; //el programa ha acabado correctamente
     }
```

NOTA: Puede intentar resolver todos los problemas que vienen a continuación sin más que leyendo el enunciado y comenzando a picar código desde el teclado. Pero no suele ser ése el mejor camino para quien está en sus primeros pasos en el aprendizaje de la programación.

Tenga en cuenta que usted quizá no sólo está aprendiendo un lenguaje de programación como es el C, sino que además tiene que aprender a programar porque, en realidad, no sabe hacerlo con ningún lenguaje.

Por lo tanto, se le recomienda encarecidamente que, al menos en sus primeros pasos, no escriba una sola línea de código hasta haber reflexionado sobre el algoritmo que usted quiere desarrollar para resolver el problema y haberlo diseñado mediante un diagrama de flujo (que respete, por supuesto, las reglas de la programación estructurada) o mediante un pseudocódigo que respete también esas reglas.

4.2. PROYECTOS PROPUESTOS

Proyecto 4.1. IMPRIMIRNUMEROS

➔ *P4.1. Ejercicio 1.*

Cree un programa que visualice por pantalla los *N* primeros números naturales ordenados de menor a mayor utilizando las tres estructuras de control: **`while`**, **`do-while`** y **`for`**. El valor de *N* debe ser previamente introducido por el usuario a través de teclado y los números se deben mostrar en pantalla separados por un espacio.

Quizá ya se ha dado cuenta de que no es posible tener, dentro de un mismo proyecto, dos funciones principales. Así pues, para resolver el problema en las tres distintas formas, deberá crear tres proyectos distintos.

I) Primero, comience con la estructura **`while`**. Debe programar una solución utilizando una variable entera `N` y un contador `i` (proyecto **IMPRIMIRNUMEROS_W**)

II) Segundo, implemente una solución utilizando un bucle **`do-while`** (guardar en el proyecto **IMPRIMIRNUMEROS_DW**)

III) Tercero, cree el proyecto **IMPRIMIRNUMEROS_F** con una solución basada en la estructura **`for`**.

➔ *P4.1. Ejercicio 2.*

Proponga ahora una solución adicional, basada en **`while`**, que utilice únicamente la variable entera `N` (proyecto **IMPRIMIRNUMEROS_W_1**). En este caso, el programa deberá mostrar los números ordenados de mayor a menor, comenzando por `N` y terminando en el número 1. Ahora no dispondremos de la variable `i`; podemos resolver el problema planteado si vamos decrementando el valor de la variable `N` hasta que ésta alcance el valor 0.

➔ *P4.1. Ejercicio 3. [EJERCICIO ADICIONAL]*

Resuelva el mismo problema planteado en el ejercicio P4.1. Ejercicio 2, usando ahora la estructura **`do-while`** y también con la estructura **`for`**.

Proyecto 4.2. IMPRIMIRPARES

➔ *P4.2. Ejercicio 1.*

A partir de la primera solución basada en **`while`** (proyecto **IMPRIMIRNUMEROS_W**), genere un nuevo proyecto **IMPRIMIRPARES** en el ahora deberá implementar un programa donde se muestren por pantalla únicamente los **números pares incluidos entre los *N* primeros números naturales**. Además, el programa **se deberá ejecutar únicamente si el usuario introduce un número positivo** (es decir, `N > 0`).

➔ *P4.2. Ejercicio 2.*

A partir del código del ejercicio anterior, genere un nuevo proyecto **IMPRIMIRPARES2,** donde deberá crear un programa para visualizar por pantalla el *conjunto de números pares incluidos entre dos números* `N` *y* `M`, introducidos por teclado. El programa debe forzar al usuario a introducir un valor para `N` *mayor que 2* y, además, `M` *tiene que ser mayor que* `N`. En caso de no existir ningún número par, deberá mostrar el mensaje "*No existen enteros pares*". En caso contrario, el programa, además de mostrar los pares existentes en el intervalo definido por el usuario, deberá también indicar, por pantalla, cuántos pares hay en el intervalo.

Proyecto 4.3. PARESIMPARES

➔ *P4.3. Ejercicio 1.*

A partir del anterior ejercicio, deberá crear un nuevo proyecto, que llamará **PARESIMPARES,** para un programa que permita calcular:

(1) el producto de los números pares incluidos entre `N` y `M`;

(2) la suma de los números impares incluidos entre `N` y `M`.

Al finalizar el programa se debe mostrar el resultado de ambas operaciones.

Proyecto 4.4. PRESIONMEDIA

➔ *P4.4. Ejercicio 1.*

Cree un nuevo proyecto llamado **PRESIONMEDIA** donde implementaremos un programa que permita calcular el promedio de las medidas realizadas por el sensor de presión de una bomba de trasiego de combustible. Las medidas son introducidas por el usuario a través de teclado **hasta que introduce el valor negativo -1**. Además, **si se introducen menos de 10 valores no se muestra el valor medio calculado** y, en su lugar, se visualiza el mensaje "Numero insuficiente de valores".

Para los valores de las sucesivas lecturas de la presión, y para las variables necesarias para el cálculo de la media, utilice variables de tipo **`double`**. Evidentemente, la variable que contabilice el número de valores introducidos no deberá ser de coma flotante, sino entera. Muestre el resultado con una precisión de 2 decimales.

➔ *P4.4. Ejercicio 2.*

A continuación, debemos limitar el rango posible de valores al siguiente intervalo [+15,+35] bares. En caso de **introducir un valor fuera de rango**, se avisa mediante el mensaje "Medida fuera de rango" y ese **dato no se tiene en cuenta para el cálculo de la presión media** ni debe considerarse como dato introducido.

Proyecto 4.5. CIFRAS

➔ *P4.5. Ejercicio 1.*

Cree un nuevo proyecto llamado **CIFRAS** donde debe escribir un programa que muestre un mensaje indicando si un determinado dígito (que el usuario introduce por pantalla) forma parte de la codificación decimal de un número entero también introducido por teclado en esa base decimal.

PRUEBA: Si introduce el entero 2345 y el dígito 7 el programa deberá indicar que el dígito 7 no está contenido en el número 2345.

PRUEBA: Si introduce el entero 2345 y el dígito 4 el programa deberá indicar que el dígito 4 sí está contenido en el número 2345.

➔ *P4.5. Ejercicio 2.*

Amplíe ahora el programa anterior de forma que verifique que el dígito introducido es efectivamente un dígito (carácter entre el `'0'` y el `'9'`), y que no admita un número entero introducido que sea menor que 10000: en ese caso solicitará otro entero hasta que el usuario ingrese uno menor que ese límite indicado.

➔ *P4.5. Ejercicio 3. [EJERCICIO ADICIONAL]*

Amplíe otra vez el programa anterior para que el entero introducido sea un número cuadrado perfecto menor que 10000.

> ***NOTA***: Un número entero es cuadrado perfecto si se puede expresar como potencia al cuadrado de otro entero: 1, 4, 9, 16, 25, 36, 49, 64, 81, …

Proyecto 4.6. TECLADO [EJERCICIO ADICIONAL]

➔ *P4.6. Ejercicio 1.*

Cree un nuevo proyecto llamado **TECLADO** donde deberá implementar un programa que permita al usuario introducir una serie de números y/o letras hasta que se introduzca el carácter `'q'`. Este proceso se irá repitiendo hasta que el usuario confirme que no desea continuar. Únicamente puede utilizar una variable para almacenar el dato introducido por teclado.

A través de dos contadores, el programa irá contando la cantidad de dígitos, por un lado, y de letras, por otro que se han introducido. Al finalizar el programa mostrara el resultado de ambos contadores.

PRUEBA: Si introduce la serie de caracteres `"Me encanta aprender a programar en pleno siglo 21q"` (el carácter `'q '` indica que se ha terminado de introducir la cadena de entrada) el programa deberá indicar que se han introducido 39 letras y 2 números. A continuación el programa

pregunta al usuario si quiere introducir otra entrada; si el usuario contesta que sí, se comienza de nuevo la aplicación; si contesta que no, la aplicación termina.

NOTA: Como usted sólo sabe, hasta el momento, introducir caracteres por teclado usando la función `scanf()`, será necesario que, después de introducir cada nuevo carácter, pulse la tecla intro. Desde luego, es una solución horrible, pero es una primera solución: más adelante aprenderá otras funciones que facilitan el ingreso de caracteres por teclado.

SOLUCIONES A LOS EJERCICIOS DE LOS PROYECTOS PROPUESTOS

Para que este manual fuera verdaderamente eficaz, casi sería mejor que no incluyera esta sección de soluciones, al final de cada capítulo. Porque ahora, cuando las mire, seguramente comprenderá perfectamente el código que le proponemos. Pero sólo si usted ha sido capaz de llegar a una correcta solución sin mirar estas otras soluciones, habrá logrado aprovechar su tiempo de trabajo y, verdaderamente, habrá aprendido a programar. Sólo hay un camino para iniciarse en la programación: programar.

Además, para verificar si las soluciones que usted ha planteado son correctas, lo más indicado no es comparar su código con el que se muestra a continuación. Lo mejor es ejecutar su código y verificar que, efectivamente, resuelve el problema para el que fue creado el programa.

Así que no parece necesario mostrar soluciones a los ejercicios planteados. De todas formas, puede ser útil que usted mire, una vez ha implementado la suya, estas otras soluciones que aquí se le ofrecen. Porque quizá, mirando el código de otros, y una vez usted ha reflexionado sobre el problema planteado y ha llegado a su propia solución, pueda descubrir nuevas vías de alcanzar una feliz solución.

El alumno que, cansado de buscar una solución y no llegar a ella, consulta como último recurso estas soluciones, debe saber que no aprende así a programar. Es como quien resuelve los crucigramas o los sudokus mirando la última página donde se hallan las soluciones. Simplemente, logrará saber cómo otros han llegado a una solución. Pero ese alumno no habrá aprendido a programar porque, de hecho, no habrá programado. Si esta advertencia queda clara, entonces podemos continuar: dejaremos las soluciones a la vista, en cada final de práctica. Que cada cual haga de ellas el uso útil que le parezca.

Código 4.7. Solución para P4.1. Ejercicio 1 (I).

```
#include <stdio.h>
int main(void)
{
      unsigned short N; // Límite superior
      unsigned short i; // Variable contador.
// Introducir valor de N...
      printf("Introduzca valor de N... ");     scanf(" %hu", &N);
// Mostrar por pantalla los enteros solicitados...
      i = 0;      // Mostrará también el entero 0.
      while(i <= N)
      {
            printf("%hu ", i);
            i++;
      }
// También podría haberse escrito (más frecuente en C)...
//    while(i <= N) printf("%hu ", i++);

      return 0;
}
```

Código 4.8. Solución para P4.1. Ejercicio 1 (II).

```
#include <stdio.h>
int main(void)
{
	unsigned short N; // Límite superior
	unsigned short i; // Variable contador.

// Introducir valor de N...
	printf("Introduzca valor de N... ");
	scanf(" %hu", &N);

// Mostrar por pantalla los enteros solicitados...
	i = 0;	// Mostrará también el entero 0.
	do
	{
		printf("%hu ", i);
		i++;
	}while(i <= N);

	return 0;
}
```

Código 4.9. Solución para P4.1. Ejercicio 1 (III).

```
#include <stdio.h>
int main(void)
{
	unsigned short N; // Limite superior
	unsigned short i; // Variable contador.

// Introducir valor de N...
	printf("Introduzca valor de N... ");
	scanf(" %hu", &N);

// Mostrar por pantalla los enteros solicitados...
	for(i = 0 ; i <= N ; i++) {
		printf("%hu ", i);
	}

	return 0;
}
```

Código 4.10. Solución para P4.1. Ejercicio 2.

```
#include <stdio.h>
int main(void)
{
	unsigned short N; //Límite superior

// Introducir valor de N...
	printf("Introducir N...");
	scanf(" %hu", &N);

	while(N > 0) {
		printf("%hu ", N);
		N = N - 1;
	}

	return 0;
}
```

Código 4.11. Solución para P4.2. Ejercicio 1.

```
#include <stdio.h>
int main(void) {
	short N; //dato introducido por el usuario (N primeros numeros)
	unsigned short i; //contador

	// Introducir valor de N...
	printf("Introducir N...");
	scanf(" %hd", &N);

	// Imprimir los N primeros pares...
	if(N > 0) {
		i = 1;
		while(i <= N) {
			if(i % 2 == 0) {
				printf("%hu ", i);
			} // Final del if
			i++;
		} // Final del while
	}
	else {
		printf("Se ha introducido un entero negativo o nulo.");
	}

	return 0;
}
```

Código 4.12. Solución para P4.2. Ejercicio 2.

```
#include <stdio.h>

int main(void)
{
	unsigned int N; //Límite inferior --> Mínimo entero
	unsigned int M; //Límite superior --> Máximo entero
	unsigned int i; //contador
	unsigned int p; //numero de pares

//Introducir N...
	do
	{
		printf("Introducir N...");
		scanf(" %u", &N);
	} while(N <= 2);

//Introducir M...
	do
	{
		printf("Introducir M...");
		scanf(" %u", &M);
	} while(M <= N);

	i = N; // contador inicializado al límite inferior
	p = 0; // inicializar a cero el contador para los pares

//Imprimir todos los pares entre N y M
	while(i <= M)
	{
		if(i % 2 == 0)
		{
			printf("%u ", i);
			p = p + 1;		// también p++;
		} // Final del if
		i++;
	} // Final del while

//Muestra la cantidad de pares entre N y M
	if(p == 0)
	{
		printf("No existen pares.");
	}
	else
	{
		printf("El numero de pares es %u", p);
	}

	return 0;
}
```

Código 4.13. Solución para P4.3. Ejercicio 1.

```
#include <stdio.h>

int main(void)
{
	unsigned int N; //Límite inferior --> Mínimo entero
	unsigned int M; //Límite superior --> Máximo entero
	unsigned int i; //contador
	unsigned int producto; //producto de pares
	unsigned int suma; //suma de impares

	//Introducir N...
	do
	{
		printf("Introducir N...");
		scanf(" %u", &N);
	} while(N < 2);

//Introducir M...
	do
	{
		printf("Introducir M...");
		scanf(" %u", &M);
	} while(M < N);

	i = N; // contador inicializado al límite inferior
	producto = 1; // inicializar a uno el producto
	suma = 0; // inicializar a cero la suma

	//Calcular producto y suma
	while(i <= M)
	{
		if(i % 2 == 0)
		{
			producto = producto * i;	// también producto *= i;
		} // Fin del if
		else
		{
			suma = suma + i;		// también suma += i;
		} // Fin del else
		i++;
	} // Fin del while

//Mostrar el resultado final para producto y suma
	printf("\nEl producto de los pares es %u", producto);
	printf("\nLa suma de los impares es %u", suma);

	return 0;
}
```

Código 4.14. Solución para P4.4. Ejercicio 1.

```
#include <stdio.h>

int main(void)
{
	float p; // presión leída
	float s; //sumatorio de presiones
	int c; // numero de lecturas

	printf("Lectura de medidas de presion en bomba de combustible\n");
	printf("Introduzca valores consecutivamente y pulse INTRO.\n");
	printf("Para finalizar introduzca -1 y pulse INTRO.\n");

	s = c = 0;

	do
	{
		scanf(" %f", &p);
		if(p != -1.0)
		{
			s = s + p;
			c = c + 1;
		} // Fin del if
	} while(p != -1.0); // Fin del do-while

	if(c >= 10)
	{
		printf("\nEl valor medio de presion es ... %.2f",s/c);
	}
	else
	{
		printf("\nNumero insuficiente de valores.");
	}

	return 0;
}
```

Código 4.15. Solución para P4.4. Ejercicio 2.

```
#include <stdio.h>

int main(void)
{
	float p; // presión leída
	float s; //sumatorio de presiones
	int c; // numero de lecturas
```

Código 4.15. (Cont.)

```
        printf("Lectura de medidas de presion en bomba de combustible\n");
        printf("Introduzca valores consecutivamente y pulse INTRO.\n");
        printf("Para finalizar introduzca -1 y pulse INTRO.\n");

        s = c = 0;

        do
        {
              scanf(" %f", &p);
              if(p >= 15 && p <= 35)
              {
                    if(p != -1.0)
                    {
                          s = s + p;
                          c = c + 1;
                    } // Fin del if
              }
              else
              {
                    printf("Valor fuera de rango\n");
              } // Fin del else
        } while(p != -1.0); // Fin del do-while

        if(c >= 10)
        {
              printf("\nEl valor medio de presion es ... %.2f",s/c);
        }
        else
        {
              printf("\nNumero insuficiente de valores.");
        }

        return 0;
  }
```

Código 4.16. Otra Solución para P4.4. Ejercicio 2. (usando las sentencias **break** y **continue**)

```
  #include <stdio.h>

  int main(void) {
        float p; // presión leída
        float s; //sumatorio de presiones
        int c; // numero de lecturas

        printf("Lectura de medidas de presion en bomba de combustible\n");
        printf("Introduzca valores consecutivamente y pulse INTRO.\n");
        printf("Para finalizar introduzca -1 y pulse INTRO.\n");
```

Código 4.16. (Cont.)

```
        s = 0;
        c = 0;
        do {
              scanf(" %f", &p);
              if(p == -1.0) break;
              if(p < 15 || p > 35) continue;
              s = s + p;
              c = c + 1;
        } while(p != -1.0);

        if(c >= 10) {
              printf("\nEl valor medio de presion es ... %.2f",s/c);
        }
        else {
              printf("\nNumero insuficiente de valores.");
        }
        return 0;
  }
```

Código 4.17. Solución para P4.5. Ejercicio 1.

```
  #include <stdio.h>

  int main(void) {
        //declaración de variables
        int n; //numero entero introducido
        int c; //cifra a comprobar que está incluida en el entero n
        int cifra; //cifra extraída en cada iteración
        int bandera; //variable auxiliar para saber si n contiene la cifra

  //Lectura de valores para n y c
        printf("\nIntroduce un entero:");        scanf(" %d", &n);
        printf("\nIntroduce una cifra:");        scanf(" %d", &c);

  //Nota: mientras el número no sea cero quedan cifras por extraer
        bandera = 0;
        while(n > 0) {
  //se extrae la cifra
              cifra = n % 10;
              if (cifra == c) {
                    bandera = 1;
              }
  //se descarta la cifra del número
              n = n / 10;
        }
```

Código 4.17. (Cont.)

```
        if(bandera == 1)  printf("\nEl numero contiene la cifra");
        else              printf("\nEl numero no contiene la cifra");

        return 0;
}
```

Código 4.18. Solución para P4.5. Ejercicio 2.

```
#include <stdio.h>

int main(void)
{
//declaración de variables
        int n; //numero entero introducido
        int c; //cifra a comprobar que está incluida en el entero n
        int cifra; //cifra extraída en cada iteración
        int bandera; //variable auxiliar para saber si contiene la cifra

//Lectura de n condicionada a que sea menor o igual que 10000
        do
        {
                printf("\nIntroduce un entero:");
                scanf(" %d", &n);
        } while(n > 10000);

//Lectura de c condicionada a que esté entre 0 y 9 (digito valido)
        do
        {
                printf("\nIntroduce una cifra:");
                scanf(" %d", &c);
        } while(c > 9 || c < 0);

//Nota: mientras el número no sea cero quedan cifras por extraer
        bandera = 0;
        while(n > 0)
        {
        //se extrae la cifra
                cifra = n % 10;
                if (cifra == c)
                {
                        bandera = 1; //el número sí contiene la cifra
                }
        //se descarta la cifra del número
                n = n / 10;
        }
```

Código 4.18. (Cont.)

```
        if(bandera == 1)
        {
              printf("\nEl numero contiene la cifra");
        }
        else
        {
              printf("\nEl numero no contiene la cifra");
        }
        return 0;
  }
```

Código 4.19. Solución para P4.5. Ejercicio 3.

```
  #include <stdio.h>

  int main(void)
  {
  //declaración de variables
        int n; //numero entero introducido
        int c; //cifra a comprobar que está incluida en el entero n
        int cifra; //cifra extraída en cada iteración
        int bandera; //variable auxiliar para saber si contiene la cifra
        int i; //variable contador para bucle for
        int cp; //variable auxiliar para saber si es cuadrado perfecto

  //Lectura de n condicionada a que sea menor o igual que 10000
  // y, además, cuadrado perfecto
        cp = 0;
        do
        {
              printf("\nIntroduce un entero:");
              scanf(" %d",&n);

        //Comprobar si es cuadrado perfecto
              for(i = 1; i <= n; i++)
              {
                    if(i * i == n)
                    {
                          cp = 1; //es cuadrado perfecto
                    } // Final del if
              } // Final del for
        } while(cp == 0 || n > 10000); // Final del do-while
```

Código 4.19. (Cont.)

```
//Lectura de c condicionada a que esté entre 0 y 9 (digito valido)
      do
      {
            printf("\nIntroduce una cifra:");  scanf(" %d", &c);
      } while(c > 9 || c < 0);
//Nota: mientras el número no sea cero quedan cifras por extraer
      bandera = 0;
      while(n > 0)
      {
      //se extrae la cifra
            cifra = n % 10;
            if (cifra == c)
            {
                  bandera = 1; //el número sí contiene la cifra
            }
      //se descarta la cifra del número
            n = n / 10;
      }
      if(bandera == 1)
      {
            printf("\nEl numero contiene la cifra");
      }
      else
      {
            printf("\nEl numero no contiene la cifra");
      }
      return 0;
}
```

Código 4.20. Solución para P4.6. Ejercicio 1.

```
#include <stdio.h>
int main(void)
{
     char c; //carácter introducido
     int letras = 0; //cantidad de letras introducidas
     int numeros = 0; //cantidad de números introducidos

     printf("Lectura sucesiva de caracteres hasta introducir 'q'\n");
     do {
           scanf(" %c", &c);
           if(c >= 97 && c <= 122) { // es una letra según el código ASCII
                 letras = letras + 1;
           }
           if(c >= 48 && c <= 57) { //es un número según el código ASCII
                 numeros = numeros + 1;
           }
     }while(c != 'q');
```

Código 4.20. (Cont.)

```
      printf("Cantidad de letras : %d\n", letras);
      printf("Cantidad de numeros : %d\n", numeros);

      return 0;
 }
```

Estructuras de repetición (II)

5

5.1. Ejemplos sencillos resueltos

Ejemplo 5.1. *PARARSICERO. Uso de* ***break****.*

Ejemplo 5.2. *PARARSICERO_2. Uso de* ***break****.*

Ejemplo 5.3. *SUMARPOSITIVOS. Uso de* ***continue****.*

Ejemplo 5.4. *BREAKCONTINUE.*

5.2. Proyectos propuestos

Proyecto 5.1. *BENEFICIOEMPRESA*.

Proyecto 5.2. *FIBONACCI*.

Proyecto 5.3. *SERIE_E*.

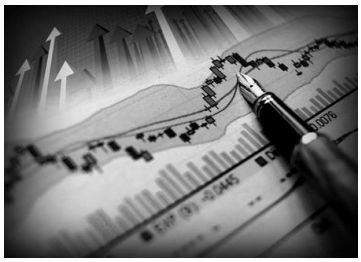
Proyecto 5.4. *SERIE_PI*.

Proyecto 5.5. *TAYLOR_COSENO*.

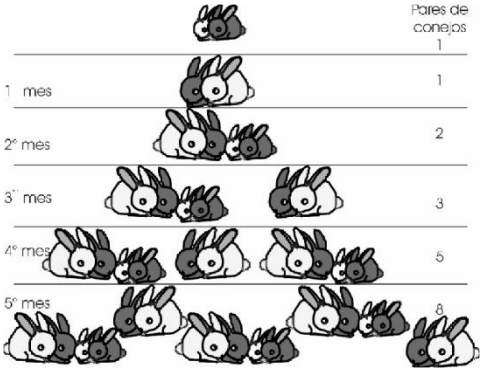
El objetivo de este capítulo es que el alumno consolide los conocimientos adquiridos sobre el uso de las estructuras de repetición **`for`**, **`while`** y **`do-while`** en el lenguaje C de programación. Se introduce, además, el uso de las sentencias de ruptura **`break`** y **`continue`** en este tipo de estructuras iteradas. Para logar alcanzar estos objetivos, inicialmente, comenzaremos con unos ejemplos sencillos que permiten al alumno conocer los fundamentos y uso de estas dos sentencias.

Al final de la práctica el alumno deberá ser capaz de resolver los siguientes ejercicios:

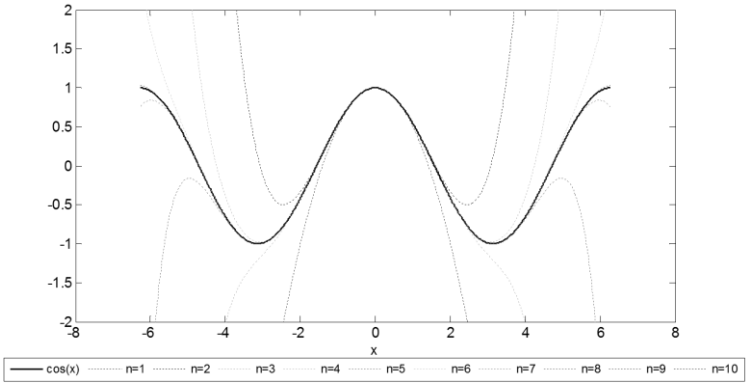
1. Un programa que permita calcular los beneficios de dos empresas ALPHA y BETA en función de su previsión de crecimiento en los próximos diez años. En estos diez años, ALPHA intentará cambiar su política empresarial para superar al líder del sector, que es la empresa BETA.



2. Un programa que permita calcular la serie de Fibonacci.



3. Obtener aproximaciones para los números e y π a través de series matemáticas.
4. Implementar la aproximación mediante series de Taylor para la función coseno.



5.1. EJEMPLOS SENCILLOS RESUELTOS

Antes de comenzar con los problemas que componen la práctica, es conveniente presentar algunos ejemplos iniciales que permitan al alumno familiarizarse en el uso de las sentencias **break** y **continue** dentro de estructuras de repetición o bucles.

Ejemplo 5.1. PARARSICERO

Creamos un nuevo proyecto denominado **PARARSICERO**. Vamos a desarrollar un programa que va a permitir introducir un total de 50 enteros por teclado e irá calculando iterativamente la suma total de los números introducidos para finalmente mostrar dicho resultado por pantalla.

En una primera aproximación planteamos (ver Código 5.1.) una solución mediante un bucle **for**.

Código 5.1. Programa planteado en Ejemplo 5.1.

```
#include <stdio.h>
int main(void) {
	int i; //contador, desde 1 hasta 50
	int suma = 0; //resultado de la suma
	int numero; //numero entero leido por teclado

	for(i = 1; i <= 50; i++) {
		printf("Introduce el numero %d de 50: ",i);
		scanf(" %d", &numero);
		suma = suma + numero;
	}
	printf("El resultado de la suma es: %d",suma);
	return 0;
}
```

A continuación, vamos a ampliar el programa anterior con una nueva exigencia: ahora nuestro programa deberá detenerse, antes de completar los 50 enteros, si el usuario introduce un cero por teclado: tenemos, pues, dos motivos que harán que el programa deje de solicitar del usuario valores por teclado: o porque éste ya ha introducido los 50 valores; o porque, en un momento del proceso de entrada de datos, el usuario ha introducido el valor 0.

Para ello, tenemos que introducir dentro del bucle una estructura de condición **if** que detecte cuándo el usuario introduce un número nulo y, después, forzar a que salga del bucle (es decir, 'romper' el bucle). Esto implica el uso de la sentencia **break**.

El nuevo programa podría quedar de la forma que se propone en Código 5.2. Como podemos observar, cuando la variable numero es igual a 0 se ejecuta la sentencia **break**, lo que provoca la interrupción de la ejecución iterativa de las sentencias incluidas en el bucle. Esto es, se abandona el bucle y se procede a mostrar por pantalla el valor de la variable suma (se ejecuta la primera sentencia después del bucle **for**).

Código 5.2. Programa planteado en Ejemplo 5.1. (segunda versión)

```
#include <stdio.h>
int main(void) {
      int i; //contador, desde 1 hasta 50
      int suma = 0; //resultado de la suma
      int numero; //numero entero leido por teclado

      for(i = 1; i <= 50; i++) {
            printf("Introduce el numero %d de 50: ", i);
            scanf(" %d",&numero);
            if(numero == 0) {
                  break;
            }
            suma = suma + numero;
      }
      printf("El resultado de la suma es: %d", suma);
      return 0;
}
```

> ***NOTA:*** Como se comentó en el Capítulo 3 de este libro, uno de los usos de la sentencia **break** es terminar un **case** en la sentencia **switch**. Otro uso es forzar la terminación inmediata de una iteración, independientemente de la condición que controle la ejecución del la misma.
>
> Si el compilador encuentra una sentencia **break** fuera de una estructura **switch**, o fuera de una estructura de iteración, terminará su proceso y dará un error de compilación.

Intente representar el flujograma del código recogido en Código 5.2.. Si lo hace, descubrirá que debe representar un flujograma… ¡¡que no respeta las reglas de la programación estructurada!! Se lo mostramos en la Figura 5.1. ¿Se le ocurre usted otro modo de representar el flujo de sentencias del programa arriba escrito?: verá, si lo intenta y no cambia la lógica del código propuesto, que no logra evitar dos salidas separadas de la estructura iterada.

Como puede ver en la Figura 5.1., el **break** insertado en nuestro programa introduce una violación a las reglas de la programación estructurada. La condición de permanencia en el bucle (`i <= 50`) cumple con las reglas; pero la segunda condición (`numero != 0`) ofrece una segunda salida al camino de iteración.

Hay muchos programadores que no aceptan de buena gana, por esta razón, el uso de las sentencias **break** y **continue**. Siempre hay un modo de llegar a una solución sin acudir a estas violaciones de las reglas. No se trata ahora de debatir sobre la conveniencia o no del uso de estas estructuras. Les mostramos ahora una solución que no incurriría en la violación de reglas. Y luego le mostraremos en el Ejemplo 5.2. un caso donde la sentencia **break** no incurre en ningún tipo de violación.

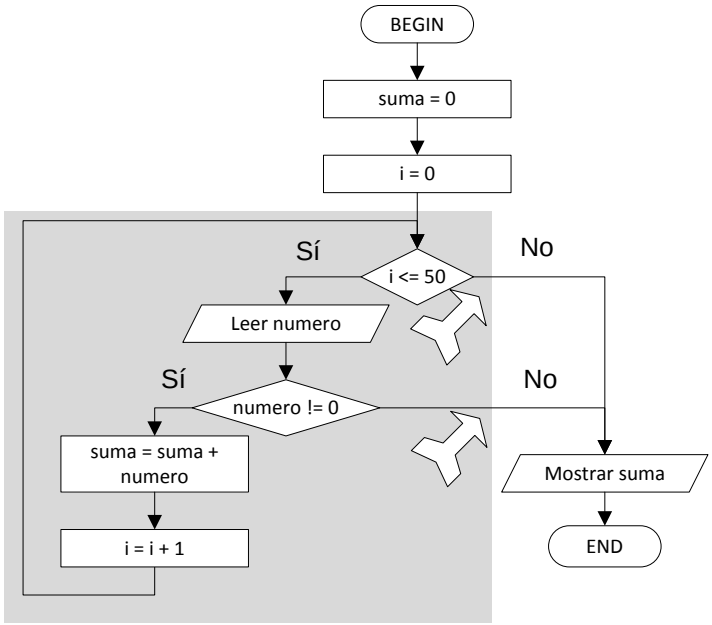


Figura 5.1. Flujograma del programa propuesto en el Ejercicio 5.1.

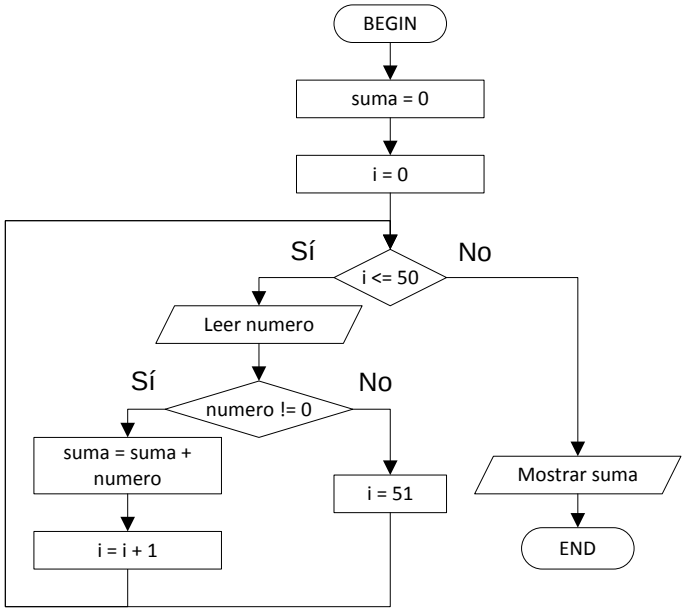


Figura 5.2. Otro posible flujograma que resuelve el problema propuesto en el Ejercicio 5.1. sin incurrir en violación de memoria. Esta solución NO es recomendable.

La solución primera mostrada en la Figura 5.2 puede ser válida: dependerá de si una vez fuera de la iteración, se necesita conocer el número de valores introducidos; de todas formas, siempre es peligroso modificar el valor de un índice dentro de una iteración: es muy poco recomendable. La solución propuesta en la Figura 5.3. no tiene restricciones. Pero vea que, en cualquier caso, necesita dentro de la iteración una estructura condicional: porque nuestro algoritmo tiene un bucle que

depende de DOS condiciones distintas; y donde la evaluación de ambas condiciones se halla en posiciones distintas dentro de la secuencias de sentencias iteradas.

En estos casos, muchos programadores sí ven justificada la posibilidad de utilizar la sentencia **break**, aunque suponga una violación de las reglas que la programación estructurada exige. No es, ni mucho menos, una cuestión cerrada.

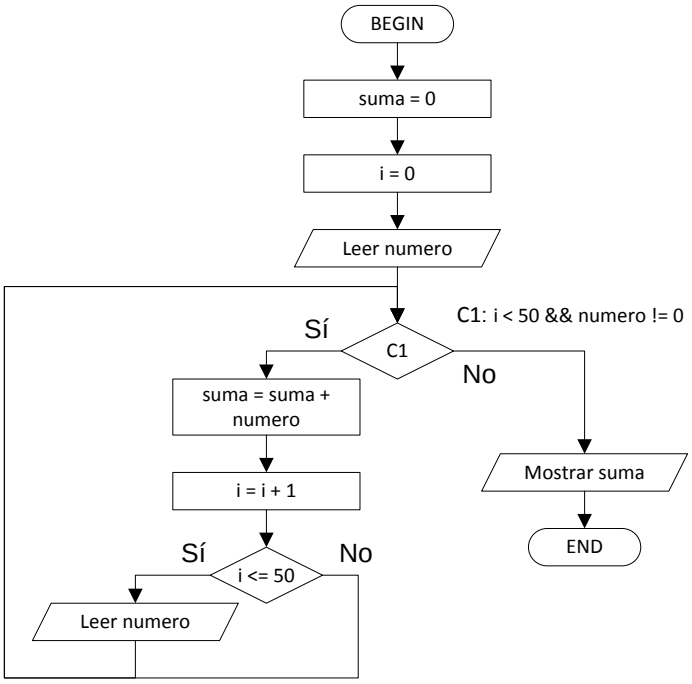


Figura 5.3. Otro flujograma que resuelve el Ejercicio 5.1. sin incurrir en violación de memoria.

Ejemplo 5.2. PARARSICERO_2

Veamos ahora un ejercicio muy parecido al anterior. Pero ahora no establecemos límite en el número de valores que el usuario puede introducir. Cuando termine el proceso de ingreso de valores mostrará la suma y terminará. Y para lograr indicar al programa que se ha finalizado las entradas, el usuario introducirá un cero.

Es decir, ahora el programa es muy parecido al anterior, pero hemos eliminado una de las dos condiciones: no hay una limitación respecto a la cantidad de valores que el usuario puede introducir. En ese caso, el flujograma quedará como el mostrado en la Figura 5.4. En Código 5.3. se recoge una implementación del algoritmo propuesto en esa figura.

Ahora, el uso de la sentencia **break** no rompe las reglas de la programación estructurada: con este código acabamos de implementar la estructura derivada llamada híbrida. Y si mira el flujograma propuesto en la Figura 5.4., tenemos una iteración que no es ni un **while** (porque hay sentencias que se ejecutan antes de llegar a la condición de permanencia), ni un **do-while** (porque hay sentencias que se ejecutan sólo después de haber evaluado como verdadera la condición de permanencia).

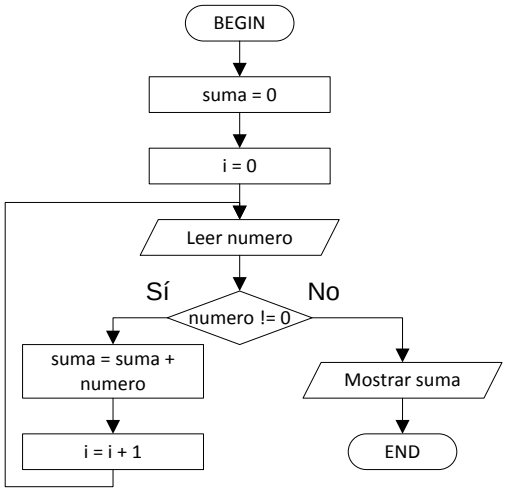


Figura 5.4. Flujograma del programa propuesto en el Ejercicio 5.2.

Código 5.3. Programa planteado en Ejemplo 5.2.

```
#include <stdio.h>
int main(void) {
      int suma = 0; //resultado de la suma
      int numero; //numero entero leído por teclado
      int i = 1;
      do {
            printf("Introduce el numero %d de la serie a sumar: ",i);
            scanf(" %d",&numero);
            if(numero == 0) {
                  break;
            }
            suma = suma + numero;
            i++;
      }while(1); // En C 1 es siempre verdadero: sólo 0 es falso.
      printf("La suma de los %d enteros introducidos es: %d", i, suma);
      return 0;
}
```

NOTA: En C, como ya hemos señalado en otros momentos, todo lo que es distinto de cero es verdadero; y el valor cero es falso. Si una estructura de control tiene como condición un literal distinto de cero (**while**(1), por ejemplo), entonces, por definición, esa condición jamás podrá dejar de ser cierta y por lo tanto habremos quedado atrapados en ella: tendremos un bucle infinito. Por lo tanto, siempre que se crea una estructura de iteración donde la condición es un literal verdadero, deberemos insertar, entre todas las sentencias iteradas, una sentencia **break** condicionada. Es así como podemos implementar las estructuras **híbridas**: estructuras de iteración, donde la condición de permanencia no está previa a la primera sentencia, ni al final, después de la última, sino intercalada entre dos grupos de sentencias.

Ejemplo 5.3. SUMARPOSITIVOS

Creamos un nuevo proyecto denominado **SUMARPOSITIVOS**. Este programa va a permitir introducir un total de 50 enteros por teclado y va calculando iterativamente la suma total de los números positivos introducidos para finalmente mostrar dicho resultado por pantalla. En este caso, y atendiendo a los conocimientos adquiridos en la sesión anterior, planteamos la solución recogida en Código 5.4.

Código 5.4. Programa planteado en Ejemplo 5.3.

```
#include <stdio.h>
int main(void)
{
	int i; //contador, desde 1 hasta 50
	int suma = 0; //resultado de la suma
	int numero; //numero entero leído por teclado

	for(i = 1; i <= 50; i++)
	{
		printf("Introduce el numero %d de 50: ",i);
		scanf(" %d", &numero);
		if(numero > 0)
		{
			suma = suma + numero;
		}
	}
	printf("El resultado de la suma es: %d", suma);
	return 0;
}
```

donde se ha empleado una estructura **if** dentro del bucle de tal forma que únicamente se realizará la operación aditiva `suma = suma + numero` cuando el valor de `numero` sea mayor que 0. Otra forma hubiera sido utilizar el operador interrogante – dos puntos (vea la línea /*01*/ en Código 5.5., donde se propone otra redacción de la estructura **for**).

Código 5.5. Modificación de la estructura **for** propuesta en Código 5.4..

```
	for(i = 1; i <= 50; i++)
	{
		printf("Introduce el numero %d de 50: ",i);
		scanf(" %d", &numero);
/*01*/		suma = numero > 0 ? suma + numero : suma;
	}
```

Ahora planteamos otra solución basada en la sentencia **continue**. En vez de forzar la salida, la sentencia **continue** fuerza el salto a la siguiente iteración del bucle, por lo que no ejecuta el código

que resta hasta el final del bloque de código iterado y evalúa directamente la condición de permanencia en la iteración. Una solución del ejercicio usando **continue** podría ser la propuesta en Código 5.6.

Código 5.6. Programa planteado en Ejemplo 5.3. usando la sentencia **continue**.

```
     #include <stdio.h>
     int main(void)
     {
           int i; //contador, desde 1 hasta 50
           int suma = 0; //resultado de la suma
           int numero; //numero entero leido por teclado

/*01*/     for(i = 1; i <= 50; i++)
           {
                 printf("Introduce el numero %d de 50: ",i);
                 scanf(" %d", &numero);
/*02*/           if(numero <= 0)
                 {
/*03*/                   continue;
                 }
/*04*/           suma = suma + numero;
           }
           printf("El resultado de la suma es: %d", suma);
           return 0;
     }
```

Podemos observar que hemos cambiado la condición que controlaba inicialmente la ejecución del bucle. En este caso, si el valor de la variable numero es menor que 0 (línea /*02*/) se ejecuta la sentencia **continue** (línea /*03*/), y con ello se dejaría de ejecutar la sentencia de la línea /*04*/ para comenzar con la siguiente iteración de la estructura **for** (línea /*01*/). Por ejemplo, si los 50 números introducidos fuesen negativos el valor final de la variable suma sería igual a 0.

De nuevo podemos plantearnos cómo sería el flujograma del programa así redactado con la sentencia **continue**. Lo mostramos en la Figura 5.5.

Fíjese que, más allá de la condición que permite la ejecución de la sentencia **continue** (línea /*02*/), nos encontramos con una última sentencia no condicionada (línea /*01*/: sentencia i++): porque en la estructura **for** se tiene un espacio para indicar una o varias sentencias que se ejecutan después de terminar el bloque de código iterado. Este es una diferencia de comportamiento, no pequeña, entre una iteración escrita mediante una estructura **while** y otra implementada con una estructura **for**: si en el bloque iterado se encuentra una sentencia **continue**, siempre podrá haber, en la iteración **for**, una última sentencia a ejecutar antes de evaluar la condición de permanencia; no ocurre así con la estructura **while**. Intente reescribir la iteración de Código 5.6. utilizando una estructura **while**: ¿dónde coloca el incremento de valor de la variable i?

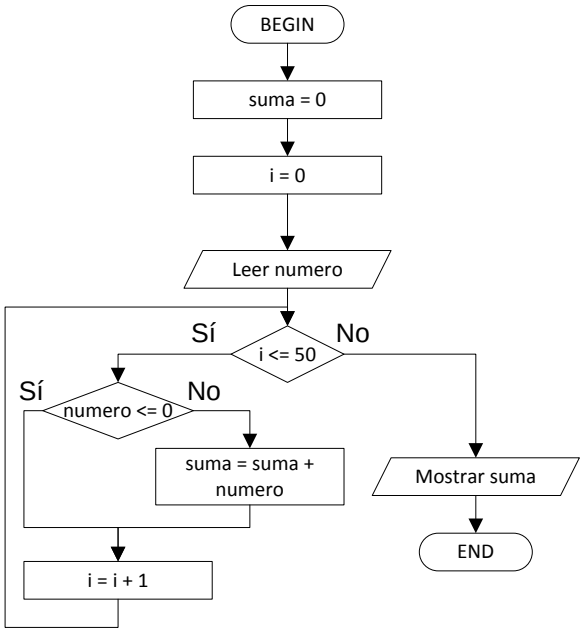


Figura 5.5. Flujograma del programa propuesto en el Ejercicio 5.2.

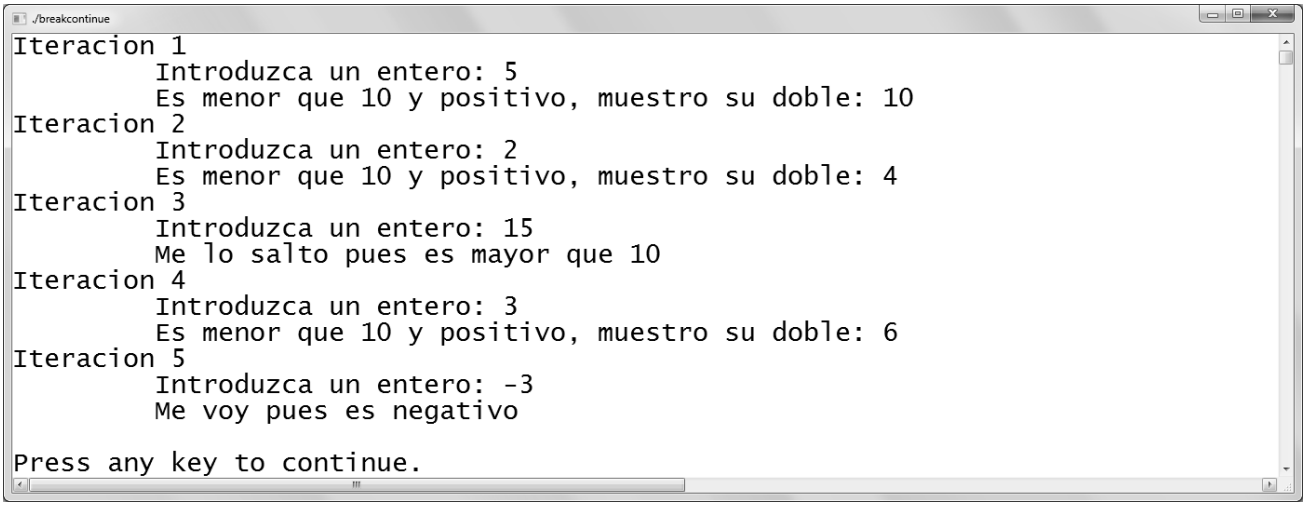
Ejemplo 5.4. BREAKCONTINUE

Por último, a modo de resumen, se muestra el programa de Código 5.7.

Código 5.7. Programa planteado en Ejemplo 5.4.

```
#include <stdio.h>
int main(void) {
      int n; //numero introducido por teclado
      int i = 1; // valor inicial del contador: 1
      while(i < 50){
            printf("Iteracion %d",i);
            printf("\n\t Introduzca un entero: ");   scanf(" %d", &n);
            i++;
            if(n < 0) { //si n es negativo, salgo del bucle
                  printf("\tMe voy pues es negativo \n");
                  break;
            }
            if (n > 10) { //si n es mayor que 10, no muestro su doble
                  printf("\tMe lo salto pues es mayor que 10 \n");
                  continue; //va a la siguiente iteracion
            }
            printf("\tPositivo menor que 10: muestro su doble:%d\n", 2*n);
      }
      return 0;
}
```

Este programa permite introducir —mientras no se dé un valor negativo— hasta 50 enteros. Cada vez que el valor positivo introducido es menor que 10, se muestra por pantalla su valor doble. A continuación se muestra el resultado obtenido en el caso de introducir progresivamente los siguientes números: 5, 2, 15, 3 y -3. Tal y como se ha indicado anteriormente, se muestra su doble para 5, 2 y 3, mientras que no se muestra para 15. La ejecución del bucle finaliza al introducir -3.



No intente comprender la utilidad de este programa: es un ejemplo meramente académico, que muestra el comportamiento de las dos sentencias de ruptura dentro de una iteración: **`break`** y **`continue`**. Si comprende su comportamiento, y si ha visto el peligro que estas sentencias traen por su gran facilidad para violar las reglas de programación estructurada, entonces podrá aprender a decidir cuándo convendrá usarlas y cuando es mejor no acudir a ellas en su código.

NOTA: En MISRA-2004, "*Guidelines for the use of the C language in critical systems*" se recoge una colección de recomendaciones de buena práctica de programación. Entre otras muchas, pueden leerse las siguientes:

*Rule 14.4 (required): The **`goto`** statement shall not be used.*

*Rule 14.5 (required): The **`continue`** statement shall not be used.*

*Rule 14.6 (required): For any iteration statement there shall be at most one **`break`** statement used for loop termination.*

Y, a continuación, el documento aclara:

*These rules are in the interests of good structured programming. One **`break`** statement is allowed in a loop since this allows, for example, for dual outcome loops or for optimal coding.*

La prohibición del uso de la sentencia **`goto`** es universalmente aceptada. Esta referencia citada prohíbe las sentencias **`continue`** con la misma contundencia que el uso de **`goto`**. No es una opinión tan extendida, al menos con la severidad de esta obra. En estas prácticas mostraremos el uso de estas sentencias. Pero debe tener sumo cuidado con ellas. Como ya hemos señalado, es muy fácil, al usarlas, violar las reglas básicas de la programación estructurada.

5.2. PROYECTOS PROPUESTOS

Proyecto 5.1. BENEFICIOEMPRESA

➔ *P5.1. Ejercicio 1.*

Creamos un nuevo proyecto llamado **BENEFICIOEMPRESA.** En él desarrollaremos un pequeño programa que nos ayude a resolver el problema que, a continuación, se plantea.

En el año actual, una empresa *ALPHA* del sector de Defensa obtiene un beneficio inicial de **130.00** Millones de euros mientras que la empresa *BETA* (líder del sector nacional) obtiene **200.00** Millones de euros. Esta empresa *BETA* tiene una previsión de incrementar su beneficio de manera geométrica a razón de **3.5%** anual en los próximos diez años. La empresa *ALPHA* quiere superar al líder de su sector a toda costa y, para ello, encarga un estudio a su departamento de dirección financiera. A partir de los datos históricos de la empresa *ALPHA* y un análisis del entorno, este departamento decide ampliar su mercado en el exterior con una política arriesgada y realiza una previsión de que este cambio conllevará un incremento geométrico en su beneficio neto a razón de un **10%** cada año.

Desarrolle un programa, que utilice estructuras de repetición, que indique si la empresa *ALPHA* logrará superar los beneficios de la empresa *BETA* con esta propuesta antes de los próximos diez años.

- En caso afirmativo, se mostrará el siguiente mensaje `"El líder del sector en el año XXXX es la empresa ALPHA con una ventaja de YYYY MILLONES sobre la empresa BETA"`, donde XXX denota el año donde se consigue el objetivo e YYYY representa la diferencia entre el beneficio neto de ambas empresa. Además, mostrará por pantalla el beneficio (en millones de euros) de ambas empresas para dicho año.
- En caso contrario, deberá indicar por pantalla el mensaje `"La propuesta no consigue el objetivo de la empresa ALPHA"` y mostrar por pantalla el beneficio (en millones de euros) de ambas empresas para el año final del periodo estudiado de 10 años a partir del actual.

Emplee dos decimales para visualizar los beneficios de cada empresa en millones de euros.

SUGERENCIAS:

- Intente utilizar sentencias **break** para la solución de este ejercicio.
- La expresión para el cálculo del beneficio anual es:

  ```
  beneficio = beneficio * incrementoAnual
  ```

  donde `incrementoAnual` expresa el tanto por ciento de previsión de incremento anual.

Proyecto 5.2. FIBONACCI

➔ *P5.2. Ejercicio 1.*

Cree un nuevo proyecto llamado **FIBONACCI** que permita mostrar por pantalla los 30 primeros números de la sucesión de Fibonacci que se define de la siguiente manera:

$$x_0 = 1$$
$$x_1 = 1$$
$$x_i = x_{i-1} + x_{i-2}, \text{ para } i \geq 2$$

➔ *P5.2. Ejercicio 2.*

Seguidamente, tenemos que crear un nuevo proyecto llamado **FIBONACCI2** cuyo programa muestre los números de Fibonacci múltiplos de 3 y menores que 30.

Proyecto 5.3. SERIE_E

➔ *P5.3. Ejercicio 1.*

Cree un nuevo proyecto llamado **SERIE_E** donde desarrolle un programa que permita al usuario calcular el valor del número *e* utilizando 10^6 términos de la siguiente serie matemática:

$$e = \sum_{n=0}^{\infty} \frac{1}{n!}$$

NOTA: Tenga en cuenta que el número de términos (10^6) es muy grande y puede afectar al cálculo del factorial: por ejemplo, el factorial de 14 ya es mayor 2^{32}. Le sugerimos que no intente calcular los sucesivos factoriales y, luego, obtener su inversa, sino que calcule directamente los sucesivos inversos de los sucesivos factoriales.

Proyecto 5.4. SERIE_PI

➔ *P5.4. Ejercicio 1.*

Cree un nuevo proyecto llamado **SERIE_PI** que permita al usuario calcular una aproximación para el número π utilizando la siguiente serie matemática propuesta por Leibniz:

$$\frac{\pi}{4} = \sum_{n=0}^{\infty} \frac{(-1)^n}{2n+1} = 1 - \frac{1}{3} + \frac{1}{5} - \ \ldots$$

NOTA: Como no podemos realizar una suma de infinitos términos, debemos considerar un número máximo suficiente. Pruebe con distintos valores y verifique que a medida que se incrementa el número de términos se obtiene una mejor aproximación.

Proyecto 5.5. TAYLOR_COSENO

➔ *P5.5. Ejercicio 1.*

Debemos crear un nuevo proyecto denominado **TAYLOR_COSENO.** Es conocido que la serie de Taylor para la función coseno viene dada por la siguiente ecuación:

$$cos\ x = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} \cdot\ x^{2 \cdot n}, \forall x$$

donde x es un ángulo expresado en **radianes**.

El programa debe calcular el valor del coseno de un ángulo utilizando el desarrollo en serie de Taylor con un error inferior al preestablecido por el usuario. Para ello, el programa debe ir incrementando el número de términos de la serie hasta conseguir la *convergencia*. *¿Qué entendemos por convergencia?* Entendemos que se ha alcanzado la convergencia cuando el valor del sumando n-ésimo es menor que el valor de error introducido previamente por el usuario. Por otro lado, el número máximo de términos es también previamente introducido. Con todo ello, la ejecución finalizará cuando se sobrepase el número máximo de iteraciones o cuando se alcance la convergencia. Al finalizar, además del valor calculado para el coseno de x, el programa debe mostrar el número de términos que se han utilizado.

De esta forma, el programa debe:

1. Solicitar el valor de x en grados, obligando a introducir un número entre -180 y 180. Convertirlo a radianes (recuerde: 180 grados son π radianes). Utilizar variables tipo **`double`**.
2. Solicitar el número máximo de términos.
3. Solicitar el error o margen o diferencia de convergencia.
4. Calcular iterativamente la aproximación de Taylor incrementando el número de términos hasta que:
   - o se alcance la convergencia (valor n-ésimo de la serie es menor que el margen de convergencia definido por el usuario),
   - o se supere el número máximo de términos.
5. Mostrar por pantalla
   - El resultado de la aproximación.
   - El número de términos.
   - El error cometido (valor n-ésimo de la serie).

SOLUCIONES A LOS EJERCICIOS DE LOS PROYECTOS PROPUESTOS

Código 5.8. Solución para P5.1. Ejercicio 1.

```
#include <stdio.h>
int main(void)
{
      double beneficioA = 130; // beneficios iniciales de A
      double beneficioB = 200; // beneficios iniciales de B
      int comienzo;            // anyo de comienzo del estudio
      int fin;                 // anyo de final del estudio
      int i;                   // contador de años
      double incA = 1.10;      // incremento anual de A
      double incB = 1.035;     // incremento anual de B
      int objetivo = 0;        // Bandera Valdrá 1 si B supera a A

// introduccion del anyo de inicio de estudio
      printf("Indique el anyo de inicio del estudio ... ");
      scanf(" %d", &comienzo);
      fin = comienzo + 10;   // Es el plazo establecido: 10 años.

      printf("BENEFICIO DE EMPRESAS.......\n");
//bucle para calcular los beneficios anuales previstos para cada empresa
      for(i = comienzo + 1 ; i <= fin ; i++)
      {
            beneficioA = beneficioA * incA;
            beneficioB = beneficioB * incB;
            if(beneficioA > beneficioB)
            {
                  printf("El lider del sector en %d ", i);
                  printf("es la empresa ALPHA con una ventaja de ");
                  printf("%.2lf Millones ",beneficioA - beneficioB);
                  printf("sobre la empresa BETA");
                  printf("\nBeneficio de ALPHA: %.2lf", beneficioA);
                  printf("\nBeneficio de BETA: %.2lf", beneficioB);
                  objetivo = 1;
                  break;
            }
      }
// en caso de que NO se haya cumplido el objetivo...
// ...el valor de la variable objetivo nos informa sobre ello:
      if(objetivo == 0)
      {
            printf("La propuesta no consigue el objetivo de ALPHA\n");
            printf("Beneficio de ALPHA: %.2lf \n", beneficioA);
            printf("Beneficio de BETA: %.2lf \n", beneficioB);
      }
      return 0;
}
```

Código 5.9. Solución para P5.2. Ejercicio 1.

```
#include <stdio.h>
int main(void) {
	int N = 30; // primeros N numeros de la serie de fibonacci
	int n = 0;  // cantidad de terminos de la serie impresos en pantalla
	int x1 = 1; // primer termino anterior al actual
	int x2 = 1; // segundo termino anterior al actual
	int x = 1;  // termino actual

	printf("%d primeros elementos de la serie de Fibonacci:\n", N);

	while(n != N)		{
		n++;
		if(n >= 3)		{
			x = x1 + x2;
			x1 = x2;
			x2 = x;
		}
		printf("%d\n", x);
	}

	return 0;
}
```

Código 5.10. Segunda solución para P5.2. Ejercicio 1. (con la sentencia **continue**)

```
#include <stdio.h>
int main(void) {
	int N = 30; // primeros N numeros de la serie de fibonacci
	int n = 0;  // cantidad de terminos de la serie impresos en pantalla
	int x1 = 1; // primer termino anterior al actual
	int x2 = 1; // segundo termino anterior al actual
	int x = 1;  // termino actual

	printf("%d primeros elementos de la serie de Fibonacci: \n", N);

	while(n != N)		{
		n++;
		if(n < 3)		{
			printf("%d\n", x);
			continue;
		}
		x = x1 + x2;
		x1 = x2;
		x2 = x;
		printf("%d\n", x);
	}

	return 0;
}
```

Código 5.11. Solución para P5.2. Ejercicio 2.

```
#include <stdio.h>
int main(void) {
	int x1 = 1; // primer termino anterior al actual
	int x2 = 1; // segundo termino anterior al actual
	int x = 1;  // termino actual

	while(x < 30)
	{
		x = x1 + x2;
		x1 = x2;
		x2 = x;
		if(x % 3 == 0)
		{
			printf("%d\n", x);
		}
	}

	return 0;
}
```

NOTA: Como ya se dijo en un capítulo previo, al concatenar con operadores lógicos (AND, OR, ó NOT) distintas expresiones relacionales que se evalúan como VERDADERO ó FALSO, se aconseja encerrar cada expresión entre paréntesis.

Código 5.12. Solución para P5.3. Ejercicio 1.

```
#include <stdio.h>
int main(void)
{
	int N = 1e6;		// numero de terminos de la serie
	double e = 1;		// valor inicial de la aproximación
	double i_f = 1;		// inverso factorialiteración:
				// cálculo acumulado de iteración en iteración.
	int n; //contador

	for(n = 1; n < N ; n++)
	{
		i_f = i_f * (1.0 / n);
		e = e + i_f;
	}
	printf("El valor aproximado de e es: %lf", e);
	return 0;
}
```

Código 5.13. Solución para P5.4. Ejercicio 1.

```
      #include <stdio.h>
      int main(void)
      {
            double pi4 = 1;         // variable para la aproximacion de pi/4
            int signo = +1.0;       // signo positivo o negativo
            double elemento;        // termino n-esimo del sumatorio
            int N;                  // numero maximo de terminos
            int n = 1;              // contador de iteraciones

            printf("Maximo numero de terminos... ");       scanf(" %d",&N);
            while(n <= N)
            {
 /*01*/             if(n % 2 == 0)
                  {
                        signo = +1.0;
                  }
                  else
                  {
                        signo = -1.0;
 /*02*/             }
                  elemento = 1.0 / (2.0 * n + 1);
                  pi4 = pi4 + signo * elemento;
                  n++;
            }
            printf("Valor aproximado de pi con %d terminos: %lf", n, 4 * pi4);
            return 0;
 }
```

NOTA: Todo el código entre las líneas marcadas como /*01*/ y /*02*/ podría dejarse más breve con la siguiente sentencia:

```
signo = -signo;
```

asignando inicialmente a la variable `signo` el valor -1.0.

Código 5.14. Solución para P5.5. Ejercicio 1.

```
#include <stdio.h>
#include <math.h>
int main(void)
{
      double x_g;       // angulo en grados
      double x_r;       // angulo en radianes
      double error;     // valor de error, margen o diferencia
      double sumando;   // cada uno de los sumandos de la serie de Taylor
      double cos_x;     // valor del coseno de x
```

Código 5.14. (Cont.).

```
      double i_f;       // incremento en cada iteracion
      int signo;        // signo positivo o negativo
      int M;            // numero maximo de terminos
      int n;            // contador para primer bucle (iteraciones)

      //introducir por teclado el valor de x en grados
      do
      {
            printf("Introduzca x en grados (entre -180 y 180)...");
            scanf(" %lf", &x_g);
      }
      while(x_g > 180.0 || x_g < -180.0);

      x_r = x_g * (M_PI / 180); //conversion a radianes
      // M_PI: constante definida en math.h que codifica el valor de pi.

      //introducir por teclado el valor para M
      printf("Introduzca el numero maximo de terminos...");
      scanf(" %d", &M);

      //introducir por teclado el error maximo para la convergencia
      printf("Introduzca el error deseado...");
      scanf(" %lf", &error);

      //bucle para incrementar los terminos de la serie de Taylor
      for(signo = -1.0 , n = 1 , i_f = 1.0 , cos_x = 1.0 ;
          n <= M ;  n++ , signo = -signo)
      {
            i_f *= 1.0 / (4 * n * n - 2 * n);
            sumando = i_f * pow(x_r, 2 * n);

            if(sumando > error)    {     cos_x += signo*sumando;       }
            else                   {     break;                        }
      }

      printf("El valor aproximado de coseno de %lf es: %lf", x_g, cos_x);
      printf("\nNumero de terminos utilizados: %d (Maximo %d)", n, M);
      printf("\nError cometido: %lf" , sumando);

      return 0;
}
```

Arrays numéricos (I)

6

6.1. Ejemplos sencillos resueltos

Ejemplo 6.1. *VENTAS.*

6.2. Proyectos propuestos

Proyecto 6.1. *CUATROVECTORES*.

Proyecto 6.2. *VECTORALEATORIO*.

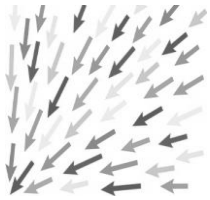
Proyecto 6.3. *BURBUJA*.

Proyecto 6.4. *CALCULADORAVECTORES*. *Ejercicio adicional*.

En las prácticas anteriores se han estudiado los diferentes tipos de datos primitivos del lenguaje C, usados para codificar valores enteros, reales o caracteres. Sin embargo, en muchas situaciones se precisa procesar una colección de valores que están relacionados entre sí, como por ejemplo un registro de medidas de presión, una matriz de asignación de recursos, etc. Procesar esos conjuntos de datos mediante variables independientes resultaría extremadamente complejo y tedioso; por ello la mayoría de los lenguajes de programación incluyen una serie de nuevos tipos de dato que permiten manipular colecciones de variables de un mismo tipo que codifican conjuntos de valores: un valor en cada una de las variables de la colección. Estos nuevos tipos de datos se llaman "**arrays**". Habitualmente nos referimos a ellos, en castellano, hablando de **vectores** (array unidimensional) y de **matrices** (array multidimensional).

El objetivo de este capítulo es lograr que el alumno aprenda a utilizar los vectores. Como es habitual, el enunciado comienza con unos ejemplos sencillos que permiten al alumno conocer los fundamentos de los vectores. A continuación, el alumno deberá ser capaz de realizar:

Una serie de sencillos programas que permitan operar con vectores: invertir el orden de sus elementos, obtener el máximo y el mínimo, calcular los productos escalar y vectorial, etc.



Un programa que permita, primero, inicializar un vector con números enteros aleatorios y, a continuación, ordenar sus elementos de menor a mayor utilizando el método o algoritmo de la burbuja.

Al final del documento, en un anexo, se recoge una breve documentación con los conceptos mínimos necesarios para poder generar valores aleatorios en un programa escrito en C.

6.1. EJEMPLOS SENCILLOS RESUELTOS

Antes de comenzar con los problemas que componen esta práctica, será conveniente realizar un par de ejemplos iniciales que permitan al alumno introducirse en el uso de los vectores.

Ejemplo 6.1. VENTAS

Cree un nuevo proyecto denominado **VENTAS**. Considere un departamento de ventas compuesto por 10 agentes comerciales. Un agente comercial obtiene una comisión sobre sus ventas si consigue vender más del promedio de las ventas conseguidas por todos los agentes. Se necesita un programa que sea capaz de leer las ventas de los 10 agentes comerciales, calcule el promedio de todas las ventas y, finalmente, a partir del promedio muestre por pantalla sólo aquellos agentes que consiguen la comisión (sus ventas son superiores al promedio).

La implementación con tipos de datos simples implica que necesitaríamos 10 variables distintas para almacenar los valores de las ventas, una por cada agente comercial (por ejemplo, `ventas1`, `ventas2`, …, `ventas10`). Hacer un programa que maneje cada una de esas variables de forma independiente exige una cantidad enorme de código, muy repetitivo, que no podríamos resumir con estructuras de repetición porque estaríamos trabajando siempre sobre variables distintas. Además, si en un momento dado debiéramos modificar el programa para ampliar el número de agentes (aunque sólo fuera a uno más), deberíamos hacer cambios de envergadura, cuando en realidad la algoritmia del problema sería sustancialmente la misma.

Para solucionar este tipo de problemas, los lenguajes de programación disponen de **arrays**. *Un array es una colección de N variables, todas ellas del mismo tipo de dato, todas ellas ubicadas en memoria de forma consecutiva una a continuación de la anterior, y todas ellas referenciadas con un nombre común: podremos distinguir una de otra porque, dentro del array, cada una de ellas se identifica y distingue entre todas las demás mediante un índice*. Para nuestro problema bajo estudio, y considerando que las ventas se almacenan en datos de tipo **`double`**, se debe declarar un array formado por 10 elementos utilizando la siguiente sentencia:

```
double ventas[10];
```

donde, tal y como se ilustra en la Figura 6.1, disponemos de un total de 10 variables de tipo **`double`** almacenados en un array denominado `ventas`. Este tipo de array se le suele denominar *array unidimensional* o *vector*. Los valores almacenados en el vector `ventas` son independientes entre sí.

Para hacer referencia a un elemento del vector, usaremos el nombre o identificador del vector seguido por un número (índice) entre corchetes, que indicará su posición dentro del array. El primer elemento del vector es `ventas[0]` y el último es `ventas[9]`. Podemos asignar a cada uno de los comerciales una de esas variables.

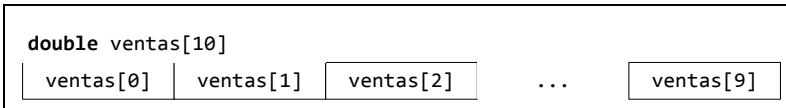


Figura 6.1. Esquema de la memoria donde han quedado creadas las 10 variables llamadas `ventas` y declaradas como **`double`** mediante un array.

NOTA: Definición de array: Colección de variables (1) del mismo tipo; (2) con el mismo nombre; (3) ubicados en memoria de forma consecutiva; (4) diferenciadas unas de otras y referenciadas mediante índices.

NOTA: El Estándar C99 introdujo una novedad importante en la declaración de arrays: la posibilidad de que la dimensión se exprese no como un literal (**`double`** `a[10];`) sino con una variable o una expresión: **`short`** `N = 10;` **`double`** `a[N];`

Este cambio no es accidental, y permite plantear soluciones de programación con un enfoque distinto al que permitía el ANSI 90 o los estándares C89 ó C90.

Quizá sea conveniente utilizar literales en la declaración de los arrays siempre que esa dimensión sea conocida en tiempo de compilación, y declarar los arrays con dimensiones expresadas con una variable cuando esas dimensiones puedan cambiar de una ejecución del programa a otra ejecución. Pero eso es únicamente una recomendación.

NOTA: Si la longitud del array es N = 10, éste comienza siempre en la posición indicada por el índice 0 y termina en la posición del índice N – 1. No hay excepciones a esta regla.

Supuesta la declaración **`double`** `array[10];` (ó **`short`** `N = 10;` **`double`** `array[N];`), acceder a la variable `array[10]` (ó `array[N]`) es siempre un error que no se detecta fácilmente. El compilador lo dará siempre por bueno, pero luego, en tiempo de ejecución, al realizar la violación de memoria, el comportamiento del programa será impredecible.

Una vez que tenemos el vector para almacenar las ventas, necesitamos recorrer los distintos elementos que lo componen. Para ello se puede hacer uso de un índice (por ejemplo, la variable entera `i`) que vaya desde 0 hasta `N - 1` (desde 0 hasta 9). Así, por ejemplo el código recogido en Código 6.1. nos permite inicializar todos los elementos del vector `ventas` a 0.

Como puede en la línea de código señalada con `/*01*/` podemos trabajar de forma independiente con cada uno de los valores del array: desde `ventas[0]` hasta `ventas[9]` en nuestro caso. Es importante que no pretenda manejar todos los elementos del array como si fueran una sola variable. Por ejemplo, no es correcto, si queremos asignar el valor 0 a todos los elementos del array, escribir `ventas = 0;`: eso daría error de compilación: más adelante sabrá por qué.

Código 6.1. Inicializa a cero todos los valores del array ventas, declarado como de tipo **double**.

```
     #include <stdio.h>
     #define _N 10 //número de agentes comerciales

     int main(void)
     {
           int i; //indice desde 0 hasta N
           double ventas[_N]; //declaracion del vector ventas

           for(i = 0; i < _N; i++)
           {
/*01*/           ventas[i] = 0; // asigna valor 0 al elemento i-esimo del array
           }

           return 0; //el programa finaliza correctamente
}
```

NOTA: En Código 6.1., _N se ha declarado como una macro usando la directiva **define** antes de la función main. Al declarar una macro no se reserva memoria para el valor: al compilar se sustituye tal cual un valor por otro (en este caso _N por 10). Tenga cuidado: como en toda directiva de preprocesador (comienzan con #), no se pone punto y coma al final.

En este caso, con la directiva **define** creamos un nuevo literal. El compilador, previo al trabajo de compilar, rastrea todo el documento más debajo de la directiva **define** en busca de la cadena de texto _N; y cada vez que la haya, cambia esa cadena e inserta un 10.

Las reglas para creación de nombres para las macros son las mismas que las reglas para la creación de cualquier identificador en C. En este manual se asume como criterio general, que todas las macros inician su nombre con el carácter guion bajo. Ese criterio es opcional.

Es más habitual inicializar los array mediante la asignación directa de valores a los elementos del vector, poniendo entre llaves y separados por comas los valores iniciales de todos los elementos del vector. La dimensión del array puede venir indicada mediante una variable entera, o mediante un literal. El programador también puede declarar el array sin dimensionarlo: en ese caso es necesario asignar sus valores en la declaración. Por último, si se introducen menos valores que los indicados en la dimensión del array, entonces los restantes son asignados al valor cero. Sin embargo, si introduce más valores de los indicados en la dimensión, el comportamiento del programa será incierto. Así, estas siguientes líneas de código crean vectores con los mismos valores iniciales:

```
double ventas1[] = {0,0,0,0,0,0,0,0,0,0};
double ventas2[10] = {0,0,0,0,0,0,0,0,0,0};
double ventas3[10] = {0};
short N = 10; double ventas4[N] = {0,0,0,0,0,0,0,0,0,0};
```

Y de estas líneas de código, hay dos que son erróneas:

```
double ventas5[];
double ventas6[10] = {0,0,0};
double ventas7[10] = {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
```

¿Sabe usted identificar las líneas erróneas y justificar por qué lo son?

Finalmente, el código propuesto para nuestro programa podría ser el propuesto en Código 6.2.

Código 6.2. Programa propuesto en Ejemplo 6.1.

```
#include <stdio.h>
#define _N 10 //numero de agentes comerciales

int main(void)
{
      int i;                  // indice desde 0 hasta N
      double ventas[_N];      // declaracion del vector ventas
      double suma = 0;        // suma de ventas
      double media;

//introduccion de ventas de los N agentes comerciales

      for(i = 0 ; i < _N ; i++){
      //introducción de ventas del agente i-esimo
            printf("Introduzca las ventas del agente %d...",i);
            scanf(" %lf", &ventas[i]);
      }

//calculo la media de las ventas
      for(i = 0 ; i < _N ; i++){    // Primero calculamos la suma...
            suma += ventas[i];
      }
      media = suma / _N;            // ... y luego calculamos la media

//determinar si el agente i-esimo tiene unas ventas superiores a la media
      for(i = 0 ; i < _N ; i++){
            if(ventas[i] > media){
                  printf("El agente %d tiene comision \n", i);
            }
      }
      return 0; //el programa finaliza correctamente
}
```

6.2. PROYECTOS PROPUESTOS

Proyecto 6.1. CUATROVECTORES

Cree un proyecto denominado **CUATROVECTORES**.

El programa que desarrollará en este ejercicio debe declarar cuatro arrays (`vector1, vector2, vector3` y `vector4`) de enteros con una misma dimensión, por ejemplo 5, (indique el tamaño mediante una directiva **`define`**). Se deben realizar las siguientes operaciones con los cuatro arrays numéricos:

- Al primero (`vector1`) se le asignarán los valores por entrada de teclado.
- El segundo (`vector2`) será copia del primero.
- El tercero (`vector3`) será copia en orden inverso del primero: el primer elemento de `vector3` será el último de `vector1`, el segundo será el penúltimo, etc.
- El cuarto array (`vector4`) será la suma de los arrays `vector2` y `vector3`.

Por último, hay que mostrar los cuatro arrays en cuatro columnas.

El código que implemente este ejercicio deber utilizar estructuras de control de iteración para recorrer los vectores y para realizar las operaciones con ellos.

Proyecto 6.2. VECTORALEATORIO

Antes de comenzar este segundo ejercicio es conveniente que lea y comprenda el Anexo I donde se indica el procedimiento para generar números aleatorios en C.

Una vez que ha leído y comprendido el Anexo I, cree un nuevo proyecto llamado **VECTORALEATORIO**. En él hay que implementar un programa que asigne valores de forma aleatoria un vector de dimensión predefinida (por ejemplo 30). El vector estará compuesto por un total de 30 números enteros generados aleatoriamente en el intervalo [a, b]. Para ello, el programa deberá:

1. Definir un vector de tamaño predefinido (30). Indique el tamaño mediante una directiva **`define`**.
2. Solicitar al usuario los valores enteros que definen el intervalo [a, b]. Deberá tener en cuenta dos aspectos:
   - Si a es mayor que b, entonces deberá intercambiar sus valores
   - Si la diferencia b – a es menor que 100, entonces se deberá volver a pedir nuevos valores para a y b.

     y esto se hará hasta se hayan introducido dos valores que si verifiquen que `b - a >= 100`.
3. Con una estructura de iteración el programa deberá recorrer cada uno de los elementos del vector e inicializarlos aleatoriamente con enteros comprendidos entre a y b.
4. Finalmente el programa deberá mostrar el vector por pantalla en formato columna.

Proyecto 6.3. INTERCAMBIO

A partir del código desarrollado en el proyecto anterior, cree un nuevo proyecto denominado **INTERCAMBIO**. El programa debe ordenar el vector de enteros aleatorios de menor a mayor utilizando el *método de ordenación utilizando el algoritmo del intercambio*. Este algoritmo resulta poco eficiente si se desea ordenar una colección grande de valores; pero para colecciones no muy extensas es el más sencillo de implementar y resulta eficiente: El algoritmo recorre, una a una, todas las posiciones del vector, desde la primera hasta la penúltima. A partir de cada posición del array se compara el valor de esa posición con los valores de todas las posiciones siguientes, uno después de otro; y cada vez que se encuentra un valor menor que el de la posición actual se procede al intercambio de valores. Al hacerlo con la primera posición del array, se logra que ésta, al final, tenga almacenado el menor de los valores del array; al hacerlo con la segunda posición del array, se logra que ésta, al final, tenga almacenado el menor de los restantes valores del array que no son el primero; al hacerlo con la tercera posición del array, se logra que ésta, al final, tenga almacenado el menor de los restantes valores del array que no son ni el primero ni el segundo;…

El algoritmo de intercambio tiene dos formas de implementarse equivalentes:

- La primera consiste en realizar sobre todas las posiciones, comenzando por la primera (índice `i = 0`) y de forma ordenada hasta la penúltima (índice `i = n – 2`), la comparación de los valores contenidos en cada una de esas posiciones con todos los valores ubicados en posiciones posteriores, es decir, de índice más alto (desde `j = i + 1` hasta `j = n – 1`), e intercambiar valores cada vez que se cumpla que el valor de la posición señalada con el índice `j` es mayor que el señalado con el índice `i`: Así, va quedando en cada posición del array el menor de todos los valores que quedan "a su derecha" y que siempre será mayor o igual que cualquiera de los valores que han quedado previamente "a su izquierda".
- La segunda consiste en realizar sobre todas las posiciones, comenzando por la última (índice `i = n - 1`) y de forma ordenada hasta la segunda (índice `i = 1`), la comparación de los valores contenidos en cada una de esas posiciones con todos los valores ubicados en posiciones anteriores, es decir, de índice más bajo (desde `j = i – 1` hasta `j = 0`), e intercambiar valores cada vez que se cumpla que el valor de la posición señalada con el índice `j` es menor que el señalado con el índice `i`: Así, va quedando en cada posición del array el mayor de todos los valores que quedan "a su izquierda" y que siempre será menor o igual que cualquiera de los valores que han quedado previamente "a su derecha".

Evidentemente ambos procedimientos son equivalentes. Veamos un ejemplo de ordenación de 10 valores según cada una de las dos formas presentadas.

De acuerdo a la primera forma propuesta, el algoritmo realizaría los siguientes cambios de valores en cada nueva iteración:

POSICIONES	1ª	2ª	3ª	4ª	5ª	6ª	7ª	8ª	9ª	10ª
VALOR DEL ÍNDICE	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Valores iniciales	8	3	1	9	2	3	6	7	0	4
Primero con todos…	0	8	3	9	2	3	6	7	1	4

Segundo con los demás…	0	1	8	9	3	3	6	7	2	4
Tercero con los demás…	0	1	2	9	8	3	6	7	3	4
Cuarto con los demás…	0	1	2	3	8	9	6	7	3	4
Quinto con los demás…	0	1	2	3	3	9	8	7	6	4
Sexto con los demás…	0	1	2	3	3	4	9	8	7	6
Séptimo con los demás…	0	1	2	3	3	4	6	9	8	7
Octavo con noveno y décimo…	0	1	2	3	3	4	6	7	9	8
Noveno con décimo…	0	1	2	3	3	4	6	7	8	9

Por ejemplo, en la primera iteración nos hemos encontrado con que el primer valor (8) era mayor que el segundo (3), y entonces el segundo a pasado a ser el 8 y el primero el 3. Luego hemos comparado el primero (3) con el tercero (1), y de nuevo hemos intercambiado valores, y ha quedado el primero con el valor 1 y el tercero con el valor 3. Luego hemos comparado el primero (1) con el cuarto, quinto, sexto, séptimo y octavo valores, y no se ha producido intercambio de valores alguno porque el primero (1) es ya menor que esos otros valores (9, 2, 3, 6 y 7). Al comparar el primer valor (1) con el noveno (0) sí se produce intercambio porque ahora están desordenados (el más a la izquierda es mayor que el ubicado más a la derecha) y dejamos el valor 1 en la posición 9 y el valor 0 en la primera posición. Finalmente se realiza la comparación del primer valor (0) con la del último (4) y no se produce intercambio. Así queda terminada la primera iteración de forma que ahora el valor de la variable primera es el más pequeño de todos. El proceso se repite con cada posición, donde se realiza la comparación con todos los valores ubicados en posiciones posteriores y el intercambio en el caso de que nos encontremos valores menores.

De acuerdo a la segunda forma propuesta, el algoritmo realizaría los siguientes cambios de valores en cada nueva iteración:

POSICIONES	1ª	2ª	3ª	4ª	5ª	6ª	7ª	8ª	9ª	10ª
VALOR DEL ÍNDICE	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Valores iniciales	8	3	1	9	2	3	6	7	0	4
Décimo con todos…	8	3	1	7	2	3	6	4	0	9
Noveno con los demás…	7	3	1	6	2	3	4	0	8	9
Octavo con los demás…	6	3	1	4	2	3	0	7	8	9
Séptimo con los demás…	4	3	1	3	2	0	6	7	8	9

Sexto con los demás…	3	3	1	2	0	4	6	7	8	9
Quinto con los demás…	3	2	1	0	3	4	6	7	8	9
Cuarto con los demás…	2	1	0	3	3	4	6	7	8	9
Tercero con segundo y primero…	1	0	2	3	3	4	6	7	8	9
Segundo con primero…	0	1	2	3	3	4	6	7	8	9

Por ejemplo, en la primera iteración nos hemos encontrado con que el último valor (4) ya era mayor que el penúltimo (0), y entonces no se ha hecho intercambio. Luego se ha comparado de nuevo ese el último valor (4) con el ubicado en la antepenúltima posición (7) que, al ser mayor, se intercambian: ahora la posición décima tiene un 7 y la octava un 4. Seguimos comparando y no hay intercambio hasta llegar a la posición cuarta donde encontramos un 9: se produce entonces un nuevo intercambio, y la posición cuarta pasa a guardar el 4 y el 9 pasa a la posición décima. El resto de valores (primero, segundo y tercero) son menores que el actual ubicado en la última posición, y ya no se producen nuevos intercambios. En la última posición ha quedado el mayor de los valores de la lista.

Proyecto 6.4. CALCULADORAVECTORES [EJERCICIO ADICIONAL]

Cree un proyecto denominado **CALCULADORAVECTORES**.

El programa debe:

1. Declarar dos vectores (u y v) de tres elementos cada uno (datos de tipo **`double`**) e inicializarlos poniendo a 0 todas las posiciones.
2. Asignar por teclado los valores de los tres componentes de cada vector.
3. Calcular el producto escalar de u y v. Si los vectores son perpendiculares (su producto escalar es nulo), el programa deberá indicarlo por pantalla.

$$u \cdot v = [u_0, u_1, u_2] \cdot [v_0, v_1, v_2] = u_0 \cdot v_0 + u_1 \cdot v_1 + u_2 \cdot v_2$$

4. Calcular la distancia euclídea (raíz cuadrada de la suma de los cuadrados de las diferencias de las componentes) entre los dos vectores.

$$d(u, v) = \sqrt[2]{(u_0 - v_0)^2 + (u_1 - v_1)^2 + (u_2 - v_2)^2}$$

5. Calcular directamente el producto vectorial de ambos vectores y almacenar las nuevas componentes calculadas en un tercer vector w.

$$w = u \times v$$

$$[w_0, w_1, w_2] = [u_0, u_1, u_2] \times [v_0, v_1, v_2] = \left[\begin{vmatrix} u_1 & u_2 \\ v_1 & v_2 \end{vmatrix}, -\begin{vmatrix} u_0 & u_2 \\ v_0 & v_2 \end{vmatrix}, \begin{vmatrix} u_0 & u_1 \\ v_0 & v_1 \end{vmatrix}\right]$$

SOLUCIONES A LOS EJERCICIOS DE LOS PROYECTOS PROPUESTOS

Código 6.3. Solución a P6.1 Ejercicio 1.

```
#include <stdio.h>

#define _D 5

int main(void)
{
      int vector1[_D];
      int vector2[_D];
      int vector3[_D];
      int vector4[_D];
      int i; //contador

      for(i = 0 ; i < _D ; i++)
      {     // vector1: asignacion por teclado
            printf("Introduzca el elemento %d para vector1...", i + 1);
            scanf(" %d",&vector1[i]);
      }

      for(i = 0 ; i < _D ; i++)
      {     // vector2:copia de vector1
            vector2[i] = vector1[i];
      }

      for(i = 0 ; i < _D ; i++)
      {     // vector3: copia inversa de vector1
            vector3[i] = vector2[(_D - 1) - i];
      }

      for(i = 0 ; i < _D ; i++)
      {     // vector4: suma de vector2 y vector3
            vector4[i]=vector2[i] + vector3[i];
      }

      printf("\nIMPRESION POR PANTALLA DE LOS CUATRO VECTORES\n");
      for(i = 0 ; i < _D ; i++)
      {
            printf("%10d - %10d - %10d - %10d\n",
            vector1[i] , vector2[i] , vector3[i] , vector4[i]);
      }

      return 0;
}
```

Código 6.4. Solución a P6.2 Ejercicio 1.

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

#define _D 30

int main(void) {
      int vector[_D];
      int a, b;         // intervalo
      int aux;          // variable auxiliar
      int i;            // contador

      i = 0;
      do
      {
            if(i > 0)
            {
                  printf("\nLa diferencia b - a debe ser mayor que 100");
                  printf("\nVuelva a introducir valores para a y b\n\n");
            }
            printf("Introduzca el valor de a ...");  scanf(" %d", &a);
            printf("Introduzca el valor de b ...");  scanf(" %d", &b);

            if(a > b)
            { // intercambio los valores de a y b
                  aux = b;
                  b = a;
                  a = aux;
                  printf("\na es mayor que b: intercambio sus valores\n");
            }
            i++;
      } while(b – a < 100);

// Asignación de valores aleatorios entre a y b al array
      srand(time(NULL)); // establecimiento de semilla

      for(i = 0 ; i < _D ; i++)
      {
            vector[i] = rand() % (b – a + 1) + a;
      }

// Mostrar los valores por pantalla
      for(i = 0 ; i < _D ; i++)
      {
            printf("%8d\n", vector[i]);
      }

      return 0;
}
```

Código 6.5. Solución a P6.3 Ejercicio 1. Primera implementación.

```
     #include <stdio.h>
     #include <stdlib.h>
     #include <time.h>
     #define _D 5

     int main(void) {
           int vector[_D];
           int a, b;         // intervalo
           int aux;          // variable auxiliar
           int i , j;        // contadores

/*01*/     i = 0;
           do
           {
                 if(i > 0)
                 {
                       printf("\nLa diferencia b - a debe ser mayor que 100");
                       printf("\nVuelva a introducir valores para a y b\n\n");
                 }
                 printf("Introduzca el valor de a ...");  scanf(" %d", &a);
                 printf("Introduzca el valor de b ...");  scanf(" %d", &b);

                 if(a > b)
                 { // intercambio los valores de a y b
                       aux = b;
                       b = a;
                       a = aux;
                       printf("\na es mayor que b: intercambio sus valores\n");
                 }
                 i++;
           } while(b – a < 100);

     // Asignación de valores aleatorios entre a y b al array
           srand(time(NULL)); // establecimiento de semilla

           for(i = 0 ; i < _D ; i++)
           {
                 vector[i] = rand() % (b – a + 1) + a;
           }
/*02*/

     //ORDENACION POR INTERCAMBIO (PRIMERA IMPLEMENTACION)

           for(i = 0 ; i < _D ; i++)
           {
                 for(j = i + 1 ; j < _D ; j++)
                 {
                       if(vector[i] > vector[j])
                       {
```

Código 6.5. (Cont.)

```
                              aux = vector[i];
                              vector[i] = vector[j];
                              vector[j] = aux;
                        }
                  }
            }
      //IMPRIMIR POR COLUMNAS
            printf("\n VECTOR ORDENADO \n");
            for(i = 0 ; i < _D ; i++)
            {
                  printf("%d\n", vector[i]);
            }

            return 0;
      }
```

Código 6.6. Solución a P6.3 Ejercicio 1. Segunda implementación.

```
// Todo el código marcado en Código 6.5., entre las líneas /*01*/ y /*02*/
// Deben insertarse, exactamente igual, en esta parte de este programa,
// en los mismos sitios donde quedan indicadas las líneas /*01*/ y /*02*/

      #include <stdio.h>
      #include <stdlib.h>
      #include <time.h>
      #define _D 5

      int main(void) {
            int vector[_D];
            int a, b;         // intervalo
            int aux;          // variable auxiliar
            int i , j;        // contadores
/*01*/
// INSERTAR AQUÍ EL CÓDIGO IGUAL QUE EN Código 6.5.
/*02*/
      //ORDENACION POR INTERCAMBIO (SEGUNDA IMPLEMENTACION)

      for(i = _D - 1 ; i > 0; i--){
            for(j = 0 ; j < i ; j++){
                  if(vector[j] > vector[i]){
                        aux = vector[i];
                        vector[i] = vector[j];
                        vector[j] = aux;
                  }
            }
      }
```

Código 6.6. (Cont.)

```
	//IMPRIMIR POR COLUMNAS
	printf("\n VECTOR ORDENADO \n");
	for(i = 0 ; i < _D ; i++){
		printf("%d\n", vector[i]);
	}

	return 0;
}
```

Código 6.7. Solución a P6.4 Ejercicio 1.

```
	#include <stdio.h>
	#include <math.h>
	#define _D 3

	int main(void)
	{
		double u[_D], v[_D], w[_D];
		int i, j;
		double p_escalar, distancia;

		for(i = 0 ; i <_D ; i++)
		{
			u[i] = v[i] = 0.0;
		}

		printf("Leer elementos de u...\n");
		for(i = 0 ; i < _D ; i++)
		{
			printf("\t Introduzca el elemento %d para u...", i+1);
			scanf(" %lf", &u[i]);
		}
		printf("Leer elementos de v...\n");
		for(i = 0 ; i < _D ; i++)
		{
			printf("\t Introduzca el elemento %d para v...", i+1);
			scanf(" %lf", &v[i]);
		}

	//PRODUCTO ESCALAR
		p_escalar=0.0;
		for(i = 0 ; i < _D ; i++)
		{
			p_escalar += u[i] * v[i];
		}
```

Código 6.7. (Cont.)

```
        printf("El producto escalar es %.2lf", p_escalar);

        if(p_escalar == 0.0)
        {
              printf("Los vectores u y v son perpendiculares");
        }

  //DISTANCIA EUCLIDEA
        distancia = 0.0;
        for(i = 0; i < _D; i++)
        {
              distancia += pow(u[i] - v[i], 2);
        }
        distancia = sqrt(distancia);
        printf("\nLa distancia euclidea entre u y v es %.2lf", distancia);

  //PRODUCTO VECTORIAL
        w[0] = u[1] * v[2] - u[2] * v[1];
        w[1] = -(u[0] * v[2] - u[2] * v[0]);
        w[2] = u[0] * v[1] - u[1] * v[0];

        printf("\nEl producto vectorial de u y v es el vector...\n");
        for(i = 0 ; i < _D ; i++)
        {
              printf("%lf\n", w[i]);
        }

        return 0;
  }
```

ANEXO I. GENERACIÓN DE NÚMEROS ALEATORIOS EN C

Para obtener números aleatorios, tenemos la función `rand()` que se encuentra en la librería `<stdlib.h>`. Esta función devuelve, según una distribución de probabilidad uniforme, un número entero aleatorio entre **0** y el **RAND_MAX**, que es un valor que depende de la arquitectura del ordenador y de la configuración del compilador. Para una arquitectura de 32 bits, su valor es 2147483647, que en hexadecimal es 7FFFFFFF. Es un valor que está definido en el archivo de cabecera `stdlib.h`.

El programa que se propone en Código 6.8. genera un número aleatorio y lo muestra por pantalla:

Código 6.8. Programa que genera un valor aleatorio y lo muestra por pantalla.

```
#include <stdio.h>
#include <stdlib.h> // libreria para función rand()

int main(void)
{
      int n; //numero aleatorio
      n = rand();//generar entero aleatoriamente entre 0 y RAND_MAX
      printf("El entero aleatorio es: %d", n);
      return 0;
}
```

La mayor parte de los generadores de números aleatorios son, en realidad, ***pseudoaleatorios***: a partir de un valor inicial, que se suele denominar *semilla*, se genera un nuevo valor, y luego otro, y otro, y otro,… de forma completamente determinista. Así, siempre que se parta de la misma semilla, se obtendrá la misma secuencia de valores. Si ejecuta el programa propuesto en Código 6.8. varias veces, verá que… ¡siempre obtiene el mismo valor aleatorio!: y es que si no le damos un valor aleatorio (semilla) inicial al generador, éste siempre genera la misma secuencia de valores. Por esta razón, debemos modificar la semilla al menos una vez antes de hacer el primer uso del generador. ¿No es extraño?: es necesario darle un valor aleatorio al generador si queremos que los valores que éste nos vaya generando sean, al menos aparentemente, aleatorios e impredecibles.

Una manera de generar una semilla, distinta de una vez a otra que se ejecute el programa, es a través de la función `time(NULL)` (incluida en el archivo de cabecera `time.h`) que nos devuelve el número de segundos transcurridos desde el 1 de Enero de 1970 hasta el instante en que se ejecuta la función `time()`. Para cambiar la semilla del generador de números aleatorios se dispone de la función `srand()`. Antes de ejecutar la función que genera nuevos aleatorios, deberemos añadir en nuestro programa la siguiente sentencia: `srand(time(NULL))`. Es suficiente con llamar a esta función una vez al principio de nuestro programa. Puede verlo en Código 6.9., en la línea marcado con `/*01*/`.

Existe el modo de resembrar el generador de pseudoaleatorios en diferentes ocasiones. Efectivamente, no tendría sentido hacerlo todas ellas con el mismo valor `time(NULL)`, que posiblemente sería siempre el mismo, o variaría de valor una sola vez cada segundo. Pero se puede reasignar un nuevo valor a la semilla con valores pseudoaleatorios obtenidos del propio generador.

En Código 6.10 se asigna un valor aleatorio a cada una de las variables de un array de 1000 elementos, y se renueva la semilla en cada iteración.

Código 6.9. Inicialización de la semilla del generador de aleatorios.

```
     #include <stdio.h>
     #include <stdlib.h> // libreria para función rand()
     #include <time.h>

     int main(void)
     {
           int n; //numero aleatorio
/*01*/     srand(time(NULL)); //reiniciar semilla
           n = rand();//generar entero aleatoriamente entre 0 y RAND_MAX
           printf("El entero aleatorio es: %d", n);
           return 0;
     }
```

Código 6.10. Renovación de la semilla después de cada nuevo valor aleatorio generado.

```
     #include <stdio.h>
     #include <stdlib.h>    // libreria para función rand()
     #include <time.h>      // libreria para la función time()

     #define _DIM 1000

     int main(void)
     {
           int array[_DIM]; // array donde iran los aleatorios
           int i;

/*01*/     for(srand(time(NULL)) , i = 0 ;
               i < _DIM ;
/*02*/         srand(rand()) , i++)
           {
                 array[i] = rand();//entero aleatorio entre 0 y RAND_MAX
           }

           return 0;
     }
```

En la línea `/*01*/` de Código 6.10 se inicializa el generador de aleatorios con una semilla que coincide con el valor que en el momento de la ejecución del programa nos dé la llamada a la función `time()`. Luego, en la línea `/*02*/` se renueva, después de cada iteración, el valor de la semilla del generador.

Si deseamos acotar el rango de los valores aleatoriamente generados (por ejemplo, que estén en el intervalo [0, b]) deberemos hacer uso de la operación resto. En Código 6.11. se puede ver cómo asignar valores en un intervalo determinado.

Código 6.11. Valores aleatorios en un rango determinado [0 , b].

```
#include <stdio.h>
#include <stdlib.h>     // libreria para función rand() y srand()
#include <time.h>       // libreria para la función time()

#define _B 100          // Límite superior de los valores aleatorios

int main(void)
{
      int n; //numero aleatorio

      srand(time(NULL)); //reiniciar semilla
      n = rand() % (_B + 1); // entero entre 0 y _B

      printf("El entero aleatorio es: %d", n);

      return 0;
}
```

La operación módulo (%) nos da el resto de dividir rand() entre (_B + 1) (en el ejemplo, 101). Este resto puede ir de 0 a _B (100 en el ejemplo de Código 6.11.).

¿Y si queremos generar aleatoriamente números enteros dentro de un intervalo [a, b]? Es sencillo y directo:

```
n = rand() % (b – a + 1) + a; //generar aleatoriamente un entero entre a y b
```

Es sencillo verlo. Si en un primer momento le desconcierta esta expresión, le sugerimos simplemente que haga la prueba con algunos valores. Si a vale 10 y b vale 50, tenemos que (b – a + 1) vale 41, y la operación rand() % (b – a + 1) será un valor entre 0 y 40. Si a ese valor le sumamos el valor de a (hemos quedado que era 10) tendremos un valor final entre 10 y 50. Otro ejemplo: si a vale -50 y b vale +50, tenemos que (b – a + 1) vale 101, y la operación rand() % (b – a + 1) será un valor entre 0 y 100. Si a ese valor le sumamos el valor de a (hemos quedado que era -50) tendremos un valor final entre -50 y +50.

Generación de números reales

Una forma muy sencilla de generar aleatoriamente un número real es la siguiente:

```
double n;
n = (double) rand()/RAND_MAX; //genera aleatoriamente un entero real entre 0 y 1
```

Como se puede ver en este código, el número entero generado por `rand()`, que como sabe está comprendido entre 0 y RAND_MAX, se fuerza al tipo **double** y se divide por RAND_MAX. Así se obtiene un número real comprendido entre 0.0 y 1.0 (ambos incluidos). A partir de ahí es fácil generar valores reales en el rango que queramos. Si queremos un rango entre 20.0 y 30.0 (o entre a y b, siendo b mayor que a y ambos con decimales) ejecutaremos las siguientes sentencias:

```
//generar aleatoriamente un entero real entre 20 y 30
n = ((double) rand()/RAND_MAX)* (30.0-20.0) + 20.0;

//generar aleatoriamente un entero real entre a y b
n = ((double) rand()/RAND_MAX) * (b - a) + a;
```

Esta forma de generar reales aleatorios es verdaderamente sencilla, y logra generar una colección de valores muy limitada. Cuando se necesita en una aplicación trabajar con valores reales verdaderamente aleatorios, se utilizan generadores más complejos y completos, que no son objeto de estudio en este manual de introducción.

Arrays numéricos (II)

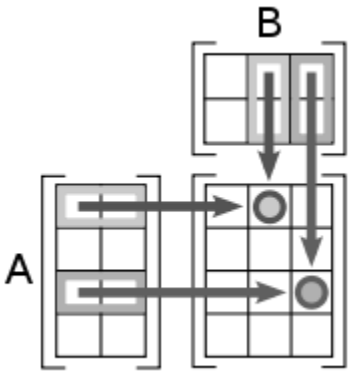
7

En la práctica anterior se ha trabajado con arrays unidimensionales o vectores. Ahora se va a seguir trabajando con **matrices** o arrays multidimensionales.

El objetivo de este capítulo es que el alumno aprenda a declarar, inicializar y recorrer los elementos dentro de las matrices. También se va a profundizar en el uso de los vectores.

Inicialmente se presenta un ejemplo sencillo que es una extensión directa del ejemplo introductorio del capítulo anterior. En este caso, en lugar de procesar las ventas correspondientes a una serie de agentes comerciales en un mes concreto, haremos uso de matrices para poder procesar los datos correspondientes a todos los meses de un año de cada uno de los agentes.

A continuación, el alumno deberá ser capaz de implementar una serie de programas para realizar distintas operaciones con matrices, como son inicializar una matriz, calcular la matriz diagonal superior, obtener la traspuesta de una matriz, encontrar el valor máximo en una matriz, obtener los elementos de la diagonal principal, realizar el producto de dos matrices…



Al igual que en la anterior sesión, para la resolución de los proyectos propuestos será necesario revisar el Anexo I del Capítulo 6 de este libro, donde se recogen unos conceptos básicos sobre la generación de números aleatorios.

7.1. EJEMPLOS SENCILLOS RESUELTOS

Antes de comenzar con los problemas que componen este capítulo, será conveniente mostrar un ejemplo inicial que permita al alumno introducirse en el manejo de las matrices.

Ejemplo 7.1. VENTASANUALES

Creamos ahora un nuevo proyecto denominado **VENTASANUALES**. Queremos ahora almacenar en una matriz las ventas que cada uno de los diez agentes comerciales ha realizado en cada uno de los doce meses del año. En este caso, un agente comercial obtendrá un incremento de sueldo si sus ventas anuales (la suma de los doce meses) son superiores a la media de ventas de todos los agentes en todo el año.

Esta situación requiere almacenar las ventas de los diez agentes para cada uno de los doce meses del año en alguna estructura de datos: lo más común es disponer de una tabla con diez filas (una por cada agente comercial) y doce columnas (una por cada mes del año), o al contrario: doce filas y 10 columnas. En programación, esto se consigue mediante el uso de **arrays multidimensionales** o **matrices**.

La declaración de matrices con dos dimensiones es sencilla. La siguiente sentencia declara una matriz que hemos llamado `ventas` y que tiene 10 filas y 12 columnas, donde se almacenarán 120 datos de tipo **`double`**:

```
double ventas[10][12];// se acaban de declarar 120 variables tipo double.
```

Desde el Standard C99 las matrices también se pueden declarar usando variables enteras para indicar sus dimensiones. Así, el código arriba escrito, también podría quedar de la siguiente forma:

```
short N = 10, M = 12;
double ventas[N][M];
```

En esta matriz podemos asociar los índices `i` y `j` a las filas y las columnas, respectivamente. De esta forma, la fila i-ésima contiene toda la información de ventas en cada mes del año para el agente comercial i-ésimo. Asimismo, la columna j-ésima contiene toda la información de ventas de todos los agentes comerciales en el mes j-ésimo del año. Al igual que en los vectores, el primer elemento de una fila o columna comienza por 0. Por ejemplo, `ventas[0][11]` es el volumen de ventas del primer agente en el mes de Diciembre (último mes del año); mientras que `ventas[9][0]` representa las ventas del décimo agente comercial en el primer mes del año (Enero).

Si deseásemos, por ejemplo, inicializar los elementos de estas matriz bidimensional a 0 podríamos hacerlo de dos formas principales:

- A través de un doble bucle:

```
for(i = 0; i < N ; i++){
     for(j = 0; j < N ; j++){
          ventas[i][j] = 0;
     }
}
```

- A través del operador asignación y llaves. Puede asignar valores a la matriz, al declararla, indicándole entre llaves los valores de cada fila, y recogiendo, también entre llaves, los valores de las distintas filas. Habitualmente será lógico asignar tantos valores como se hayan declarado. Pero no es necesario que se haga así: puede asignar menos valores de los declarados, y el compilador asignará directamente el valor 0 al resto de los elementos de la matriz.

  Suponga, por ejemplo, el programa recogido en Código 7.1. Como ve, sólo se le asignan 4 filas de valores, cuando se han declarado 5 filas en la matriz. Y además no siempre se asignan, en cada fila, tantos valores como columnas tiene la matriz: a la primera fila le faltan 2 valores, a la segunda le falta 1 y a la cuarta fila… ¡le sobra un valor! La salida que, por pantalla, dará este programa se puede ver en la Figura 7.1. Que en una fila falten valores no es problema, y el compilador simplemente completa la asignación del resto de valores de la fila, o la asignación del resto de filas no asignadas, con el valor 0. Pero el exceso de valores es un error práctico a evitar y que, de hecho, el compilador advierte.

Código 7.1. Ejemplo de inicialización de valores en la declaración de una matriz.

```
#include <stdio.h>

int main(void)
{
      int matriz[5][5] = {{0, 0, 0},{1, 1, 1, 1},
                          {2, 2, 2, 2, 2},{3, 3, 3, 3, 3, 3},};
      short i, j;

      for(i = 0 ; i < 5 ; i++)
      {
            for(j = 0 ; j < 5 ; j++)
                  printf("%5d", matriz[i][j]);
            printf("\n");
      }

      return 0;
}
```

Una buena inicialización de nuestra matriz de ventas podría ser la siguiente:

```
double ventas [10][12] =    {{0,0,0,0,0,0,0,0,0,0,0,0},
{0,0,0,0,0,0,0,0,0,0,0,0} , {0,0,0,0,0,0,0,0,0,0,0,0},
{0,0,0,0,0,0,0,0,0,0,0,0} , {0,0,0,0,0,0,0,0,0,0,0,0},
{0,0,0,0,0,0,0,0,0,0,0,0} , {0,0,0,0,0,0,0,0,0,0,0,0},
{0,0,0,0,0,0,0,0,0,0,0,0} , {0,0,0,0,0,0,0,0,0,0,0,0},
{0,0,0,0,0,0,0,0,0,0,0,0}};
```

```
0  0  0  0  0
1  1  1  1  0
2  2  2  2  2
3  3  3  3  3
0  0  0  0  0
```

Figura 7.1. Salida que, por pantalla, ofrecería el programa de Código 7.1.

NOTA: Recordará que también con los arrays unidimensionales se podía hacer este tipo de declaración, donde se asignaban valores iniciales, mediante llaves, al array declarado. Cuando trabajábamos con los vectores no era necesario, en ese caso, indicar la dimensión del array, que quedaba implícita con la misma asignación.

Quizá, pudiera pensarse, ahora ocurra la mismo, y se puedan omitir las dimensiones de la matriz cuando en su declaración se le asignan valores iniciales. Pero de hecho, si hace eso, el compilador le señalará un error: algo así como que no se ha declarado el tipo del array de forma completa. No es éste el momento de explicarle por qué pasa eso. Simplemente, por ahora, tenga en cuenta que cuando declara matrices y asigna valores iniciales en esa declaración, deberá, obligatoriamente, indicar las dimensiones de la matriz: al menos deberá indicar la segunda de las dos dimensiones:

```
double ventas [][12] = {{...}} ;
```

En Código 7.2. se muestra el código completo del programa propuesto sobre las ventas de los agentes comerciales, para la solución basada en matrices.

Código 7.2. Programa propuesto en Ejemplo 7.1.

```
#include <stdio.h>
#define _N 10 //numero de agentes comerciales
#define _M 12 //numero de meses del año
```

Código 7.2. (Cont.)

```
     int main(void)
     {
          int i;                  // indice desde 0 hasta _N
          int j;                  // indice desde 0 hasta _M
          double ventas[_N][_M];  // matriz ventas
          double suma[_N];        // ventas anuales para cada agente
          double total = 0.0;     // total de ventas de todos los agentes
          double media = 0.0;     // valor promedio de ventas

     //inicializacion del vector suma
          for(i = 0 ; i < _N ; i++) {
               suma[i] = 0.0;
          }

     //introduccion de ventas de los N agentes comerciales en los M meses
          for(i = 0 ; i < _N ; i++) {
               for(j = 0 ; j < _M ; j++) {
               //introduccion de ventas del agente i-esimo en el mes j-esimo
                    printf("Ventas del agente %d en mes %d...", i, j);
                    scanf(" %lf", &ventas[i][j]);
               }
          }

     //calculo la media de las ventas
          for(i = 0 ; i < _N ; i++) {
               for(j = 0 ; j < _M ; j++) {
               //incremento suma con las ventas del agente i en el mes j
/*01*/               suma[i] += ventas[i][j];
               }
/*02*/          media += suma[i];
          }
/*03*/     media = media / _N;

     //determinar si el agente i tiene ventas anuales superiores a la media
          for(i = 0 ; i < _N ; i++) {
               if(suma[i] > media) {
                    printf("El agente %d tiene incremento de sueldo \n", i);
               }
          }
          return 0; //el programa finaliza correctamente
     }
```

En esta solución propuesta, se ha empleado un vector que almacena las ventas anuales de cada agente. Los elementos de este vector `suma` son previamente inicializados a 0 y posteriormente actualizados dentro del doble bucle a través de la sentencia de la línea `/*01*/`.

Necesitamos además calcular el número medio de ventas de todos los agentes en todo el año. Para ello, vamos sumando en la variable `media` (línea `/*02*/`) el número total de ventas anuales para

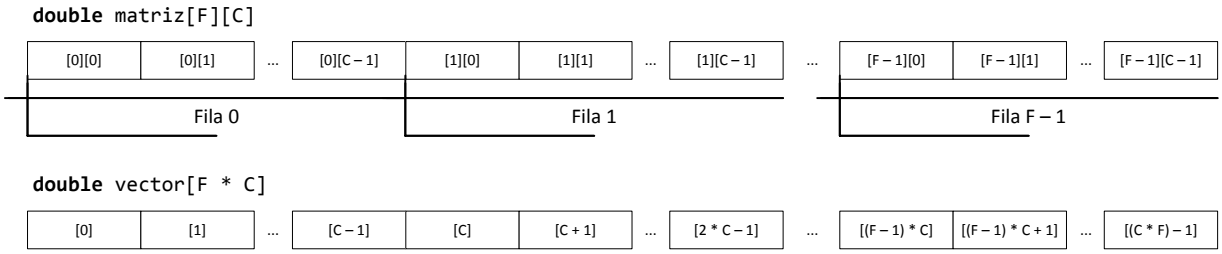
cada agente dentro del primer bucle y, posteriormente (al salir del bucle que recorre la matriz: línea `/*03*/`), se calcula la media dividiendo esa suma calculada por el número total de comerciales: N.

Una observación final sobre cómo interpretar los vectores y las matrices. ¿Qué diferencia existe entre estas dos declaraciones de arrays?:

```
double matriz[_F][_C];
double vector[_F * _C];
```

En el primer caso se crean `_F * _C` variables ubicadas en memoria de forma consecutiva y a las que podemos referenciar con el nombre del array (`matriz`) y dos índices: uno para las filas y otro para las columnas. El modo de interpretar esta estructura de datos es considerar la matriz como un array de arrays de variables tipo **`double`**: un array de `_F` vectores, cada uno de ellos con `_C` elementos tipo **`double`**. Y así, el elemento `matriz[f][c]` (con `f` menor que `_F`, y `c` menor que `_C`) está ubicado en el elemento `c` del vector `f`. A cada vector de la matriz lo consideramos una fila, y todas las filas tienen el mismo número de elementos, que son lo que llamamos columnas. (Nada impediría, sin embargo, interpretar los vectores como columnas y, dentro de cada vector, los elementos distribuidos por filas: pero hay que elegir una interpretación, y la propuesta es, quizá, más afín al esquema mental que nos hacemos del algebra de una matriz.)

Con la segunda declaración se crean también `_F * _C` variables ubicadas en memoria de forma consecutiva y a las que podemos referenciar con el nombre del array (`vector`) y un solo índice. No se requiere en este caso más explicación.



Así las cosas, puede parecer que en ambas declaraciones se ha creado la misma estructura de datos. Pero en realidad no es así: no podría acceder a los elementos del array `matriz` si no tuviera dos índices, ni sabríamos qué hacer en realidad con un segundo índice en el array `vector`. Y es que en ambas declaraciones, además de las variables tipo **`double`**, se ha creado un sistema de acceso a esas variables: lo podrá conocer y comprender más adelante, cuando hablemos de los punteros. Esa red de punteros son los que permiten acceder a los distintos espacios de memoria con uno o con dos índices.

De todas formas, como la distribución de variables es la misma en ambas declaraciones, es fácil hacer uso de un array como si se tratase de una matriz (es decir, con dos índices), y es fácil hacer uso de una matriz como si fuera un array (es decir, con un solo índice). Vea por ejemplo el código propuesto en Código 7.3. Como en ese ejemplo puede ver, podríamos recorrer el array bidimensional `matriz` con un solo índice, donde la fila sería en cada caso el resultado del cociente `i / _C` y donde la columna sería la indicada con el valor del resto `i % _C`. De la misma forma, podríamos recorrer un array con dos índices. Vea sino el ejemplo sugerido en Código 7.4. Así que podríamos recorrer el array monodimensional `vector` con dos índices, donde la posición de cada elemento quedaría indicada con la expresión `f * _C + c`.

Código 7.3. Recorrer una matriz con dos índices y con un solo índice.

```
      #define _F 4
      #define _C 5

      int main(void)
      {
            double matriz[_F][_C];
            short f, c;       // Para recorrer el array matriz con dos índices.
            short i;          // Para recorrer el array matriz con un índice.
            double k;

// Asignamos a la matriz valores consecutivos desde 1 hasta _F * _C:
            for(f = 0 , k = 1.0 ; f < _F ; f++)
            {
                  for(c = 0 ; c < _C ; c++ , k++)
                  {
                        matriz[f][c] = k;
                  }
            }

// Hacemos la misma operación, utilizando sólo un índice i:
            for(i = 0 , k = 1.0 ; i < _F * _C ; i++ , k++)
            {
                  matriz[i / _C][i % _C] = k;
            }

            return 0;
      }
```

Código 7.4. Recorrer un array monodimensional con dos índices y con un solo índice.

```
      #define _F 4
      #define _C 5

      int main(void)
      {
            double vector[_F * _C];
            short f, c;       // Para recorrer el array vector con dos índices.
            short i;          // Para recorrer el array vector con un índice.
            double k;

// Asignamos al vector valores consecutivos desde 1 hasta _F * _C:
            for(i = 0 , k = 1.0 ; i < _F * _C ; i++ , k++)
            {
                  vector[i] = k;
            }
```

Código 7.4. (Cont.)

```
// Hacemos la misma operación, utilizando ahora dos índices f y c:
            for(f = 0 , k = 1.0 ; f < _F ; f++)
            {
                  for(c = 0 ; c < _C ; c++ , k++)
                  {
                        vector[f * _C + c] = k;
                  }
            }

            return 0;
      }
```

7.2. PROYECTOS PROPUESTOS

Proyecto 7.1. MATRIZ

Cree un proyecto denominado **MATRIZ**. El programa debe:

1. Declarar una matriz X con dimensiones N x M (no necesariamente cuadrada, es decir, N no tiene que ser igual a M). Para fijar los valores de las dimensiones, utilice dos directivas **`define`**.
2. Asignar valores a cada uno de sus elementos: *a cada posición asignar un entero consecutivo respecto a la posición anterior y comenzando por el 1*.
3. Finalmente, debe mostrar por pantalla la matriz creada.

Proyecto 7.2. MATRIZTRIANGULAR

Cree un proyecto denominado **MATRIZTRIANGULAR**. El programa debe

1. Declarar una matriz X cuadrada de dimensión N. Para fijar el valor de N, utilice una directiva **`define`**.
2. Asignar valores a cada uno de sus elementos de tal forma que se obtenga una matriz triangular superior. Debe recorrer toda la matriz y asignar esos valores mediante estructuras de iteración.
3. Finalmente, debe mostrar por pantalla la matriz creada.

NOTA: La matriz triangular superior es aquella cuyos elementos de la diagonal principal y aquellos que están ubicados por encima de ella son igual a 1; y los elementos de la matriz por debajo de esa diagonal principal son igual a 0. Ejemplo de la matriz triangular superior 5 X 5:

1 1 1 1 1
0 1 1 1 1
0 0 1 1 1
0 0 0 1 1
0 0 0 0 1

Proyecto 7.3. MATRIZTRASPUESTA

Cree un proyecto denominado **MATRIZTRASPUESTA**. El programa debe:

1. Declarar una matriz X con dimensiones N x M (no necesariamente cuadrada, es decir, N no tiene que ser igual a M). Para fijar los valores de las dimensiones, utilice dos directivas **`define`**.
2. Declarar otra matriz Y con dimensiones M x N (no necesariamente cuadrada, es decir, N no tiene que ser igual a M).

3. Asignar valores a cada uno de los elementos de X: *a cada posición asignar el resultado del producto de los índices correspondientes a la fila y a la columna a la que estén asociados cada posición*. Así, por ejemplo, si `N = 5` y `M = 4`, la matriz X deberá ser igual a

0	0	0	0
0	1	2	3
0	2	4	6
0	3	6	9
0	4	8	12

4. Asignar valores a cada uno de los elementos de Y para que ésta sea **matriz traspuesta** de X.
5. Finalmente, debe mostrar por pantalla las dos matrices creadas.

Proyecto 7.4. MAXIMOMATRIZ

Cree un proyecto denominado **MAXIMOMATRIZ**. El programa debe:

1. Declarar una matriz X con dimensiones `N x M`. En este caso, tome `N = 20` y `M = 10`.
2. Asignar valores a cada uno de los elementos de X: *a cada elemento asignar un entero aleatorio comprendido entre -10 y +10*.
3. Obtener el **máximo de todos los elementos de la matriz** y determinar el número de veces que ocurre dicho valor.
4. Finalmente, además de la matriz X, debe mostrar por pantalla el valor máximo encontrado y el número de veces que ocurre.

Ejercicio 7.5. MATRIZDIAGONAL

Cree un proyecto denominado **MATRIZDIAGONAL**. El programa debe:

1. Declarar dos matrices cuadradas X e Y de dimensión `N = 10`.
2. Asignar valores a cada uno de los elementos de X: *a cada elemento asignar un entero aleatorio comprendido entre -10 y +10*.
3. Asignar cero a cada uno de los elementos de Y.
4. Una vez las matrices han sido inicializada, **obtener los elementos de la diagonal principal** de X y almacenarlos en la diagonal de la matriz Y.
5. Finalmente, debe mostrar por pantalla ambas matrices.

Ejercicio 7.6. MEDIAMATRIZ

Cree un proyecto denominado **MEDIAMATRIZ**. El programa debe:

1. Declarar una matriz X con dimensiones `_N` y `_M`. Para definir las dimensiones utilizar la directiva define y considerar `_N = 100` y `_M =10`.

2. Declarar un vector z de tamaño _M.
3. Asignar valores a cada uno de los elementos de X: a cada elemento de la columna j-ésima (con j = 1, 2, …, *_M*) asignar un entero aleatorio comprendido entre [-10 *j, +10*j].
4. Una vez la matriz X ha sido inicializada, obtener la media de todos los elementos de cada columna de X y almacenarlos estos promedios en el vector z. Así, el primer elemento de la variable z contiene la media de los valores almacenados en la primera columna de X.
5. Finalmente, debe mostrar por pantalla los elementos del vector z.

Ejercicio 7.7. PRODUCTOMATRICES [EJERCICIO ADICIONAL]

Cree un proyecto denominado **PRODUCTOMATRICES**.

El programa debe:

1. Declarar dos matrices X e Y con dimensiones _N x _M y _M x _R, respectivamente.
2. Declarar una tercera matriz Z con dimensiones _N x _R.
3. Asignar valores a cada uno de los elementos de X e Y: *a cada elemento asignar un entero aleatorio comprendido entre -5 y +5*.
4. La matriz Z debe almacenar el resultado del producto de las matrices X e Y.
5. Finalmente, debe mostrar por pantalla las tres matrices.

Probar el correcto funcionamiento del programa considerando, por ejemplo, las siguientes dimensiones para las matrices:

- _N = 2, _M = 1 y _R = 2
- _N = 2, _M = 2 y _R = 2
- _N = 3, _M = 2 y _R = 1

OBSERVACIÓN: Para indicar las dimensiones de las matrices, quizá pueda serle de utilidad la siguiente declaración de directivas:

```
#define _fil_X          5
#define _col_X          7
#define _fil_Y          _col_X
#define _col_Y          8
#define _fil_Z          _fil_X
#define _col_Z          _col_Y
```

Y así, para redimensionar las matrices, deberá cambiar únicamente los tres enteros que aparecen en esta lista de 6 macros.

SOLUCIONES A LOS EJERCICIOS DE LOS PROYECTOS PROPUESTOS

Código 7.5. Solución para el Ejercicio P7.1.

```
#include <stdio.h>

#define _N 3
#define _M 5

int main(void) {
	int matriz[_N][_M];
	int i, j, contador;

	for(contador = 1 , i = 0 ; i < _N ; i++) {
		for(j = 0 ; j < _M ; j++) {
			matriz[i][j] = contador++;
		}
	}

	for(i = 0 ; i <_ N ; i++) {
		for(j = 0 ; j < _M ; j++) {
			printf("%d\t", matriz[i][j]);
		}
		printf("\n");
	}

	return 0;
}
```

Código 7.6. Solución para el Ejercicio P7.2.

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

int main(void) {
	int matriz[_D][ _D];
	int i, j;

	for(i = 0 ; i < _D ; i++)
	{
		for(j = 0 ; j < _D ; j++)
		{
			if(i <= j)
			{
				matriz[i][j] = 1;
			}
```

Código 7.6. (Cont.)

```
                else
                {
                        matriz[i][j] = 0;
                }
        }
    }

    for(i = 0 ; i < _D; i++)
    {
        for(j = 0 ; j < _D ; j++)
        {
            printf("%6d", matriz[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Código 7.7. Solución para el Ejercicio P7.2. (Otra solución, con operador "? :".

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

int main(void)
{
    int matriz[_D][ _D];
    int i, j;

    for(i = 0 ; i < _D ; i++)
    {
        for(j = 0 ; j < _D ; j++)
        {
            matriz[i][j] = i <= j ? 1 : 0;
        }
    }

    for(i = 0 ; i < _D; i++)
    {
        for(j = 0 ; j < _D ; j++)
        {
            printf("%6d", matriz[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

Código 7.8. Solución para el Ejercicio P7.3.

```
#include <stdio.h>
#define _N 5
#define _M 4

int main(void) {
      int X[_N][_M],Y[_M][_N];
      int i, j;

      for(i = 0 ; i < _N ; i++) {
            for(j = 0 ; j < _M ; j++) {
                  X[i][j] = i * j;        // valor a la matriz X.
                  Y[j][i] = X[i][j];      // trasposición de la matriz X.
            }
      }

      printf("MATRIZ X\n");
      for(i = 0 ; i < _N ; i++) {
            for(j = 0 ; j < _M ; j++) {
                  printf("%6d", X[i][j]);
            }
            printf("\n");
      }

      printf("\nMATRIZ Y (Traspuesta de X)\n");
      for(i = 0 ; i < _M ; i++) {
            for(j = 0 ; j < _N ; j++) {
                  printf("%6d", Y[i][j]);
            }
            printf("\n");
      }
      return 0;
}
```

Código 7.9. Solución para el Ejercicio P7.4.

```
#include <stdio.h>
#include <stlib.h>
#include <time.h>
#define _N 20
#define _M 10

int main(void) {
      int X[_N][_M];
      int i, j;
      int a = -10, b = +10;
      int maximo, veces;
```

Código 7.9. (Cont.)

```
//inicializacion aleatoria de la matriz X
      for(srand(time(NULL)) , i = 0 ; i < _N ; i++ , srand(rand())) {
            for(j = 0 ; j < _M ; j++) {
                  X[i][j] = rand() % (b - a + 1 ) + a;
            }
      }

      printf("MATRIZ X\n");
      for(i = 0 ; i < _N ; i++) {
            for(j = 0 ; j < _M ; j++) {
                  printf("%d\t", X[i][j]);
            }
            printf("\n");
      }

      maximo = X[0][0]; //primer posible valor para maximo
      veces = 0; //numero de veces que aparece el maximo
      for(i = 0 ; i < _N ; i++) {
            for(j = 0 ; j < _M ; j++) {
                  if(X[i][j] > maximo) {
                        veces = 1;
                        maximo = X[i][j];
                  } else if(X[i][j] == maximo) {
                        veces++;
                  }
            }
      }

      printf("EL MAXIMO ES %d Y OCURRE %d VECES \n", maximo, veces);

      return 0;
}
```

Código 7.10. Solución para el Ejercicio P7.5.

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

#define _N 10

int main(void) {
      int X[_N][_N], Y[_N][_N];
      int i, j;
      int a = -10, b = +10;
```

Código 7.10. (Cont.)

```
	for(i = 0; i < _N; i++) {
		for(srand(time(NULL)) , j = 0 ; j < _N ; j++ , srand(rand()))
		{
			X[i][j] = rand() % (b - a + 1) + a;
			if(i != j) {
				Y[i][j] = 0;
			}
			else {
				Y[i][j] = X[i][j];
			}
		}
	}

	printf("MATRIZ X\n");
	for(i = 0 ; i < _N ; i++) {
		for(j = 0 ; j < _N ; j++) {
			printf("%d\t", X[i][j]);
		}
		printf("\n");
	}

	printf("MATRIZ Y (DIAGONAL DE X)\n");
	for(i = 0 ; i < _N ; i++) {
		for(j = 0 ; j < _N ; j++) {
			printf("%d\t", Y[i][j]);
		}
		printf("\n");
	}

	return 0;
}
```

Código 7.11. Solución para el Ejercicio P7.6.

```
#include <stdio.h>
#include <stlib.h>
#include <time.h>

#define _N 100
#define _M 10

int main(void) {
	int X[_N][_M];
	float z[_M];
	int i, j;
	int a, b;
```

Código 7.11. (Cont.)

```
        srand(time(NULL));

        for(j = 0 ; j < _M ; j++) {
              a = -10.0 * (j + 1);
              b = +10.0 * (j + 1);
              z[j] = 0.0;
              for(srand(time(NULL)) , i = 0 ; i < _N ; i++ , srand(rand()))
              {
                    X[i][j] = rand() % (b – a + 1) + a;
                    z[j] += X[i][j];
              }
              z[j] /= _N;
        }

        printf("MATRIZ X\n");
        for(i = 0 ; i < _N ; i++) {
              for(j = 0 ; j < _M ; j++) {
                    printf("%d\t", X[i][j]);
              }
              printf("\n");
        }
        printf("VECTOR MEDIAS\n");
        for(j = 0 ; j < _M ; j++) {
              printf("%f\t", z[j]);
        }
        return 0;
  }
```

Código 7.12. Solución para el Ejercicio P7.7.

```
  #include <stdio.h>
  #include <stlib.h>
  #include <time.h>

  #define _fil_X          3
  #define _col_X          5
  #define _fil_Y          _col_X
  #define _col_Y          2
  #define _fil_Z          _fil_X
  #define _col_Z          _col_Y

  int main(void) {
        int X[_fil_X][_col_X], Y[_fil_Y][_col_Y], Z[_fil_Z][_col_Z];
        int i, j, k;
        int a = -5, b = +5;
```

Código 7.12. (Cont.)

```
	//INICIALIZAR X, Y, Z
		for(i = 0 ; i < _fil_X ; i++)
		{
			for(  srand(time(NULL)) , j = 0 ;
				j < _col_X ;
				j++, srand(rand()))
			{
				X[i][j] = rand() % (b - a + 1) + a;
			}
		}
		for(j = 0 ; j < _fil_Y ; j++)
		{
			for(  srand(time(NULL)) , k = 0 ;
				k < _col_Y ;
				k++ , srand(rand()))
			{
				Y[j][k] = rand() % (b - a + 1) + a;
			}
		}
		for(i = 0 ; i < _fil_Z ; i++)
		{
			for(k = 0 ; k < _col_Z ; k++)
			{
				Z[i][k] = 0;
			}
		}

	//MOSTRAR POR PANTALLA X E Y
		printf("MATRIZ X\n");
		for(i = 0 ; i < _fil_X ; i++)
		{
			for(j = 0 ; j < _col_X ; j++)
			{
				printf("%d\t", X[i][j]);
			}
			printf("\n");
		}

		printf("MATRIZ Y\n");
		for(j = 0 ; j < _fil_Y ; j++)
		{
			for(k = 0 ; k < _col_Y ; k++)
			{
				printf("%d\t", Y[j][k]);
			}
			printf("\n");
		}

	//CALCULAR PRODUCTO DE MATRICES
		printf("MATRIZ Z\n");
```

Código 7.12. (Cont.)

```
            printf("MATRIZ Z\n");
            for(i = 0 ; i < _fil_Z ; i++)
            {
                  for(k = 0 ; k < _col_Z ; k++)
                  {
                        for(j = 0 ; j < _fil_Y ; j++)
                        {
                              Z[i][k] += X[i][j] * Y[j][k];
                        }
                        printf("%d\t", Z[i][k]);
                  }
                  printf("\n");
            }

      //MOSTRAR POR PANTALLA Z
            printf("MATRIZ Z\n");
            for(i = 0 ; i < _fil_Z ; i++)
            {
                  for(j = 0 ; j < _col_Z ; j++)
                  {
                        printf("%d\t", Z[i][j]);
                  }
                  printf("\n");
            }

            return 0;
      }
```

Cadenas de caracteres

8

8.1. Ejemplos sencillos resueltos

Ejemplo 8.1.*USO_CTYPE. Funciones para manejo de caracteres.*

Ejemplo 8.2.*BASICOCADENAS. Manejo básico de cadenas.*

Ejemplo 8.3.*IMPRIMIRCADENA.*

8.2. Proyectos propuestos

Proyecto8.1.*COPIARLETRAS.*

Proyecto8.2.*COPIARINVERTIDA*.

Proyecto8.3.*PALINDROMO*.

Proyecto8.4.*ORDENARLETRAS*.

Proyecto8.5.*CONTARPALABRAS*.

Proyecto8.6.*TRADUCTOROTAN*.*Ejercicio adicional*.

Una vez que se ha trabajado con *arrays* numéricos (ya sean vectores o matrices), esta práctica aborda un tipo de *arrays* muy especiales: las **cadenas de caracteres** (***strings***, en inglés). Una cadena de caracteres es un array de caracteres (variables de tipo **char**).

Cuando se trabaja con una array numérico, parece lógico que se declare ese array con una dimensión acorde con la cantidad de variable que necesitemos. Y habitualmente será una cantidad concreta y prefijada: matrices de 4 x 4, vector de 100 elementos, un sudoku de 9 x 9, etc.

Pero cuando se trabaja con cadenas de texto, la dimensión del array creado no es, habitualmente, un valor conocido previamente. Si, por ejemplo, creamos una variable para guardar el nombre y apellidos de un alumno… ¿cuántas variables tipo **char** necesitaremos para codificar todas las letras necesarias para ese nombre? Que no es lo mismo llamarse "Ana García", que llamarse "Ana María de los Ángeles García Martínez". ¿Qué longitud asignamos al array que deberá almacenar un nombre del que, a priori, desconocemos su longitud?

La solución a esta cuestión es sencilla: definir un carácter que signifique final de cadena. Y así, toda cadena de texto está formada por los caracteres que se encuentran entre el primer elemento de la cadena y hasta que se llega por primera vez al valor del carácter final de cadena. Ese carácter se llama carácter NULO, y se representa, como ya hemos visto que se hace con los caracteres especiales, con una barra invertida y seguida del carácter cero (`'\0'`). También se identifica por su valor en la tabla ASCII: valor 0.

Y así, cuando se trabajan con arrays de tipo **char**, no solamente hay que vigilar que no se llegue a un valor del índice mayor del permitido (los elementos van desde 0 hasta N – 1, donde N es la dimensión de la cadena de texto), sino que también hay que vigilar la llegada del carácter final de cadena.

Así las cosas, hay que distinguir entonces entre el literal `"a"`y el literal `'a'`. El primero es una cadena de texto formada por dos caracteres: el carácter `'a'`, cuyo ASCII es 97, y el carácter `'\0'`, cuyo ASCII es 0. El segundo es, simplemente, el carácter `'a'`: 1 byte. Y así, si se tiene la declaración

```
char cadena[] = "a";
```

y se lanza la sentencia

```
printf("%hd\n", sizeof(cadena));
```

se obtiene como salida el valor 2: porque serán 2 los bytes necesarios para almacenar esta cadena de texto.

Como se verá a lo largo de esta práctica, existen numerosas funciones preparadas para trabajar con caracteres y sus cadenas. La mayoría de estas funciones se encuentran en las bibliotecas `<stdio.h>`, (destacan las funciones `scanf`, `getchar` y `gets`) `<ctype.h>` (manejo individual de caracteres –no cadenas–) y `<string.h>` (manejo de cadenas).

El objetivo de esta práctica es que el alumno aprenda a utilizar los caracteres y las cadenas de caracteres y el conjunto de funciones para su manejo. En cierta medida, podríamos considerar a las cadenas de texto como un nuevo tipo de dato, donde su dominio es el que se puede formar con cualquier conjunto finito y ordenado de caracteres y cuyos operadores son los definidos en la biblioteca `string.h`. El manejo de arrays tipo **`char`** (cadenas de caracteres) difiere sustancialmente del manejo de arrays de cualquier otro tipo de dato; y ello justifica la existencia de una colección de funciones propias para estas cadenas.

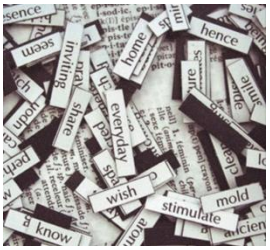
Como suele ser habitual, comenzamos con ejemplos sencillos para que el alumno tenga unos conceptos básicos del funcionamiento y manejo de los caracteres y de las cadenas de caracteres. A continuación, el alumno deberá ser capaz de programar los siguientes ejercicios:

1. Extraer las letras de una cadena de caracteres origen y copiarlas a otra cadena destino, poniéndolas en mayúsculas.

2. Copiar el texto de una cadena de caracteres a otra de manera invertida. También se propone una extensión para identificar si una palabra es un palíndromo.

3. Copiar todas las letras de una cadena origen a otra destino, de tal forma que primero queden las letras pares y seguidamente las letras impares.

4. Contar las palabras de un texto, omitiendo espacios en blanco y signos de puntuación.



Esta práctica también incluye un ejercicio complementario de carácter opcional y con una complejidad mayor. En concreto se debe implementar un traductor empleando el alfabeto OTAN.

8.1. EJEMPLOS SENCILLOS RESUELTOS

Antes de comenzar con la práctica es recomendable que el alumno lea y comprenda unos ejemplos sencillos que le introducirán en el uso de caracteres y de cadenas de caracteres y las funciones que permiten su manejo.

Ejemplo 8.1. USO_CTYPE.Funciones para manejo de caracteres.

Como ya sabe, una variable de tipo**char** ocupa 1 byte en memoria y se declara con la siguiente sentencia:

```
char ch;
```

Podemos asignar a la variable chalmacene el carácter'A'de la siguiente forma:

```
char ch = 'A';
```

NOTA: Los literales de tipo **char** se representan entre comillas simples: 'A'(letra A mayúscula); '3'(dígito 3); '\n'(carácter salto de línea); '\0'(carácter nulo). Cada carácter tiene un valor ASCII. Al final del capítulo podrá encontrar un anexo con la tabla de caracteres ASCII.

También vimos que una manera de leer desde teclado un carácter era utilizando la función scanf, el correspondiente especificador de formato (%c) y la dirección de memoria de la variable (&ch):

```
scanf("%c",&ch);
```

NOTA: Cuando la función scanfse utiliza para leer por teclado un valor de tipo **char**con frecuencia aparecen problemas, ya que esta función no toma la entrada directamente desde teclado: realmente toma la información que está disponible en el buffer. Un consejo útil para evitarlo es insertar entre la primera de las comillas y el carácter % un espacio en blanco.

Una de las bibliotecas más utilizadas para el manejo de caracteres es <ctype.h>, donde existen diversas funciones que nos permiten manipular variables de tipo carácter y obtener información acerca de las mismas. A continuación a través de un sencillo ejemplo vamos a introducir las distintas funciones que proporciona esta biblioteca.

Creamos un nuevo proyecto denominado **USO_CTYPE**. Debemos desarrollar un programa que lea caracteres por teclado hasta que el usuario introduzca `'@'`. Al finalizar el programa nos debe indicar el número de veces que se ha introducido una letra, un digito, un carácter alfanumérico (letra ó digito), un signo de puntuación y un espacio de texto (ya sea espacio en blanco, tabulador, retorno de carro, etc.). Antes de programar, es necesario revisar la siguiente tabla donde se resume las funciones incluidas en <ctype.h>. La mayoría de estas funciones reciben el código ASCII de un carácter y devuelve el valor 0 ó 1 en función de que el carácter pertenezca al grupo que define dicha función. En Código 8.1. mostramos una posible implementación de este programa propuesto.

FUNCIÓN	***SIGNIFICADO***	***EJEMPLOS***		
`int isalpha(int ch);`	Comprueba si **ch** es una **letra**	`isalpha('r') → 1`	`isalpha('A') → 2`	`ispalpha('9') → 0`
`int isdigit(int ch);`	Comprueba si **ch** es un **dígito**	`isdigit('9') → 1`		`isdigit('r') → 0`
`int isalnum(int ch);`	Comprueba si **ch** es **alfanumérico** (letra ó dígito)	`isalnum('r') → 1`	`isalnum('r') → 1`	`isalnum(':') → 0`
`int iscntrl(int ch);`	Comprueba si **ch** es **carácter de control** ([0…31,127] en ASCII)	`iscntrl('\n') → 1`		`iscntrl('r') → 0`
`int ispunct(int ch);`	Comprueba si **ch** es un **signo de puntuación**	`ispunct(':') → 1`		`ispunct('9') → 0`
`int isspace(int ch);`	Comprueba si **ch** es un **espacio de texto**	`isspace('') → 1`	`isspace('\n') → 1`	`isalnum('r') → 0`
`int islower(int ch);`	Comprueba si **ch** es una **letra minúscula**	`islower('r') → 1`		`islower('A') → 0`
`int isupper(int ch);`	Comprueba si **ch** es un **letra mayúscula**	`isupper('R') → 1`		`isupper('a') → 0`
`int isgraph(int ch);`	Comprueba si **ch** es un **carácter imprimible**	`isgraph('9') → 1`		`isgraph('\n') → 0`
`int isascii(int ch);`	Comprueba si **ch** tiene **código ASCII < 128**	`isascii('r') → 1`		`isascii('±') → 0`
`int isxdigit(int ch);`	Comprueba si **ch** es un **dígito hexadecimal**	`isxdigit('0') → 1`	`isxdigit('F') → 1`	`isxdigit('h') → 0`
`int tolower (int ch);`	Convierte **ch** de **mayúscula a minúscula**	`tolower('A') →'a'`		`tolower('b') →'b'`
`int toupper(int ch);`	Convierte **ch** de **minúscula a mayúscula**	`toupper('a') →'A'`		`toupper('B') →'B'`

Tabla 8.1. Funciones incluidas en la biblioteca `ctype.h`para el manejo de caracteres (en realidad, salvo las dos últimas, las demás no son funciones, sino macros: pero no podemos ahora entrar en esta cuestión).

Con todo ello, el programa propuesto para este proyecto es el siguiente:

Código 8.1. Programa propuesto en Ejemplo 8.1.

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

int main(void) {
      char ch;
      char parar = '@'; //caracter para detener lectura
      short l, d, ld, s, e;
      //contadores->   l:letras;
      //               d: digitos;
      //               ld: letras o digitos(alfanumerico);
      //               s: signos de puntuacion;
      //               e: espacios de texto

l=d = ld= s= e=0; //inicializar contadores

printf("Introduzca un caracter y pulse INTRO: \n");

do {
      fflush(stdin); scanf("%c", &ch);

      // tambien se puede usar: ch=getchar(); de <stdio.h>
      //ch=getche(); ch=getch(); de <conio.h> (solo Windows)

      if(isalpha(ch))        l++;  //contar letras
      if(isdigit(ch))        d++;  //contar digitos
      if(isalnum(ch))        ld++; //contar alfanumericos
      if(ispunct(ch))        s++;  //contar signos de puntuacion
      if(isspace(ch))        e++;  //contar espacios de texto

}while(ch != parar); //valor centinela para finalizar el programa

printf("\nEl numero de letras es: %hd\n",l);
printf("\nEl numero de digitos es: %hd\n",d);
printf("\nEl numero de caracteres alfanumericos: %hd\n",ld);
printf("\nEl numero de signos de puntuacion: %hd\n",s);
printf("\nEl numero de espacios de texto: %hd\n", e);

return 0; //el programa termina correctamente
}
```

En este programa se ha empleado un bucle **do-while** para realizar una lectura continuada de caracteres hasta introducir `'@'` y, dentro de dicho bucle, se han definido 5 estructuras de tipo **if**(independientes unas de otras: cada nuevo carácter recibido por teclado, es testeado por cada una de las cinco funcionesisalpha(), isdigit(), isalnum(), ispunct() e ispace()). Cada vez que, una función devuelve un valor verdadero se incrementa su correspondiente contador. Así, por ejemplo, si se introduce `'-'`(`ch = '-'`) únicamente ispunct(ch)devolverá un 1 y por tanto el

valor de la variables se verá incrementado en una unidad (s++), mientras que el resto de contadores no son incrementados.

La función `scanf()`espera la entrada de un carácter más el carácter INTRO, por lo que no resulta cómoda cuando se espera únicamente un carácter. Otra manera de leer caracteres, aunque también exige pulsar *Intro*, es mediante la función `getchar()`de la biblioteca `<stdio.h>`:

```
ch=getchar();
```

Aunque, al igual que la función `scanf()`, presenta problemas con el buffer de teclado y, por esta razón, suele ser habitual (y necesario) vaciar el buffer antes de leer un carácter (ya sea con `scanf()` o con `getchar()`). Esto es:

```
fflush(stdin); //limpiar buffer de entrada antes de leer un carácter
```

En el caso de programas ejecutados en entorno Windows, es muy útil emplear las funciones `getche()` y `getch()` de la biblioteca `<conio.h>`. Ambas funciones no presentan los problemas que existen al emplear `scanf()`y `getchar()`para la lectura de caracteres. Sin embargo estas funciones sólo se pueden usar bajo sistema operativo Windows, por lo que generalmente no haremos uso de ellas con objeto de que nuestros programas no sean dependientes del sistema operativo donde se ejecutan.

Ejemplo 8.2. BASICOCADENAS. Manejo básico de cadenas

Ya ha quedado dicho que una **cadena de caracteres** es un array de variables de tipo **char**, cuyo último elemento es el carácter nulo (carácter `'\0'` ó carácter ASCII de valor 0). Este carácter indica donde termina la cadena.

Suponga que desea declarar una cadena de caracteres denominada *cadena* para almacenar el texto `"Buenas tardes"`. En este caso, la cadena tiene un total de 13 caracteres (12 letras y 1 espacio), por tanto, diremos que la **longitud** del vector **cadena** debe ser **13**pero el **número de bytes** que ocupa es igual a **14** (*13 caracteres -bytes- para el texto más un byte el carácter nulo*).

A continuación se indican las distintas formas de crear la cadena de caracteres y asignarle el texto anterior:

1. Asignación elemento a elemento (carácter a carácter)

```
char cadena[14]={'B','u','e','n','a','s','','t','a','r','d','e','s','\0'};
```

En este caso se asigna cada uno de los caracteres individuales a cada uno de los elementos decadena. Podemos ver que el último elemento es el carácter nulo (es decir, `cadena[14]='\0';`). El compilador de C no puede determinar el final de la cadena si no se incluye el carácter nulo.

NOTA: Para un tipo de dato cadena de caracteres, los literales se expresarán en comillas dobles: `"Buenas tardes"`, `"123"`. Toda cadena de texto tiene tantos caracteres como se ven más uno final, que cierra la cadena, que es el carácter de fin de cadena ó`'\0'`cuyo código ASCII es 0. Así, por ejemplo, el carácter `'A'` se diferencia de `"A"`en que esta última cadena será de 2 bytes: uno para el código ASCII de la letra a mayúscula (65) y otro para el carácter nulo (0).

Y así, si tenemos el siguiente código:

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

intmain(void)
{
      charcad[] = "Buenas tardes";
      printf("Bytes que ocupa la cadena: %hd\n", sizeof(cad));
      printf("Longitud de la cadena: %hd\n\n", strlen(cad));
      return 0;
}
```

Tendremos la siguiente salida por pantalla;

```
Bytes que ocupa la cadena: 14
Longitud de la cadena: 13
```

Que indica que, efectivamente, la longitud de la cadena es de 13 caracteres (es el valor que ofrece la función `strlen()`), pero para codificarla en memoria hemos necesitado 14, donde ese carácter decimocuarto es el carácter final de cadena.

2. Asignación mediante comillas dobles

```
charcadena[14] = "Buenas tardes";
```

Esta forma de proceder es más adecuada y habitual. Ahora seráel compilador de C quien se encargue de introducir el carácter nulo final.

A la hora de imprimir la cadena sería tan sencillo como:

```
printf("%s",cadena);
```

Si para el array tipo **char**se emplea una mayor dimensión, por ejemplo 20, la cadena irá desde el elementoo variable `cadena[0]`que tendrá el valor `'B'` hasta el elemento o variable `cadena[13]` que tendrá el valor `'\0'`. Los elementos restantes (desde `cadena[14]`, hasta `cadena[19]`) no forman parte del actual valor de la cadena de caracteres: no importan los valores que tomen. Cuando se imprima la cadena de texto no se mostrarán todos los valores que sigan al nulo.

A la hora de introducir cadenas de caracteres es posible hacerlo mediante dos procedimientos fundamentales:

1. A través de la función**scanf**

```
scanf("%s",cadena);
```

2. A través de la función **`gets()`**: modo muy habitual y muy poco recomendable. Mire lo que se puede leer, por ejemplo, en la manual page de Linux, al solicitar información sobre esta función: "***Never use gets()****. Because it is impossible to tell without knowing the data in advance how many characters gets() will read, and because gets() will continue to store characters past the end of the buffer,* ***it is extremely dangerous to use****. It has been used to break computer security.* ***Use fgets() instead****."*Para dar valor a una cadena con la función `gets()`, la sintaxis es muy sencilla:

```
char cadena[20];
gets(cadena);
```

Pero recuerde esa severa advertencia sobre su uso (¿qué ocurre si el usuario introduce una entrada de longitud mayor a 19?). Se nos recomienda, a cambio, el uso de la función `fgets()`, que es una función que recibe 3 parámetros: (1) la cadena de texto a la que se desea asignar una entrada. (2) el tamaño de la cadena, o el número máximo de caracteres que se desean leer en esta entrada: la función asignará a la cadena, como máximo, un carácter menos del total indicado en este segundo parámetro; al final de la entrada, se asignará al siguiente byte el carácter nulo. (3) el origen de la entrada: si se desea cargar la información desde el teclado, como es nuestro caso ahora, el origen será **`stdin`**. Esta llamada a la función `fgets()` tendrá un efecto muy similar al obtenido con la anterior, mostrada con la función `gets()`:

```
char cadena[20];
fgets(cadena , 20 , stdin);
```

Así, si la entrada es de longitud mayor a 19, simplemente no se cargará completa en la cadena de texto: porque hacerlo implicaría violación de memoria. Esta función, a cambio, tiene una

pega respecto a `gets()`: si la entrada es de longitud menor que 19, y el usuario termina su entrada pulsando la tecla Intro, ese carácter queda como uno más de la cadena de entrada.

El nuevo estándar C11 ofrece una nueva función más sencilla de usar: la función `gets_s()`, que recibe dos parámetros: los dos primeros indicados en la descripción de la función `fgets()`. Además no da el problema descrito con el carácter Intro de fin de entrada (valor ASCII 0xA, ó 10 en base decimal). Si el IDE de programación que usted utiliza dispone de un compilador y de las funciones incluidas en el nuevo estándar del lenguaje C, entonces podrá hacer uso de ella: de lo contrario, tendrá que contentarse con estas dos previamente descritas.

Es más cómodo emplear la función `gets_s()`–o la función `gets()`–que la función`scanf()`.El uso correcto de `scanf()` para obtener una cadena de texto requiere de alguna explicación adicional: por ejemplo, porque esta función termina de leer la cadena en cuanto encuentra el carácter espacio en blanco: este comportamiento puede modificarse: puede consultar referencias más detalladas para eso.

Hay que tener en cuenta que el número de caracteres introducidos no debeser igual o mayor al número de bytes que se han reservado en memoria para la cadena de caracteres (recuerden que una posición de memoria la usa el carácter nulo). En caso contrario, tendremos violaciones de memoria y el funcionamiento del programa será impredecible.

Creamos un nuevo proyecto denominado **BASICOCADENAS** con el código sugerido en Código 8.2.

Código 8.2. Programa propuesto en Ejemplo 8.2.

```
#include<stdio.h>

int main(void)
{
      char cad1[20] ={'B','u','e','n','o','s',' ','d','i','a','s','\0'};
      charcad2[10]="OVNI";
      char cad3[50];

//leer texto para cadena3
      gets(cad3);

//imprimir las tres cadenas
      printf(" %s\n", cad1);
      printf(" %s\n",cad2);
      printf(" %s\n",cad3);

      return 0;
}
```

Este sencillo programa declara tres cadenas distintas (`cad1, cad2ycad3`) y asigna las dos primeras (`cad1y cad2`) a los textos `"Buenos dias"` y `"OVNI"`, respectivamente, mediante dos procedimientos distintos. La tercera cadena es introducida por el usuario y para ello se ha empleado la función `gets()`. Finalmente se muestran por pantallalas tres cadenas. A continuación se muestra el resultado obtenido al introducir el texto `"Objeto Volante No Identificado"`para la tercera cadena.

Ejemplo 8.3. IMPRIMIRCADENA.

El manejo de cadenas de caracteres es similar al caso de arrays numéricos, con la excepción de que estamos trabajando con datos de tipo **char**y la longitud viene delimitada por el carácter nulo (`'\0'`). Así, por ejemplo, el programa mostrado en Código 8.3., correspondiente al ejemplo **IMPRIMIRCADENA**, nos permite recorrer los distintos elementos de una cadena de caracteres e imprimirlos uno a uno por pantalla. Para ello se ha utilizado un bucle **while**controlado por la condición: (`cadena[i] != '\0'`: en línea `/*01*/`); es decir, el bucle finaliza cuando se llega al carácter nulo o de fin de cadena. Dentro del bucle se va imprimiendo uno a uno los distintos caracteres que componen la cadena de texto.

Código 8.3. Programa propuesto en Ejemplo 8.3.

```
     #include<stdio.h>

     int main(void)
     {
           char cadena[50] = "Carpe Diem";
           short i = 0;

/*01*/     while(cadena[i] !='\0'){
                 printf("%c", cadena[i]);
                 i++;
           }
           return 0;
     }
```

Otra forma de hacerlo es utilizando bucles `for`:puede verlo en Código 8.4.

La condición de finalización del bucle es directamente `cadena[i]` (el valor del dato almacenado en el elemento i-ésimo del vector de caracteres), por tanto, finalizará cuando esté sea nulo.

NOTA: Recuerde que el valor ASCII del carácter nulo (`'\0'`) es 0.

Código 8.3. Programa propuesto en Ejemplo 8.3., utilizando estructura **for**.

```
#include <stdio.h>

int main(void)
{
	char cadena[50]="Carpe Diem";
	short i;

	for(i = 0; cadena[i]; i++) {
		printf("%c", cadena[i]);
	}

	return 0;
}
```

Por último, en la Tabla 8.2.se resumen algunas de las funciones más utilizadas para el manejo de caracteres y la conversión a valores numéricos. Todas ellas están incluidas en las bibliotecas <string.h>y <math.h>.

FUNCIÓN	*SIGNIFICADO*	*BIBLIOTECA*
strcpy(cad1,cad2);	**Copia** los caracteres de la cadenacad2a la cadena cad1	string.h
n=strlen(cad1);	Devuelve el **número de caracteres** (n) de la cadena cad1	string.h
strcat(cad1,cad2);	**Concatena** en cad1todos los caracteres de cad2después de los caracteres de cad1	string.h
s=strcmp(cad1,cad2);	**Compara** la cad1con cad2de tal forma que s>0 si cad1es mayor que cad2y s<0 si cad1es menor que cad2.	string.h
numero=atoi(cad)	**Convierte** los **caracteres** de cada número **entero** (numero es de tipo **int**)	math.h
numero=atof(cad)	**Convierte** los **caracteres** de cada número **double** (numero es de tipo **double**)	math.h

Tabla 8.2. Resumen de las funciones más utilizadas en las bibliotecas string.hy math.h.

NOTA: Hasta el momento, en los programas que hemos desarrollado para manejo de caracteres se ha trabajado fundamentalmente con el ASCII original de 128 caracteres (véase Anexo II. Código ASCII) y –muy a nuestro pesar– hemos evitado el uso de caracteres propios del castellano, como son las vocales con tilde y nuestra internacional letra ñ/Ñ (todos ellos recogidos en el código ASCII extendido Latin-I, Tabla 8.5). Así, por ejemplo, hemos tenido dificultades para mostrar por pantalla frases como las siguientes: `"Me llamo José Ramón y soy español"`, `"Está volando un avión del Ejército"`. Una manera muy sencilla de poder escribir directamente texto en castellano (considerando las particularidades de su alfabeto y su gramática) es emplear la función `setlocale()` de la biblioteca `locale.h`. Para ello, tras incluir esta biblioteca en nuestro programa, se deberá invocar, entre las líneas de código de la función `main()` y antes de la primera sentencia con la función `printf()`, a la función `setlocale()` de la siguiente forma:

```
setlocale(LC_CTYPE , "Spanish");
```

No es objeto de este libro de prácticas profundizar en más detalles sobre esta función. Únicamente se pretende darle las indicaciones mínimas para poder manejar sin problemas cadenas de caracteres en castellano. Para finalizar, a modo de ejemplo, le dejamos este código.

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

int main(void)
{
      setlocale(LC_CTYPE , "Spanish");
      printf("Ahora puedo mostrar tildes por pantalla, y la letra Ñ\ñ");
      printf("Mira si no: áéíóúÁÉÍÓÚ\n\n");
      printf("¡¡¡ Qué fácil era !!!\n");
      printf("Bastaba decirle que estábamos en ESPAÑA.....\n\n");
      return 0;
```

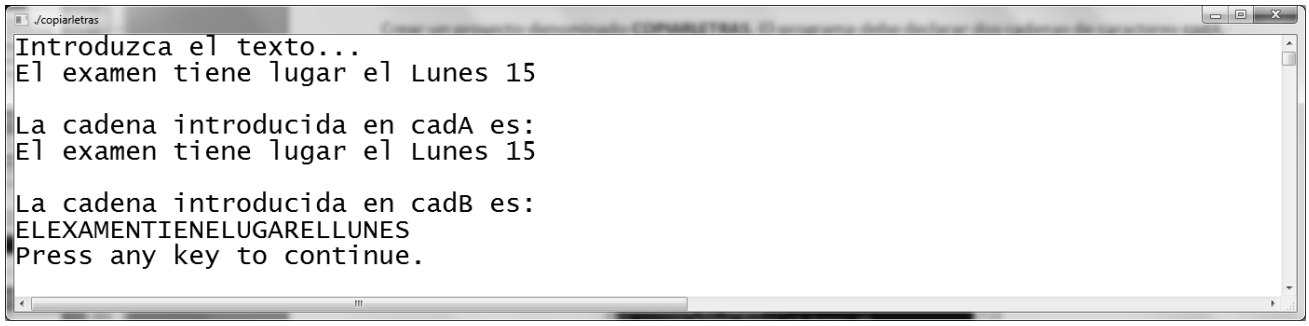
8.2. PROYECTOS PROPUESTOS

Proyecto 8.1. COPIARLETRAS

Creeun proyecto denominado **COPIARLETRAS**. En él deberá implementar un programa que declare dos cadenas de caracteres `cadA`y`cadB`dedimensión igual a 100 (indique el tamaño mediante una directiva **define**). A continuación debe realizar las siguientes operaciones:

1. Leer por teclado el texto introducido por el usuario–se recomienda, si no dispone del compilador con las incorporaciones de C11, utilizar la función `gets()`, si dispone del compilador, entonces haga uso de la función `gets_s()`– y almacenarlo en `cadA`. Introducir por ejemplo el siguiente texto: `"El examen tiene lugar el Lunes 15."`
2. Copiar en `cadB` todos los caracteres de `cadA` que sean alfabéticos (es decir, letras); el programa deberá copiar esos caracteres en mayúsculas.
3. Mostrar por pantalla ambas cadenas.

Atendiendo al texto de ejemplo introducido, el resultado deberá ser algo parecido a lo siguiente:



Proyecto 8.2. COPIARINVERTIDA

Creeun proyecto denominado **COPIARINVERTIDA**. En él deberá implementar unprograma quedeclare dos cadenas de caracteres `cadA`y `cadB`de dimensión 100 (indique el tamaño mediante una directiva **`define`**). A continuación debe realizar las siguientes operaciones:

1. Leer por teclado el texto introducido por el usuario y almacenarlo en `cadA`.
2. ***Copiar en cadB todos los caracteres de cadA en orden inverso***.
3. Mostrar por pantalla ambas cadenas.

En el caso de introducir en `cadA` el literal `"Alpha Beta"`el valor final de `cadB`será`"ateB ahplA"`

Proyecto 8.3. PALINDROMO

Tenemos que desarrollar un nuevo programa,en un nuevo proyecto denominado **PALINDROMO**,que permita introducir una cadena de texto y que indique si el texto introducido es un *palíndromo*: un palíndromo se lee igual de derecha a izquierda como de izquierda a derecha (sin considerar los espacios y signos de puntuación: introduzca las cadenas de texto sin acentuar las palabras). El programa deberá eliminar los espacios y signos de puntuación que introduzca el

usuario por teclado.Algunos ejemplos de palíndromos (palabras y frases completas) son los siguientes:

Reconocer
Radar
No deseo yo ese don
A cavar a Caravaca
Dabale arroz a la zorra el abad
Never odd or even
No it is opposition

Proyecto8.4.ORDENARLETRAS

Creamos un proyecto denominado **ORDENARLETRAS**. Realizamos ahora un programa que deberá declarar dos cadenas de caracteres `cadA`y `cadB`dedimensión 100 (indique el tamaño mediante una directiva **`define`**). A continuación debe realizar las siguientes operaciones:

1. Leer por teclado el texto introducido por el usuario y almacenarlo en `cadA`.
2. Copiar en `cadB`todas las letras de `cadA`, **poniendo primero las letras ubicadas dentro de la cadena en una posición de índice par (0, 2, 4,…) y, a continuación, las letras ubicadas en las posiciones de los índices impares (1, 3, 5…).**
3. Mostrar por pantalla ambas cadenas.

En el caso de introducir en la primera cadena el literal `"ABCDEFGHIJK"`tendremos que la segunda cadena valdrá `"BDFHJACEGIK"`.

Proyecto8.5. CONTARPALABRAS

Creamos un proyecto denominado **CONTARPALABRAS**. Queremos ahora implementar un programa que cuente las palabras incluidas en una cadena de entrada recibida por teclado.Este programa debe:

1. Leer por teclado el texto introducido por el usuario y almacenarlo en una cadena de caracteres (por ejemplo **`char`**`texto[200];`**).**
2. **Cuente el número de palabras que se han introducido omitiendo espacios y signos de puntuación.**
3. Mostrar por pantalla el número de palabras introducidas.

Probar el correcto funcionamiento del programa considerando, por ejemplo, los siguientes textos (donde hemos indicado los espacios en blanco mediante el carácter `''`):

`"Militar·e·Ingeniero"`

`"El·dia·de·la·Hispanidad·es·12·de·Octubre. "`

`"·····según·el·código·OTAN:·Kayak·es·un·misil·aire-superficie·(empieza·por·K)·"`

Que deberán dar como salida 3, 9 y 12 palabras, respectivamente.

IMPORTANTE: Este ejercicio tiene una complejidad mayor a los anteriores, por lo que será necesario emplear más tiempo durante su implementación.

Proyecto 8.6. TRADUCTOROTAN [EJERCICIO ADICIONAL]

El Alfabeto radiofónico es un lenguaje utilizado internacionalmente para radiocomunicaciones de transmisión de voz para marina, aviación, servicios civiles y militares. Fue establecido por la Organización de Aviación Civil Internacional (OACI), agencia de la ONU creada en 1944. También conocido como Alfabeto fonético OACI (ICAO en inglés), el alfabeto fonético aeronáutico es un sistema creado para poder dar mayor certeza a las radiocomunicaciones aeronáuticas.Su empleo resulta clave para deletrear códigos como pueden ser el número de identificación de un contenedor de carga o inclusive de una aeronave, o similar. La siguiente lista muestra este alfabeto:

A = Alpha	N = November
B = Bravo	O = Oscar
C = Charlie	P = Papa
D = Delta	Q = Quebec
E = Echo	R = Romeo
F = Foxtrot	S = Sierra
G = Golf	T = Tango
H = Hotel	U = Uniform
I = India	V = Victor
J = Juliet	W = Whiskey
K = Kilo	X = X-ray
L = Lima	Y = Yankee
M = Mike	Z = Zulu.

Se le pide que desarrolle un programa denominado **TRADUCTOR_OTAN** que permita traducir una palabra a alfabeto OTAN. El programa debe solicitar una palabra y mostrar su equivalente según el alfabeto OTAN.

PRUEBA: Si introduce la palabra `"AIRE"` se debe mostrar

```
Alpha
India
Romeo
Echo.
```

PRUEBA: Si introduce la palabra `"CARTAGENA"`se debe mostrar

```
Charlie
Alpha
Romeo
Tango
Alpha
Golf
Echo
November
Alpha
```

SOLUCIONES A LOS EJERCICIOS DE LOS PROYECTOS PROPUESTOS

Código 8.4. Solución al Ejercicio P8.1.

```
#include<stdio.h>
#include<ctype.h>
#define_N 100

int main(void)
{
      char cadA[_N];
      char cadB[_N];
      short i, j;

      printf("Introduzca el texto...\n");
      gets_s(cadA , _N);
      // gets(cadA); si no dispone del compilador C11

      for(i =0,j=0;cadA[i]&& i < N;i++){
            if(isalpha(cadA[i])){
                  cadB[j]=toupper(cadA[i]);
                  j++;
            }
      }
      cadB[j]='\0';

      printf("\nLa cadena introducida en cadA es:\n");
      printf("%s\n",cadA);
      printf("\nLa cadena introducida en cadB es:\n");
      printf("%s",cadB);

      return 0;
}
```

Código 8.5. Solución al Ejercicio P8.2.

```
#include<stdio.h>
#include<string.h>
#define _N 100

int main(void)
{
      char cadA[_N];
      char cadB[_N];
      short longitud;
      short i, j;

      printf("Introduzca el texto...\n");
      gets(cadA);
```

Código 8.5. (Cont.)

```
        longitud = strlen(cadA);

        for(i = 0, j = longitud - 1; cadA[i]&& i < N; i++){
              cadB[i] = cadA[j];
              j--;
        }
        cadB[i] = '\0';

        printf("\nLa cadena introducida en cadA es:\n");
        printf("%s\n",cadA);
        printf("\nLa cadena invertida en cadB es:\n");
        printf("%s",cadB);

        return 0;
  }
```

Código 8.6. Solución al Ejercicio P8.3.

```
  #include<stdio.h>
  #include<string.h>
  #include<ctype.h>
  #define _N 100

  int main(void)
  {
        char cadA[_N];    //cadena original.
        char cadB[_N];    //cadena invertida solo alfanumericos.
        short longitud;
        short i, j;

        printf("Introduzca el texto...\n");      gets(cadA);
                                                 // gets_s(cadA , _N);
        i = strlen(cadA) - 1;
        j = 0;

        while(i >= 0)
        {
              if(isalnum(cadA[i]))
              {
                    cadB[j++] = toupper(cadA[i]);
              }
              i--;
        }
        cadB[j] = '\0';   // se cierra la cadena recién asignada
```

Código 8.6. (Cont.)

```
        longitud = strlen(cadB);
    //comprobar si es un palindromo
        for(  i = 0, j = longitud - 1 ;
              (i <longitud)&&(cadB[i] == cadB[j]) ;
              i++, j--);
    // OBSERVE: la estructura for lleva punto y coma;
        printf("%s es un palindromo\n\n", i == longitud ? "SI" : "NO");

        return 0;
    }
```

Código 8.7. Solución al Ejercicio P8.4. (Primera implementación)

```
//Nota: Esta implementacion considera que todos los caracteres son letras

    #include<stdio.h>
    #include<string.h>
    #define_N 100

    int main(void)
    {

        charcadA[_N], cadB[_N];
        short i, j;

        printf("Cadena a...");
        gets(cadA);
        //gets_s(cadA, _N);

        for(i = 1, j = 0 ; i<strlen(cadA) ; i += 2 , j++)
        {
            cadB[j] = cadA[i];
        }

        for(i = 0; i<strlen(cadA) ; i += 2 , j++)
        {
            cadB[j] = cadA[i];
        }
        cadB[j] = 0;

        printf("Cadena a ... %s\n", cadA);
        printf("Cadena b ... %s\n", cadB);

        return 0;
    }
```

Código 8.8. Solución al Ejercicio P8.4. (Segunda implementación)

```
//Nota: Esta segunda implementacion ordena las letras en posiciones pares e
impares descartando todos aquellos caracteres introducidos que no son letras

     #include<stdio.h>
     #include<string.h>
     #define_N 100

     int main(void)
     {
           char cadA[_N]; //cadena original
           char cadB[_N]; //cadena final (primero letras pares y luego impares)
           char cadC[2][_N]; //la primera fila almacena las letras pares y
                             //la segunda almacena las impares
           short i,j,k;

           printf("Introduzca el texto...\n");      gets(cadA);
                                                 // gets_s(cadA , _N);

           for(i=0,j=0,k=0; cadA[i]; i++)
           {
                 if(isalpha(cadA[i]))
                 {
                       if(i%2==0)
                       { //posiciones pares
                             cadC[0][j] = cadA[i];
                             j++;
                       }
                       else
                       { //posiciones impares
                             cadC[1][k] = cadA[i];
                             k++;
                       }
                 }
           }

           cadC[0][j] = '\0';
           cadC[1][k] = '\0';

           strcpy(cadB,cadC[0]);  //copiar letras en posiciones pares
           strcat(cadB,cadC[1]);  //copiar letras en posiciones impares
           cadB[j+k]='\0';  // Esta sentencia NO es necesaria

           printf("\nLa cadena introducida en cadA es:\n");
           printf("%s\n",cadA);
           printf("\nLa copia en cadB es:\n");
           printf("%s", cadB);

           return 0;
     }
```

Código 8.9. Solución al Ejercicio P8.5.

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

#define _N 100

int main(void) {
      char cad[_N];
      short i, palabras = 0;

      printf("Cadena de entrada ... ");        gets(cad);
                                            // gets_s(cad , _N);
      if(isalnum(cad[0])) palabras++;

      for(i = 1 ; cad[i] ; i++)
      {
            if(isalpha(cad[i]) &&
                        (isspace(cad[i - 1]) || ispunct(cad[i - 1])))
            {
                  palabras++ ;
            }
      }

      printf("Numero de Palabras ... %hd\n", palabras);
      return 0;
}
```

Código 8.11. Solución al Ejercicio P8.6.

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

#define _D 20

int main(void)
{
      char otan[][10] = {"Alpha","Bravo","Charlie","Delta","Echo",
                  "Foxtrot", "Golf","Hotel","India","Juliet","Kilo",
                  "Lima","Mike","November","Oscar","Papa","Quebec",
                  "Romeo","Sierra","Tango","Uniform","Victor","Whiskey",
                  "X-ray","Yankee","Zulu"};
```

Código 8.11. (Cont.)

```
        char palabra[_D];
        int i;
        int correcto = 1;

        printf("Introduzca una palabra...\n");
        fgets(palabra, _D, stdin);

        //eliminar caracter de salto de linea
        if(strlen(palabra) < _D - 1)
              palabra[strlen(palabra) - 1] = 0;

        //Comprobar que todos son letras y pasar a mayusculas
        for(i = 0; palabra[i] && correcto == 1; i++){
              if(isalpha(palabra[i]))
              {
                    palabra[i] = toupper(palabra[i]);
              }
              else
              {
                    correcto = 0;
              }
        }

        if(correcto)
        {
              printf("\nEnalfabeto OTAN es:\n");
              for(i = 0; palabra[i]; i++){
                    printf("%s \n",otan[palabra[i]-'A']);
              }
        }
        else
        {
              printf("Palabra introducida incorrecta.");
        }

        return 0;
  }
```

ANEXO II. CÓDIGO ASCII

Los sistemas informáticos trabajan internamente con información de tipo numérica. Entonces, ¿cómo pueden presentar caracteres en pantalla? Es sencillo: se utiliza **codificación de caracteres**,donde se convierte el conjunto de caracteres en una representación numérica que garantiza que cada carácter tiene un único número que lo representa. Ahora bien, si deseamos que dos o más sistemas informáticos puedan intercambiar información de tipo carácter, es necesario que todos ellos tengan una misma codificación de caracteres. Así, en la década de 1960, la necesidad de estandarización llevó a la creación del código norteamericano estándar para el intercambio de información (**ASCII** o **American Standard Code for Information Interchange**).

En el código ASCII original, cada carácter está representado por 7 bits, por lo que se dispone de un total de 128 caracteres (desde 0 a 127 en formato decimal). Los números del 0 al 31 del código ASCII están asignados a caracteres de control utilizados para controlar dispositivos periféricos como, por ejemplo, impresoras. Así, el 12 representa la función de avance de papel/nueva página. Este comando indica a la impresora que pase directamente a la parte superior de la siguiente página.

Número decimal	***Carácter***	***Número decimal***	***Carácter***
0	nulo	16	escape de vínculo de datos
1	principio de encabezado	17	control de dispositivo 1
2	inicio de texto	18	control de dispositivo 2
3	fin del texto	19	control de dispositivo 3
4	fin de transmisión	20	control de dispositivo 4
5	consulta	21	confirmación negativa
6	reconocimiento	22	inactividad síncrona
7	campana	23	fin del bloque de transmisión
8	retroceso	24	cancelar
9	tabulación horizontal	25	fin del medio
10	avance de línea/nueva línea	26	sustitución
11	tabulación vertical	27	escape
12	avance página/nueva página	28	separador de archivos
13	retorno de carro	29	separador de grupos
14	desplazamiento hacia fuera	30	separador de registros
15	desplazamiento hacia dentro	31	separador de unidades

Tabla 8.3. Código ASCII para caracteres de control no imprimibles.

Los números 32 a 126 del código ASCII están asignados a caracteres básicos incluidos en el teclado (principalmente, alfabeto latino y signos de puntuación). El número 127 es el comando SUPRIMIR.

Número decimal	*Carácter*	*Número decimal*	*Carácter*	*Número decimal*	*Carácter*	*Número decimal*	*Carácter*
32	espacio	33	!	34	"	35	#
36	$	37	%	38	&	39	'
40	(	41	)	42	*	43	+
44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3
52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;
60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C
68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K
76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S
84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[
92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c
100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k
108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s
116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{
124	|	125	}	126	~	127	Supr

Tabla 8.4. Código ASCII para caracteres imprimibles

Debido al limitado número de caracteres del ASCII original, se definieron varios códigos de caracteres de 8 bits, entre ellos el **ASCII extendido.** Sin embargo, el problema de estos códigos de 8 bits es que cada uno de ellos se define para un conjunto de lenguas con escrituras semejantes y por tanto no dan una solución unificada a la codificación de todas las lenguas del mundo. El código ASCII extendido incluye los 128 caracteres existentes en el código ASCII original y agrega otros 128 caracteres para obtener un total de 256. Incluso con estos caracteres adicionales, muchos idiomas poseen símbolos que no pueden condensarse en 256 caracteres. Por esta razón, hay variantes del código ASCII para abarcar los caracteres y símbolos de ciertas regiones.Por ejemplo, la tabla que se muestra a continuación es usada por muchos programas para idiomas usados en Norteamérica, Europa Occidental, Australia y África, y es conocida como ISO 8859-1 (Latín I).

Número decimal	Carácter	Número decimal	Carácter	Número decimal	Carácter	Número decimal	Carácter
128	€	129	ü	130	‚	131	ƒ
132	„	133	…	134	†	135	‡
136	ˆ	137	‰	138	Š	139	‹
140	Œ	141	ì	142	Ž	143	Å
144	É	145	‘	146	’	147	“
148	”	149	•	150	–	151	—
152	˜	153	™	154	š	155	›
156	œ	157	Ø	158	ž	159	Ÿ
160	á	161	¡	162	¢	163	£
164	¤	165	¥	166	¦	167	§
168	¨	169	©	170	ª	171	«
172	¬	173	-	174	®	175	¯
176	°	177	±	178	²	179	³
180	´	181	µ	182	¶	183	·
184	¸	185	¹	186	º	187	»
188	¼	189	½	190	¾	191	¿
192	À	193	Á	194	Â	195	Ã
196	Ä	197	Å	198	Æ	199	Ç
200	È	201	É	202	Ê	203	Ë
204	Ì	205	Í	206	Î	207	Ï
208	Ð	209	Ñ	210	Ò	211	Ó
212	Ô	213	Õ	214	Ö	215	×
216	Ø	217	Ù	218	Ú	219	Û
220	Ü	221	Ý	222	Þ	223	ß
224	à	225	á	226	â	227	ã
228	ä	229	å	230	æ	231	ç
232	è	233	é	234	ê	235	ë
236	ì	237	í	238	î	239	ï
240	ð	241	ñ	242	ò	243	ó
244	ô	245	õ	246	ö	247	÷
248	ø	249	ù	250	ú	251	û
252	ü	253	ý	254	þ	255	Ÿ

Tabla 8.5. Código ASCII extendido (ISO 8859-1, Latin-I)

Funciones: Parámetros por valor

9

9.1. Ejemplos sencillos resueltos

Ejemplo 9.1. *CUADRADO. Nociones básicas de funciones.*

Ejemplo 9.2. *LOCALGLOBAL. Ámbito y vida de las variables.*

Ejemplo 9.3. *FACTORIAL. Concepto básico de recursividad.*

9.2. Proyectos propuestos

Proyecto 9.1. *FUNCION_ECUACION.*

Proyecto 9.2. *FUNCION_PRIMOS*.

Proyecto 9.3. *FUNCIONES_TARTAGLIA*.

Proyecto 9.4. *INTERESCOMPUESTO*.

Proyecto 9.5. *LEERENTERO*. *Ejercicio adicional.*

Todo el código de los programas que hemos realizado hasta ahora se ha insertado íntegramente dentro de la llamada función `main`, que es la función principal. En algunos de los problemas que hemos estado trabajando en prácticas anteriores (por ejemplo, el problema del cañón antiaéreo) ha sido necesario escribir multitud de líneas de código en esa función `main`. Sin embargo, es posible y muy recomendable descomponer el problema inicial en subproblemas más sencillos y estructurar así luego el código final en distintos bloques diferenciados (*módulos*) que permitan entender y desarrollar el programa siguiendo una ***metodología estructurada y modular***. A la hora de definir estas partes diferenciadas es muy habitual hacer uso de **funciones**: una *función* es un grupo o bloque de instrucciones identificado mediante un nombre, que realiza una tarea concreta y determinada, y que puede ofrecer finalmente un valor resultante obtenido a partir de otros valores iniciales recibidos como parámetros de entrada.

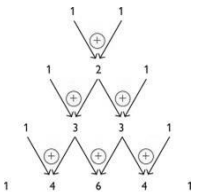
Usted ya conoce muchas funciones, de las que ha hecho uso frecuentemente; por ejemplo: `sqrt()`, para el cálculo de la raíz cuadrada de un número; ó `cos()` para el cálculo del coseno de un ángulo expresado en radianes, etc. Pero hasta el momento sólo ha definido una función: la función `main`. Ahora vamos a proponer ejercicios donde usted podrá crear otras funciones y hacer luego uso de ellas allá donde las necesite. El objetivo de este capítulo es que usted aprenda a declarar, a definir o implementar, y a utilizar funciones en un entorno de programación. Además, junto con la implementación de funciones, se desarrollaran simultáneamente los conceptos de **tiempo de vida de variables** y **recursividad**. Antes de comenzar con los ejercicios de esta práctica, usted puede revisar las nociones y fundamentos de las funciones a través de una serie de ejemplos sencillos. A continuación, el alumno deberá ser capaz de dar solución a un total de cinco ejercicios que se le plantean: la mayoría de ellos ya han sido abordados en prácticas anteriores; ahora hay que resolverlos con metodología de programación modular.

Los ejercicios son los siguientes:

1. Implementar una función que resuelva una ecuación de primer grado cuyos parámetros de entrada sean los coeficientes que definen la ecuación.

2. Desarrollar funciones para determinar si un número entero es primo y calcular el número de enteros primos que hay en un intervalo definido por dos números.

3. Programar las funciones necesarias para mostrar el triángulo de Tartaglia por pantalla.



4. Implementar una función que determine el capital obtenido en una inversión considerando interés compuesto.

5. Desarrollar una función que permita leer cada uno de los caracteres introducidos por teclado, omitiendo todos aquellos que no sean dígitos, y los almacene en una cadena.

9.1. EJEMPLOS SENCILLOS RESUELTOS

Antes de comenzar con la práctica es recomendable que el alumno lea y comprenda unos ejemplos sencillos que le permitirán introducirse en la declaración, definición o implementación, y llamada o invocación de funciones.

Ejemplo 9.1. CUADRADO. Nociones básicas de funciones.

Desde nuestros primeros pasos en programación hemos empleado funciones. Recordamos ahora el código de nuestro primer programa:

Código 9.1. Nuestro primer programa.

```
#include <stdio.h>

int main(void)
{
      printf("Este es mi primer programa.");
      return 0;
}
```

En este ejemplo tan sencillo, y como en cualquier programa, existe una función principal (`main`) que contiene el código que ejecuta nuestro programa. En este caso, se invoca a la función `printf()` cuya tarea es imprimir por pantalla la cadena de caracteres `"Este es mi primer programa."`. Esta cadena de texto es el parámetro de entrada para la función `printf()`.

De manera general, una **función** es un *bloque de código o conjunto de sentencias referenciadas con un nombre o identificador, que realizan una tarea específica y que puede retornar un valor a partir de una serie de valores recibidos como parámetros de entrada*. Las funciones pueden tomar parámetros (valores de entrada) que modifiquen su comportamiento. Las funciones son utilizadas para descomponer un problema en tareas más simples y para implementar operaciones que son comúnmente utilizadas durante un programa y de esta manera reducir la cantidad de código. Cuando una función es invocada se le pasa el control del programa y, una vez que ésta finaliza con su tarea, el control es devuelto al punto desde el cual la función fue llamada.

Ejemplo sencillo para la declaración, definición e invocación de una función

Para comenzar, vamos a considerar que se desea declarar, definir e invocar a la función `cuadrado()` que deberá devolver el cuadrado de un número real (de tipo **`double`**), es decir, la función `cuadrado()` aceptará un valor de entrada de tipo **`double`** y devolverá un valor, en este caso también de tipo **`double`**. Hemos utilizado los verbos declarar, definir e invocar: las tres cosas hay que hacer si queremos crear una nueva función y ponerla en ejecución. **Primero es necesario declarar la función** (indicar, en una sentencia declarativa, su tipo, su nombre y sus parámetros de entrada). Pero con declarar la función no basta: es necesario también **definir la función** (es decir, explicitar el conjunto de sentencias que realizan la tarea para la cual se ha creado la función).

Finalmente, una función declarada y definida ya puede ser invocada; **invocaremos o llamaremos a la función** siempre que deseemos ejecutar el código de su definición: para ello indicaremos entre paréntesis, y después de escribir el nombre de la función, las expresiones que definan los distintos parámetros que la función recibirá como valores de entrada.

Primero, a continuación se muestra la **declaración de la función** que queremos implementar.

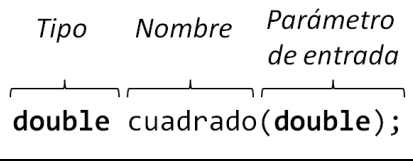


Figura 9.1. Declaración de la función `cuadrado()`.

Así, queda declarada una función de tipo **`double`**, llamada `cuadrado()` y que, como parámetros de entrada, tiene una variable de tipo también **`double`**.

Tenga en cuenta que :

(1) la declaración de una función es una sentencia: termina por tanto con punto y coma;

(2) la función se debe declarar fuera de cualquier otra función y siempre antes de hacer uso de ella;

(3) también se puede indicar, en la declaración, el nombre de cada una de las variables que son parámetros de entrada (**`double`** `n`). Puede verlo en Figura 9.2.

(4) y recuerde que toda función requiere ser declarada antes de su definición.

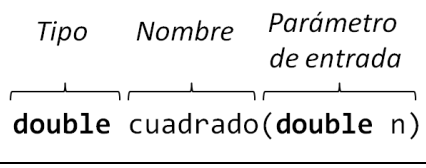


Figura 9.2. Declaración de la función `cuadrado()` indicando el nombre del parámetro de entrada.

Una vez que se ha declarado la función, se procede con la **definición de la función**:

Dentro de la función implementada, y como suele ser habitual en cualquier función, hay tres partes principales:

1. *Declaración de variables locales*. Las variables declaradas dentro de una función son locales a ella y por tanto se dice que son variables de **ámbito local**. Así, por ejemplo, la variable `m` (ver Figura 9.3.) es de ámbito local para la función `cuadrado()`. Los parámetros de entrada de la función son también, de hecho, variables locales a la función: no puede por tanto utilizar, como nombre de una variable local de la función, un nombre ya utilizado en uno de los parámetros.
2. *Cuerpo de la función*, que contiene las sentencias que llevarán a cabo la tarea para la que ha sido creada la función. En nuestro ejemplo (ver Figura 9.3.), sería la realización de la operación producto `n * n` y la asignación de su resultado a la variable local `m`.

3. *Sentencia* **return**. Esta sentencia fuerza la salida de la función y devuelve inmediatamente el control del programa a la función que la invocó. Si la función es de un tipo distinto a **void**, debe existir al menos una sentencia **return** donde se devuelva el valor de una expresión del mismo tipo que el especificado para la función bajo estudio. En el ejemplo analizado, la función `cuadrado()` devuelve un valor de tipo **double** que está almacenado en la variable local `m` (de tipo **double**).

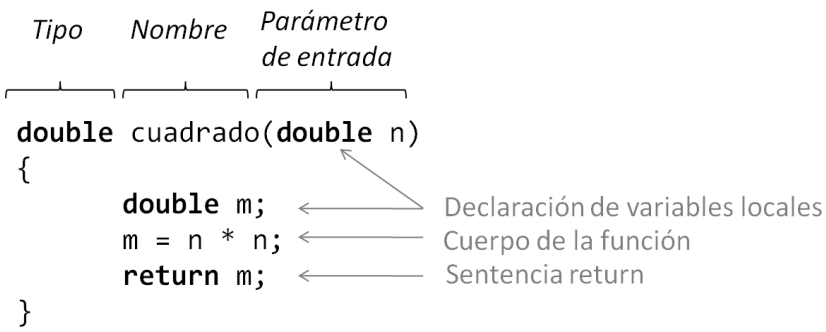


Figura 9.3. Definición de la función `cuadrado()`.

Una vez que tenemos la función declarada y definida podemos hacer uso de ella en la función principal de nuestro programa, o desde cualquier otra función, sin más que llamándola o invocándola. Por ejemplo, la siguiente sentencia invoca a la función implementada para calcular el cuadrado del valor codificado en la variable `a`. El valor que devuelve la función queda guardado en la variable `b`.

```
double a = 3.3;
double b;
b = cuadrado(a); ← Invocación/llamada a la función
```

Figura 9.4. Innovación o llamada a la función `cuadrado()` para calcular el cuadrado del valor codificado en la variable `a` y guardar el resultado correspondiente en la variable `b`.

Por último, el Código 9.2. recoge la implementación de un programa que primero solicita al usuario un número por teclado y, tras introducirlo, muestra luego por pantalla el cuadrado de dicho número. El cálculo de dicho valor cuadrado se realiza haciendo uso de la función `cuadrado()` desarrollada por el usuario. Como se puede ver Código 9.2., la línea `/*01*/` corresponde a la declaración de la función `cuadrado()`; mientras que las líneas de código entre `/*02*/` y `/*03*/` corresponden a la definición de dicha función.

Se ha definido la función **double** `cuadrado(`**double**`)` de una manera muy 'amigable' ya que se ha declarado una variable local `m` que almacena el resultado del cuadrado del dato de entrada. En realidad esta variable no es necesaria para el correcto funcionamiento del programa. Una mejor

solución (o al menos más frecuente) se muestra en Código 9.3., donde directamente se devuelve el cuadrado del dato de entrada, sin hacer uso de una variable local.

Código 9.2. Programa planteado en Ejemplo 9.1.

```
           #include <stdio.h>

     //declaracion de la funcion
/*01*/     double cuadrado(double);

           //programa principal
           int main(void)
           {
                 double a;
                 double b;

                 printf("Introduzca un numero...");
                 scanf(" %lf", &a);

                 b = cuadrado(a); //llamada a la funcion

                 printf("Su cuadrado es... %lf", b);

                 return 0;
           }

     //definicion de la funcion
/*02*/     double cuadrado(double n)
           {
                 double m;
                 m = n * n;
                 return m;
/*03*/     }
```

Código 9.3. Otra implementación para la función cuadrado del Ejemplo 9.1.

```
           double cuadrado(double n)
           {
                 return n * n;
           }
```

Una vez definida esta función, podremos invocarla donde queramos en nuestro programa. Como la función es de tipo **double**, en aquella expresión que contenga una llamada a esta función se considera que en esa invocación hay un valor de tipo **double**.

Supongamos que queremos hacer un programa que calcule el volumen de un cilindro del que conocemos el radio de su base y la altura. Si contamos con la implementación de la función

cuadrado() que ya hemos visto, el programa, en su función main(), podría ser algo así como lo que se muestra en Código 9.4.

Código 9.4.

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

         int main(void)
         {
            double radio, altura, volumen;
            printf("Valor del radio ..... ");  scanf(" %lf", &radio);
            printf("Valor de la altura .. ");  scanf(" %lf", &altura);

/*01*/      volumen = M_PI * cuadrado(radio) * altura;
            printf("Volumen del cilindro: %lf\n", volumen);
            return 0;
         }
```

Así, en la expresión correspondiente a línea /*01*/ del Código 9.4., la llamada a la función cuadrado(radio) es equivalente a haber puesto una variable u otra expresión cualquiera de tipo **double**.

Si la función es de tipo **void**, entonces esa función no tiene valor de retorno, y por eso mismo no puede formar parte de ninguna expresión. Cuando se declara una función de tipo **void** ésta sólo puede ser invocada en una sentencia para ella sola: no puede formar parte de una expresión porque su tipo de dato (**void**) no tiene dominio de valores y, por tanto, no representa ningún valor posible.

Ejemplo 9.2. LOCALGLOBAL. Ámbito y vida de las variables

En el ejemplo anterior se ha mencionado que las variables que se declaran dentro de una función son de ámbito local. ¿Qué significa exactamente esto? Todas las variables que creamos dentro de un programa tienen un ámbito donde su uso es válido, fuera de este ámbito la variable no puede ser usada por ninguna función: no será reconocida y el compilador dará error en tiempo de compilación. Por tanto, podemos definir como ***ámbito de una variable*** al bloque de sentencias dentro de un programa en las que la variable es visible y está disponible. Asimismo, se puede definir el ***tiempo de vida de una variable*** como el tiempo que está disponible para poder acceder y manipular su contenido.

Hasta ahora hemos trabajado con variable de **ámbito local**, pero además de variables locales también es posible definir variables de **ámbito global**. Por tanto, podemos clasificar una variable según su ámbito: **local** o **global**.

- Una *variable local* es aquella cuyo ámbito se restringe a la función o al bloque de código donde se ha declarado y se dice entonces que la variable es local a esa función o al correspondiente bloque de código. Esto implica que esa variable sólo va a poder ser manipulada en dicho espacio

de código y no se podrá hacer referencia fuera del mismo. *Las variables locales existen sólo dentro de un bloque de código el cual se encuentra limitado por llaves*, de tal manera que la variable existe sólo dentro de los límites de éstas. De esta forma, cualquier variable que se defina dentro de las llaves del cuerpo de una función se interpreta como una variable local a esa función.

- Una *variable global* es aquella que se define fuera del cuerpo de cualquier función, normalmente al principio del programa, después de la inclusión de los archivos de biblioteca (**#include**), de la definición de constantes o macros (**#define**) y antes de la definición de cualquier función. El ámbito de una variable global está formado por todas las funciones que componen el programa, cualquier función puede acceder a dichas variables para leer y escribir en ellas. Por convenio, y para facilitar la inteligibilidad de los programas, los nombres de las variables globales los definiremos con mayúsculas. No se aficione a este tipo de variables.

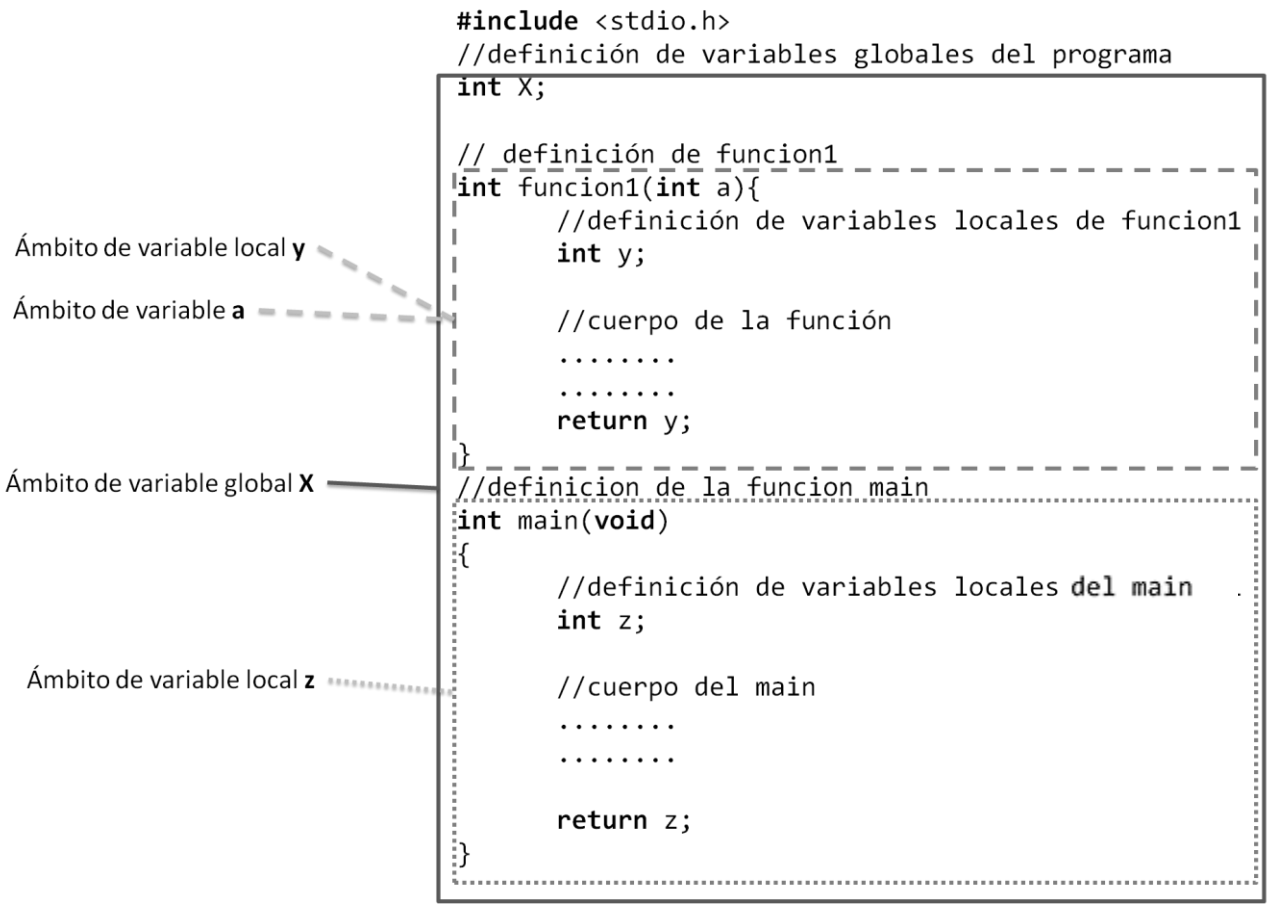


Figura 9.5. Ámbito y vida de variables locales y globales.

El ***correcto uso de*** **variables globales no es trivial** a pesar de que aparentemente nos pueda parecer muy útil, e incluso puede resultar desaconsejable debido las siguientes razones**:**

- Legibilidad menor.
- Atenta contra uno de los principios de la programación: la modularidad.

- El uso indiscriminado de variables globales produce efectos colaterales. Esto sucede cuando existe una alteración no deseada del contenido de una variable global dentro de una función, bien por invocación, bien por olvidar definir en la función una variable local o un parámetro formal con ese nombre. La detección y corrección de dichos errores puede ser muy ardua.

La Figura 9.5. resume estos conceptos de ámbito y de vida de variables mediante un sencillo ejemplo ilustrativo. Para finalizar, se muestra en Código 9.5. un sencillo programa que puede ayudar a comprender los conceptos de ámbito y vida de variables.

Código 9.5. Implementación correspondiente al Ejemplo 9.2.

```
            #include <stdio.h>

 /*01*/     double PI = 3.141592; //PI es una variable global

 /*02*/     double cuadrado(double); //declaracion de la función

            int main (void)
            {
                  double a; //a es una variable local en la función principal
                  a = 2.0;
 /*03*/           printf ("El cuadrado de a es %lf\n", cuadrado(a));
                  printf ("Con seis decimales, PI = %lf\n", PI);

                  PI = 3.1416; //actualizo el valor de PI
                  printf ("Con cuatro decimales PI = %lf\n", PI);
                  return 0;
            }

            double cuadrado(double n)
            {
                  return n * n;
            }
```

Tal y como se puede apreciar en las líneas `/*01*/` y `/*02*/` de Código 9.5., la variable `PI` es de ámbito global al estar declarada al principio del código del programa (fuera de todas las llaves correspondientes a las funciones `main` y `cuadrado`). En este programa, el valor de `PI` es inicialmente establecido a 3.141592 fuera de ambas funciones, mientras que en la quinta sentencia de la función principal cambia posteriormente su valor a 3.1416. Esto es posible hacerlo gracias a que, al ser `PI` una variable global, es accesible desde cualquier ámbito de código del programa.

Por otro lado, las variables `a` y `n` son de ámbito local para las funciones `main()` y `cuadrado()`, respectivamente. Por tanto, `a` no está directamente disponible para la función `cuadrado()` y `n` no es visible para la función `main()`. Sin embargo, en la línea `/*03*/`de Código 9.5. se hace una llamada a la función `cuadrado()` pasando el **valor** de la variable local a como parámetro de entrada. Este **valor** (en concreto, 2.0) queda asignado al parámetro correspondiente de la función `cuadrado()`: está por tanto disponible en la variable `n` (parámetro de entrada) de ámbito local a la

función `cuadrado()`: el valor que en el ámbito de la función `main()` está almacenado en la variable `a` se copia ahora en la variable `n`, local a la función `cuadrado()`.

Una variable permanece "viva" mientras no se haya ejecutado la última sentencia del bloque en el que ha sido declarada. Puede ocurrir que una variable permanezca viva pero no podamos acceder a ella porque hayamos salido de su ámbito. Esto ocurre, por ejemplo, con la variable `a` de la función `main()`. Cuando en esta función principal invocamos a la función `cuadrado()` abandonamos el ámbito de la función `main()` y por tanto no podemos ahora acceder a sus variables. Pero estas variables no han "muerto": esto quiere decir que su espacio de memoria no puede ser utilizado por otra variable en el nuevo ámbito ni por ningún otro programa en ejecución: el sistema operativo se encarga de ello. No podemos acceder a su valor de forma directa; pero cuando regresemos al ámbito de la función `main()` —cosa, ésta, que ocurrirá en cuanto la función `cuadrado()` ejecute su **`return`**— allá estará la variable `a` dispuesta para ser usada como antes de haber abandonado el ámbito de `main()`. Mientras tanto, al invocar a la función `cuadrado()` se ha creado la variable local `n` (parámetro de la función): pero en cuanto hemos ejecutado la sentencia **`return`** no sólo hemos abandonado el ámbito de la función `cuadrado()`, sino que además todas sus variables (variable `n`) han "muerto": el espacio de memoria que ocupaban vuelve a estar disponible para quien quiera necesitarlo. Desde luego, podríamos invocar a la función `cuadrado()` otra vez u otras muchas veces: pero nada nos garantiza que cuando lo hiciéramos íbamos a tener la variable `n` ubicada en el mismo espacio que tenía antes, en su anterior "vida".

Ejemplo 9.3. FACTORIAL. Concepto básico de recursividad

Una **función recursiva** es *aquella función que se llama a sí misma*. La recursividad es una propiedad de muchos sistemas de la naturaleza: el mismo desarrollo orgánico de la vida es un proceso fuertemente recursivo; por eso, con frecuencia, a la hora de buscar soluciones a problemas planteados, o cuando se pretende crear un programa que describa un comportamiento de la realidad creada, es fácil acabar en una definición de tipo recursivo. En general:

(1) la recursividad es una técnica que se puede utilizar en lugar de las estructuras iterativas;

(2) habitualmente los programas recursivos resultan más elegantes y simples que los escritos de manera iterativa; de todas formas también suelen exigir mayor cantidad de recursos.

Un ejemplo clásico y sencillo de recursividad es la **función factorial**. Si tenemos en cuenta que es posible definir el concepto de factorial de forma recursiva ($n! = n \cdot (n-1)!$) entonces también deberíamos poder hacer uso de esa nueva definición a la hora de implementar una función (**`long`** `fact(`**`short`** `n)`) de forma recursiva. Y así, podremos calcular el factorial de un valor `n` con una función que calcule el producto de `n` por el resultado de calcular el factorial de `n - 1`: es decir, que el cuerpo de nuestra función podría tener una sentencia muy parecida a: **`return`** `n * fact(n-1);`. Esta forma de proponer una definición exige encontrar un final a esta referencia a sí misma. Un caso límite en el que no sea necesario acudir a la propia definición: de lo contrario habríamos caído en un proceso infinito. Por ejemplo, en nuestro ejemplo del cálculo del factorial ese caso límite es el del cero o del uno: $0! = 1! = 1$. En Código 9.6. y Código 9.7. se muestran dos implementaciones, una iterativa y otra recursiva, del factorial de un número entero.

Código 9.6. Implementación ITERATIVA del Ejemplo 9.3.

```
long fact(short n)
{
	long i, prod = 1;
	for (i = 1; i <= n; i++){
		prod *= i;
	}
	return prod;
}
```

Código 9.7. Implementación RECURSIVA del Ejemplo 9.3.

```
long fact(short n)
{
	if (n == 0)
	{
		return 1;
	}
	else
	{
		return n * fact(n - 1);
	}
}
```

Como se ve en la definición recursiva (Código 9.7.), la función `fact()` devuelve el valor 1 si el valor del parámetro `n` de entrada es 0 ó 1; y devuelve el producto de `n` por el factorial de `n – 1` si el valor del parámetro de entrada es distinto a 0 ó 1. Desde luego la función mostrada en Código 9.7. se implementa en menos líneas con el operador ternario interrogante, dos puntos. Puede verlo en Código 9.7.bis.: acostúmbrese a esta forma de mostrar el código, típica en las definiciones recursivas.

Código 9.7.bis. Implementación RECURSIVA, más breve, del Ejemplo 9.3.

```
long fact(short n)
{
	return n ? n * fact(n - 1) : 1;
}
```

9.2. PROYECTOS PROPUESTOS

Proyecto 9.1. FUNCION_ECUACION

El primer paso es crear un proyecto denominado **FUNCION_ECUACION**. En este proyecto debe programar una función denominada `solveEq1()` que permita resolver una ecuación de primer grado, del tipo `a·x + b = 0` dados los parámetros a y b que definen dicha ecuación. El prototipo de esta función es el siguiente:

Código 9.8. Prototipo para la función a implementar en el Proyecto 9.1.

```
double solveEq1(double a, double b);
```

A modo de ejemplo, se dispone del siguiente código para probar el correcto funcionamiento de la función implementada:

Código 9.9. Implementación para la función principal del Proyecto 9.1.

```
#include <stdio.h>

double solveEq1(double a, double b);

int main(void)
{
	double a, b, x;
	printf("ax+b=0\n\n");
	printf("Introduzca el valor para a....");	scanf("%lf", &a);
	printf("Introduzca el valor para b....");	scanf("%lf", &b);
	if(a) {
		x = solveEq1(a, b);
		printf("Resultado de %.2lf x + %.2lf = 0 --> %.2lf", a, b, x);
	}
	else
		printf("NO hay ecuación...\n\n");

	return 0;
}
```

Puede comprobar que la implementación de la función es correcta considerando que si `a = 5` y si `b = 10`, entonces se obtiene como resultado el valor -2. Habría que decidir quién es el encargado de verificar que el valor del primer coeficiente NO es cero. Por ser éste el primer ejercicio de funciones vamos a dejarlo fácil: la función no debe preocuparse por ello: será tarea de la función principal.

Proyecto 9.2. FUNCION_PRIMOS

Cree ahora un proyecto denominado **FUNCION_PRIMOS**.

➔ *P9.2. Ejercicio 1.*

Primero, debe desarrollar una función denominada `esPrimo()` que determine si un número entero (parámetro de entrada) es o no es número primo. Esta función debe devolver un 1 (valor verdadero) si el número recibido como parámetro es efectivamente primo y un 0 (valor falso) en caso contrario. El prototipo de esta función queda recogido en Código 9.10.

Código 9.10. Prototipo para la función a implementar en el P9.2. Ejercicio 1.

```
short esPrimo(short n);
```

A modo de ejemplo, se dispone de una posible función principal (cfr. Código 9.11.) para probar el correcto funcionamiento de la función implementada.

Código 9.11. Implementación para la función principal del Proyecto 9.2.

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

short esPrimo(short n);

int main(void)
{
	short numero;
	printf("Introduzca un numero entero....");
	scanf("%hd", &numero);

	if(esPrimo(numero))
	{
		printf("SI es PRIMO");
	}
	else
	{
		printf("NO es PRIMO");
	}
	return 0;
}
```

➔ *P9.2. Ejercicio 2.*

A partir del programa anterior, se pide una función `nPrimos()` que calcule el número de primos que hay en un intervalo definido por los dos números positivos (`a` y `b`) que se pasan como parámetros a la función.

Código 9.12. Prototipo para la función a implementar en el P9.2. Ejercicio 2.

```
short nPrimos(short a, short b);
```

Esta función debe hacer uso de la función `esPrimo()` que determina si un número es primo o no. Además, sólo calculará el número de primos si `b` es mayor o igual que `a`, en caso contrario, la función devolverá un 0.

Para comprobar que la implementación que usted propone es correcta considere por ejemplo que entre los valores `a = 20` y `b = 50` hay 7 enteros primos (que son: 23, 29, 31, 37, 41, 43 y 47)

Proyecto 9.3. FUNCIONES_TARTAGLIA

En este ejercicio creamos un proyecto denominado **FUNCIONES_TARTAGLIA**.

El ***Triángulo de Tartaglia***, también denominado de Pascal, es una colección infinita de números dispuestos en forma triangular que se obtienen de una manera muy sencilla. En la parte superior del triángulo hay un número 1, en la segunda fila hay dos 1, y las demás filas empiezan y terminan siempre con 1. Además, cada número intermedio se obtiene sumando los dos que se encuentran justo encima. Existe un procedimiento para obtener este triángulo mediante números combinatorios. La Figura 9.6. muestra dicho procedimiento.

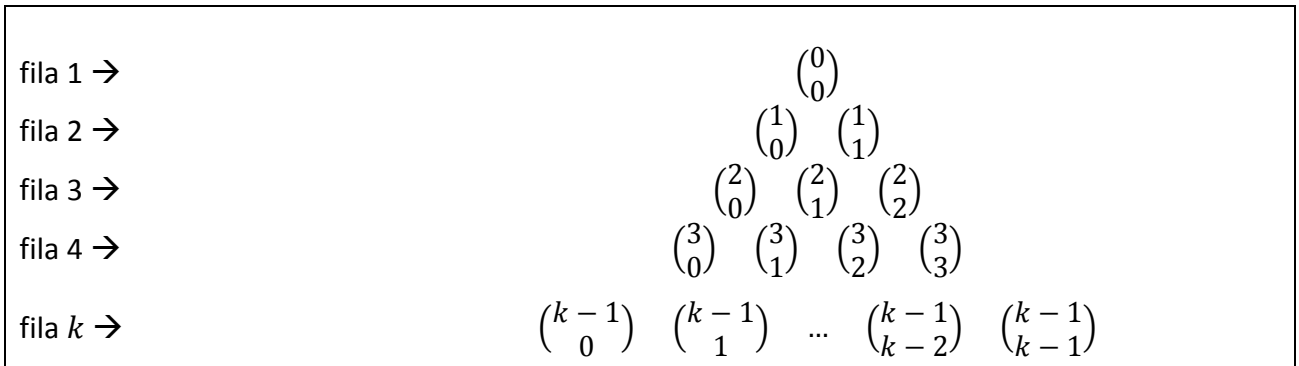


Figura 9.6. Representación del Triángulo de Tartaglia.

donde, como es conocido, la expresión de los números combinatorios es:

$$\binom{m}{k} = \frac{m!}{k! \times (m-k)!}$$

➔ *P9.3. Ejercicio 1.*

A partir de la implementación recursiva de la función `factorial()` del Código 9.7. (ó Código 9.7.bis), debemos desarrollar una función denominada `binomio()` que permita calcular la anterior expresión teniendo como parámetros de entrada los valores enteros de `m` y `k`. El prototipo de esta función es el siguiente:

Código 9.13. Prototipo para la función a implementar en el P9.3. Ejercicio 1.

```
unsigned long binomio(unsigned short m, unsigned short k);
```

Considere que, para la implementación de esta función, quizá le sea útil hacer uso de otra función que calcule el factorial de un entero recibido como parámetro.

➔ *P9.3. Ejercicio 2.*

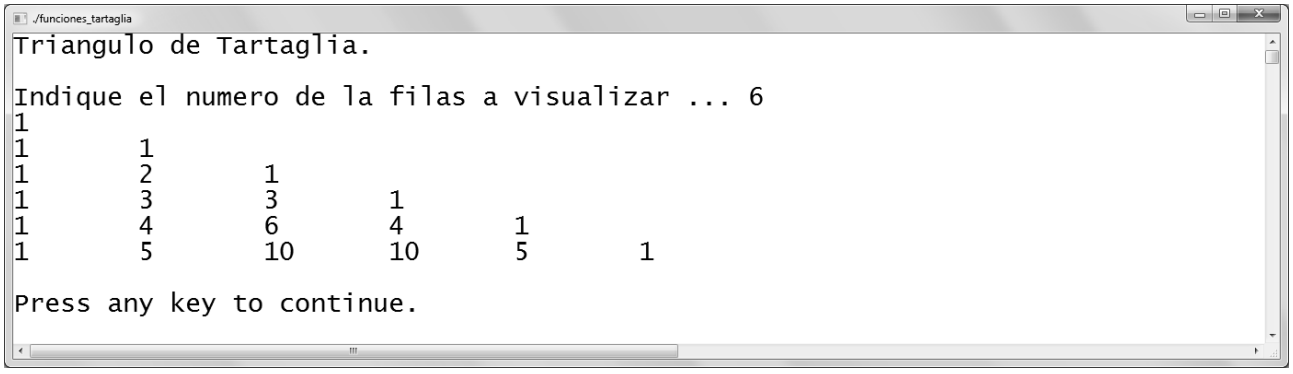
Implemente una nueva función denominada `tartaglia()` que imprima por pantalla la fila i-esima del triangulo de Tartaglia, teniendo como parámetro de entrada el valor del entero i. El prototipo de esta función es el siguiente:

Código 9.14. Prototipo para la función a implementar en el P9.3. Ejercicio 2.

```
void tartaglia(unsigned short i);
```

➔ *P9.3. Ejercicio 3.*

Desarrolle un programa principal que haga uso de las funciones anteriores de tal forma que: (1) solicite el número de filas que se desean imprimir (por ejemplo n filas); y (2) imprima el triángulo de Tartaglia de las n filas. A modo de ejemplo, a continuación se muestra el resultado que debería mostrar por pantalla en el caso de un triángulo de Tartaglia con 6 filas.



Proyecto 9.4. INTERESCOMPUESTO

Simplificando su concepto financiero, el *interés compuesto* se puede definir como la reinversión de los beneficios. Es decir, obtener intereses sobre los intereses que ya genera una inversión. El interés compuesto se basa en una fórmula matemática que calcula los intereses sobre la base inicial más los intereses acumulados en periodos anteriores.

Supongamos que C_0 denota el capital inicial; que I denota la tasa de interés por período de tiempo; y que N denota el número de períodos durante el plazo que dura la inversión. De esta forma, el capital al finalizar la inversión es igual a

$$C_N = C_0 \cdot (1 + I)^N$$

Una vez que se comprende el concepto de interés compuesto, creamos un proyecto denominado **INTERESCOMPUESTO**.

➔ *P9.4. Ejercicio 1.*

Desarrolle una **función iterativa** que permita calcular el capital obtenido en una inversión basada en el interés compuesto. El prototipo de dicha función es

Código 9.15. Prototipo para la función a implementar en el P9.4. Ejercicio 1.

```
double interesCompuesto_i(double C, double I, short N);
```

A la hora de probar el correcto funcionamiento de nuestra función, haremos uso de este código

Código 9.16. Implementación para la función principal del P9.4. Ejercicio 1.

```
#include <stdio.h>

double interesCompuesto_i(double C, double I, short N);

int main(void)    {
      double cantidad, cantidadfinal;
      short anyos;
      short interes;

      printf("BENEFICIO DE INVERSION\n");
      printf("Introduzca su cantidad inicial....\n");
      scanf(" %lf", &cantidad);
      printf("Introduzca el tipo de interes en tanto por ciento....\n");
      scanf(" %hd", &interes);
```

Código 9.16. (Cont.)

```
        printf("Introduzca el numero de anyos....\n");
        scanf(" %hd", &anyos);

        cantidadfinal = interesCompuesto_i(cantidad, interes/100.0, anyos);
        printf("Cantidad Final: %.2lf \n",cantidadfinal);

        return 0;
}
```

➔ *P9.4. Ejercicio 2.*

A partir del programa anterior, implementar una función **recursiva** para calcular el capital obtenido en una inversión basada en el interés compuesto. En este caso, el prototipo de la función es

Código 9.17. Prototipo para la función a implementar en el P9.4. Ejercicio 2.

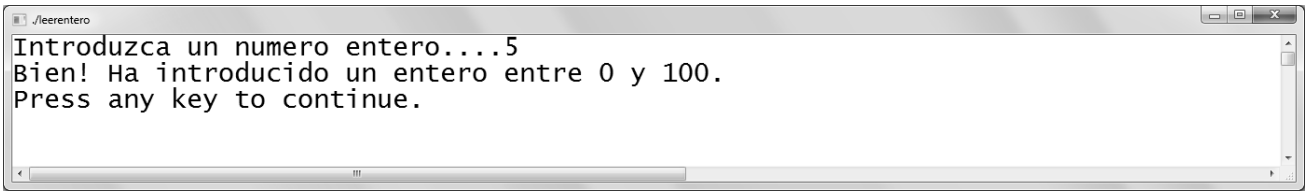
```
double interesCompuesto_r(double C, double I, short N);
```

Con objeto de que pueda comprobar que la implementación de ambas funciones es correcta, puede considerar el siguiente caso: una cantidad inicial de 1000 €, un interés anual igual a 4% y un deposito a 5 años; con estos valores iniciales, la cantidad final deberá ser 1216,25.

Proyecto 9.5. LEERENTERO [EJERCICIO ADICIONAL]

En prácticas anteriores hemos leído un entero desde teclado utilizando la función `scanf` a través de sentencias del tipo `scanf(" %d", &numero)`. Sin embargo, la función `scanf()` provoca muchos problemas cuando el usuario no introduce por teclado un número (es decir, el usuario introduce letras, signos de puntuación, espacios etc. y seguidamente pulsa Intro).

A modo de ejemplo, escriba y compile el programa recogido en Código 9.18. Si lo ejecuta e introduce un 5 y pulsa Intro, tendrá un resultado (ver figura siguiente) que es el esperado:



Código 9.18.

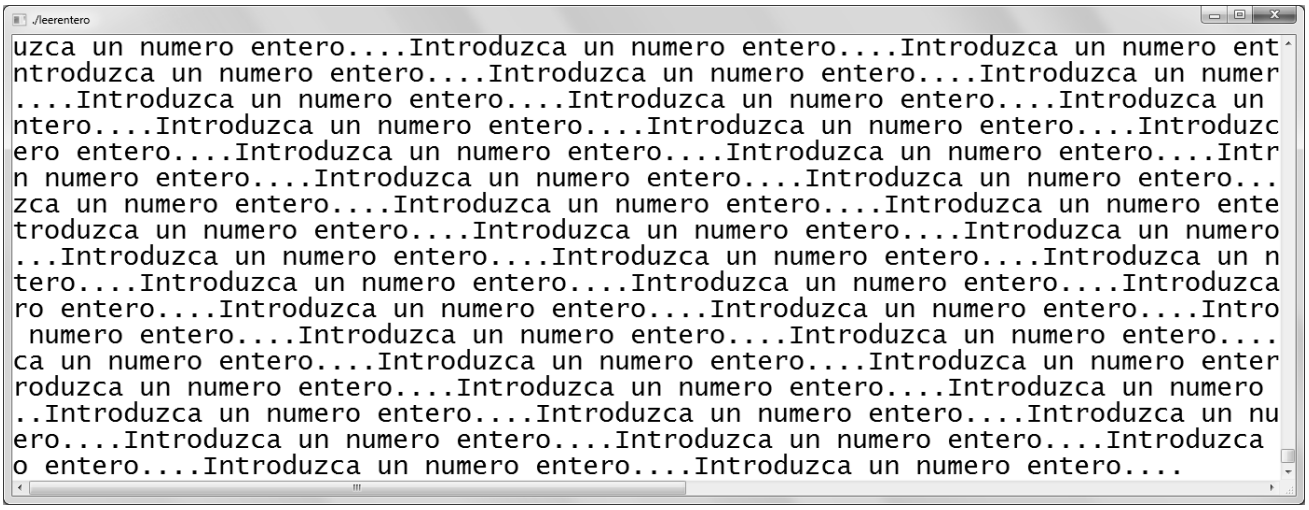
```
#include <stdio.h>

int main(void) {
	int numero;
	do{
		printf("Introduzca un numero entero....");
		scanf(" %d", &numero);
	}while(numero > 100 || numero < 0);

	printf("Bien! Ha introducido un entero entre 0 y 100.");

	return 0;
}
```

Ahora volvemos a ejecutar el programa e introducimos una letra cualquiera y pulsar Intro. En este caso, el programa "se vuelve loco" y se obtiene (por ejemplo) el siguiente resultado por pantalla:



Al no haber introducir un dato del tipo esperado (un entero) el funcionamiento de la función `scanf()` es erróneo y se ejecuta el bucle de manera infinita sin poder salir del mismo. Deberemos cerrar directamente el programa.

Para solucionar este problema, vamos a desarrollar una nueva función que permita leer sucesivamente los distintos caracteres introducidos por teclado omitiendo todos aquellos que no sean dígitos y los almacene en una cadena. Al introducir un carácter sólo deben visualizarse si son dígitos. Es decir, si introduzco la letra `'t'` no se debe mostrar por pantalla. Los dígitos son almacenados en una cadena de caracteres que debe ser convertida a entero. El prototipo de esta función queda recogido en Código 9.19.

Como ve, esta función no requiere ningún parámetro de entrada y devuelve un entero. Para la definición de esta función, se recomienda hacer uso de `getch()` de la biblioteca `<conio.h>` en el

caso de desarrollar la aplicación para entornos basados en sistemas operativos de Windows. También puede ser útil la función `isdigit()` de la biblioteca `<string.h>`. Así, por ejemplo, si el usuario introduce por teclado `"6hy789-3"` sólo se debe visualizar por pantalla `"67893"` y la función desarrollada deberá devolver el entero 67893.

Código 9.19. Prototipo para la función a implementar en el Proyecto 9.5.

```
int readInteger(void);
```

SOLUCIONES A LOS EJERCICIOS DE LOS PROYECTOS PROPUESTOS

Código 9.20. Solución para Proyecto 9.1.

```
#include <stdio.h>

double solveEq1(double a, double b);

int main(void) {
	double a, b, x;

	printf("ax+b=0\n\n");
	printf("Introduzca el valor para a....");
	scanf(" %lf", &a);
	printf("Introduzca el valor para b....");
	scanf(" %lf", &b);

	if(a) {
		x = solveEq1(a, b);
		printf("Resultado de %.2lf x + %.2lf = 0 --> %.2lf", a, b, x);
	}
	else
		printf("No hay ecuación: coeficiente a igual a cero!");
	return 0;
}

double solveEq1(double a, double b){
	return -b/a;
}
```

Código 9.21. Solución para P9.2.Ejercicio 1.

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

short esPrimo(short n);

int main(void) {
	short numero1, numero2;

	printf("Introduzca un numero entero....");
	scanf(" %hd", &numero1);

	printf("%s es PRIMO\n" , esPrimo(numero1) ? "SI" : "NO");

	return 0;
}
```

Código 9.21. (Cont.)

```
//short esPrimo(short n);
//devuelve 1 si n es primo; devuelve 0 si n no es primo
short esPrimo(short n){
	short i;

	for(i = 2 ; i <= sqrt(n) ; i++){
		if(n % i == 0) {
			return 0;
		}
	}
	return 1;
}
```

Código 9.22. Solución para P9.2.Ejercicio 2.

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

short esPrimo(short n);
short nPrimos(short a, short b);

int main(void) {
	short numero1, numero2;
	printf("Introduzca un numero entero....");
	scanf(" %hd", &numero1);
	printf("\nIntroduzca un segundo numero entero....");
	scanf(" %hd", &numero2);
	printf("\nEl numero de primos entre ");
	printf("%hd y %hd es %hd",
			numero1, numero2, nPrimos(numero1 , numero2));
	return 0;
}

//short esPrimo(short n);
//devuelve 1 si n es primo;devuelve 0 si n no es primo
short esPrimo(short n){
	short i;

	for(i = 2 ; i <= sqrt(n) ; i++){
		if(n % i == 0) {
			return 0;
		}
	}
	return 1;
}
```

Código 9.22. (Cont.)

```
//short nPrimos(short a,short b);
//devuelve el numero de primos entre a y b
short nPrimos(short a, short b){
      short i, contador = 0;

      for(i = a; i <= b; i++) {
            if(esPrimo(i)) {
                  contador++;
            }
      }
      return contador;
}
```

Código 9.23. Solución para Proyecto 9.3.

```
#include <stdio.h>

void              tartaglia   (unsigned short);
unsigned long     binomio     (unsigned short, unsigned short);
unsigned long     factorial   (unsigned short);

int main(void)
{
      unsigned short fila, i;

      printf("Triangulo de Tartaglia.\n\n");
      printf("Indique el numero de la filas a visualizar ... ");
      scanf(" %hu", &fila);

      for(i = 1; i <= fila; i++){
            tartaglia(i);
            printf("\n");
      }
      return 0;
}

void tartaglia(unsigned short f)
{
      unsigned short i;

      for(i = 0 ; i < f ; i++)
      {
            printf("%lu\t", binomio(f - 1 , i ));
      }
      return;
}
```

Código 9.23. (Cont.)

```
unsigned long binomio(unsigned short m, unsigned short k)
{
      return factorial(m) / (factorial(k) * factorial(m - k));
}

unsigned long factorial(unsigned short n)
{
      return n ? n * factorial(n - 1) : 1 ;
}
```

Código 9.24. Solución para Proyecto 9.4.

```
#include <stdio.h>

double interesCompuesto_i(double C, double I, short N);
double interesCompuesto_r(double C, double I, short N);

int main(void)
{
      double cantidad, cantidadfinal_i, cantidadfinal_r;
      short anyos;
      short interes;

      printf("BENEFICIO DE INVERSION\n");
      printf("Introduzca su cantidad inicial....\n");
      scanf(" %lf", &cantidad);

      printf("Introduzca el tipo de interes en tanto por ciento....\n");
      scanf(" %hd", &interes);
      printf("Introduzca el numero de anyos....\n");
      scanf(" %hd", &anyos);

      cantidadfinal_i =
            interesCompuesto_i(cantidad, interes/ 100.0, anyos);

      printf("\nImplementacion iterativa\n");
      printf("\tCantidad Final: %.2lf \n", cantidadfinal_i);

      cantidadfinal_r =
            interesCompuesto_r(cantidad, interes / 100.0, anyos);

      printf("\nImplementacion recursiva\n");
      printf("\tCantidad Final: %.2lf \n", cantidadfinal_r);

      return 0;
}
```

Código 9.24. (Cont.)

```
//implementacion iterativa del interes compuesto
double interesCompuesto_i(double C, double I, short N) {
	short i = N;
	double Cf = C;
	while(i > 0) {
		Cf = Cf * (1 + I);
		i--;
	}
	return Cf;
}

//implementacion recursiva del interes compuesto
double interesCompuesto_r(double C, double I, short N) {
	return N > 0 ? interesCompuesto_r(C , I , N - 1) * (1 + I) : C;
}
```

Código 9.25. Solución para Proyecto 9.5.

```
#include <stdio.h>
#include <ctype.h>
#include <math.h>
#include <conio.h>

int readInteger(void);

int main(void)
{
	int numero;

	do{
		printf("Introduzca un numero entero....");
		numero = readInteger();
	} while(numero > 100 || numero < 0);

	printf("Bien! Ha introducido un entero entre 0 y 100.");

	return 0;
}


int readInteger(void)
{
	char c, numero[50];
	short i = 0;
```

Código 9.25. (Cont.)

```
        do
        {
              c = getch();
              if(isdigit(c))
              {
                    printf("%c", c);
                    numero[i++] = c;
              }
        } while(c != 13 && c != 10);
        numero[i] = 0;
        return atoi(numero);
  }
```

Funciones (II): Parámetros por referencia

10

10.1. Ejemplos sencillos resueltos

Ejemplo 10.1. *PUNTEROSBASICO.*

Ejemplo 10.2. *PUNTEROSARRAYS.*

Ejemplo 10.3. *PUNTEROSFUNCIONES.*

10.2. Proyectos propuestos

Proyecto 10.1. *PUNTEROS_INTERCAMBIO.*

Proyecto 10.2. *PUNTEROS_BASICO_VECTORES*.

Proyecto 10.3. *PUNTEROS_BASICO_CADENAS*.

Proyecto 10.4. *PUNTEROS_OPERACIONES_VECTORES*.

Proyecto 10.5. *PUNTEROS_DETERMINANTE*.

Proyecto 10.6. *PUNTEROS_DNI*

Proyecto 10.7. *PUNTEROS_ORDENACION*. *Ejercicio adicional.*

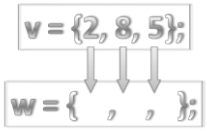
Proyecto 10.8. *PUNTEROS_RIEL*. *Ejercicio adicional.*

A lo largo de estas prácticas se ha trabajado con variables que almacenan valores según uno u otros tipos de dato preestablecidos, como pueden ser **`int`**, **`char`**, **`float`** o **`double`**. Según el tipo de dato, la variable correspondiente ocupa en memoria un número concreto y determinado de bytes: 1 byte las variables de tipo **`char`**, dos las enteras **`short`**, cuatro las enteras **`long`** y las **`float`**, y ocho las **`double`** o las **`long long`**.

Además de los tipos de datos analizados hasta ahora, existe un tipo muy especial: el tipo de dato *puntero*. Una variable de tipo *puntero* codifica direcciones de memoria. Habitualmente, a una variable puntero le asignaremos la dirección de memoria de otra variable de nuestro programa.

El objetivo de esta práctica es que el alumno aprenda a manejar punteros y a crear e invocar a funciones que tienen punteros entre algunos de sus parámetros. Antes de comenzar con los ejercicios de esta práctica, el alumno puede revisar las nociones y fundamentos de los punteros a través de una serie de ejemplos sencillos. A continuación, el alumno deberá ser capaz de un total de seis ejercicios. Los ejercicios propuestos son los siguientes:

1. Implementar una función que intercambie los valores de dos variables haciendo uso de llamadas por referencia.



2. Desarrollar una función que copie todos los elementos de un vector de enteros en otro haciendo uso de punteros.

3. Implementar una función que copie todos los caracteres de una cadena en otra en sentido inverso.

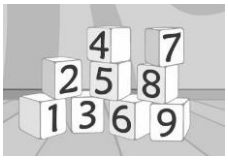
4. Programar funciones para generar aleatoriamente un vector de números enteros y calcular los estadísticos principales de la serie de números almacenados en dicho vector.

5. Implementar una función que calcule el valor del determinante de una matriz cuadrada de dimensiones 3x3.

$$\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & i \end{vmatrix}$$

6. Desarrollar una función que determine si la letra del DNI es correcta.

7. Desarrollar una función que reciba un array de enteros y lo deje ordenado de menor a mayor, mediante el algoritmo de ordenación por selección.



8. Crear una función que reciba una cadena de texto y cree otra formada por el desdoblamiento y concatenación de las subcadenas de caracteres de posición par y de posición impar.

10.1. EJEMPLOS SENCILLOS RESUELTOS

Antes de comenzar con los problemas que componen esta última práctica, será conveniente realizar una serie de ejemplos iniciales que permita al alumno introducirse en el uso de punteros.

Ejemplo 10.1. PUNTEROSBASICO.

Un *puntero* es un tipo de dato cuyo dominio es el de todas las direcciones de memoria del ordenador. En los sistemas de arquitectura de 32 bits estas direcciones se codifican con 4 bytes.

> ***NOTA**: Si nuestro ordenador (suponemos una arquitectura de 32 bits) utiliza 4 bytes para dar nombre a una posición cualquiera de memoria (a cualquiera de los bytes de la memoria)… ¿A cuántos bytes podremos dar nombre si tenemos 32 bits para construir nombres distintos?: pues a 4.294.967.296 (2 elevado a 32)*
>
> *Que podamos dar nombre a 2^{32} bytes quiere decir que podemos dárselo a 4×2^{30} bytes, es decir, por tanto, a 4 GigaBytes.*
>
> *Y así, podemos concluir que con posiciones de memoria direccionadas con códigos de 32 bits se pueden tener memorias de hasta 4 Gigas. Si su ordenador tiene más RAM que ésa, puede estar seguro que o bien no la ve entera, o bien su arquitectura no es de 32 bits.*

Una variable tipo puntero contiene la dirección de memoria de otra variable y es posible declarar punteros a variables de cualquier tipo. Así, por ejemplo, en el Código 10.1. se han declarado un puntero a **`int`** (la variable `p_i`), un puntero a **`float`** (la variable `p_f`) y dos variables de tipo **`int`** y **`float`** (`i` y `f` respectivamente). Como se puede comprobar la declaración de un puntero implica hacer uso del carácter asterisco (*) precediendo al nombre de la variable en cuestión.

> ***NOTA**: El asterisco que en la declaración precede al nombre de la variable tipo puntero no forma parte del nombre, sino que se comporta más bien como modificador del tipo de dato.*

Para el correcto uso de punteros, existen una serie de operadores que es necesario revisar:

Operador dirección (&). Se aplica a cualquier variable y devuelve la dirección de memoria de esta variable. Es, pues, un operador monario. Este operador ya ha sido utilizado desde la primera práctica: se hace uso de él al leer datos desde teclado utilizando la función `scanf`: esta función necesita conocer de la dirección de memoria sobre la que deberá guardar ese valor introducido. Por eso, la función `scanf` recibe dos parámetros: en el primero, mediante una cadena de texto, conoce el tipo de dato del valor que se espera; con el segundo se indica a la función la posición que, en memoria, ocupa la variable donde la función deberá almacenar el valor introducido.

***NOTA**: Cuando a un puntero se le asigna la dirección de una variable, se suele decir que ese puntero "apunta" a esa variable.*

```
double x = 3.14, *p;
p = &x;
```

Diremos entonces que p vale la dirección de x o que p apunta a x.

Al crear las variable x y p, tenemos:

⟨x **double** R_x 3.14⟩.

⟨p **double*** R_p R_x⟩.

Es decir, x es una variable de tipo **double***, ubicada en* R_x *y de valor* 3.14*; y p es una variable de tipo puntero a* **double***, ubicada en* R_p *y de valor* R_x*. Así pues,* p *vale una dirección de memoria (*R_x*), porque es de tipo puntero. Y* p *tiene una dirección de memoria (*R_p*), porque es una variable y requiere por tanto ocupar un espacio en ella.*

Este cruce de doble significado de la noción de dirección en los punteros (tiene una dirección y vale una dirección) suele ocasionar quebraderos de cabeza en los inicio del manejo de los punteros. Son conceptos muy sencillos… pero eso no quiere decir que su manejo también lo sea.

Operador indirección (*). Se aplica solamente a los punteros. Al aplicar este operador a un puntero, se obtiene el contenido de la posición de memoria a la que el puntero apunta. Es un operador monario.

Código 10.1. Declaración de variables y de variables puntero.

```
//variables de tipo int y float
int i;
float f;

//punteros a variables de tipo int y float
int *p_i;
float *p_f;
```

En Código 10.2 se muestra de manera ilustrativa el uso de punteros y los correspondientes operadores.

Al ejecutar este código se mostrarán por pantalla los valores y las direcciones de memoria de las variables `i` y `f`. El código asigna a los punteros `p_i` y `p_f`, las direcciones de memoria de las variables `i` y `f`, a las que previamente se les ha asignado los valores `5` y `1.16` (ver línea `/*01*/` del Código 10.1.), respectivamente. Esto se ha conseguido a través de las sentencias correspondientes a las líneas `/*02*/` y `/*03*/` del Código 10.2.

Código 10.2. Implementación correspondiente al Ejemplo 10.1.

```
     #include <stdio.h>

     int main(void)
     {
           //variables de tipo int y float
           int i;
           float f;

     //punteros a variables de tipo int y float
           int *p_i;
           float *p_f;

           //asignacion de valores a las variables i y f
/*01*/     i = 5; f = 1.16;
     //asignacion de direcciones de memoria para los punteros
/*02*/     p_i = &i;
/*03*/     p_f = &f;

     //Visualizacion de valores y direcciones de memoria
           printf("i tiene almacenado un valor igual a");
           printf(" %d en la posicion de memoria %p\n", i, p_i);
           printf("f tiene almacenado un valor igual a");
           printf(" %.2f en la posicion de memoria %p\n", f, p_f);
           printf("\n\n");

           //Estas lineas son equivalentes a las anteriores
           printf("i tiene almacenado un valor igual a");
           printf(" %d en la posicion de memoria %p\n", *p_i, &i);
           printf("f tiene almacenado un valor igual a");
           printf(" %.2f en la posicion de memoria %p\n",*p_f, &f);

           return 0;
     }
```

Como se ha podido ver al ejecutar el código del ejemplo anterior, con `*p_i` y `*p_f` se obtienen los valores de `i` y `f`, respectivamente. La ejecución de este código ofrece una salida como ésta:

```
i tiene almacenado un valor igual a 5 en la posicion de memoria 0028FF14
f tiene almacenado un valor igual a 1.16 en la posicion de memoria 0028FF10

i tiene almacenado un valor igual a 5 en la posicion de memoria 0028FF14
f tiene almacenado un valor igual a 1.16 en la posicion de memoria 0028FF10
```

donde los valores mostrados como direcciones de memoria no serán necesariamente los mismos que usted pueda ver en su pantalla si ejecuta el mismo programa. De hecho, la decisión sobre dónde su van a ubicar las variables no es, en principio (hay excepciones), competencia del programador.

Es importante no dejar punteros “apuntando” de forma aleatoria a cualquier lugar de la memoria. Eso ocurre de hecho cuando no se le inicializa a un valor concreto. Es muy conveniente, mientras

que no se le de una valor concreto, dejar siempre los punteros inicializados al llamado puntero NULL (0), en cuyo caso se dice que el puntero es nulo (no apunta a nada). Este valor nulo nos permite identificar el puntero como no inicializado. Así, en un programa, siempre se puede condicionar la operación indirección al hecho de que el puntero sea diferente de NULL: no tendría sentido solicitar su valor indireccionado si no estuviera apuntando a alguna variable.

***NOTA**: ¿Cómo distinguir el operador AND (&) a nivel de bit del operador dirección?; ¿o cómo distinguir el operador producto (*) del operador indirección (*)?*

Los operadores dirección e indirección son monarios, mientras que los operadores AND a nivel de bit y producto son binarios.

No es lo mismo decir

```
x*=y;       // a x se le asigna el resultado de multiplicar x e y.
```

que decir (suponiendo ahora que y es de tipo puntero)

```
x=*y;       // a x se le asigna el valor de la variable apuntada por y.
```

Ni es lo mismo decir

```
x&=y;       // Es decir, a x se le asigna el resultado de x & y (and).
```

que decir (suponiendo ahora que x es de tipo puntero)

```
x=&y;       // Es decir, a x se le asigna la dirección de la variable y.
```

Ejemplo 10.2. PUNTEROSARRAYS.

Tal y como se ha indicado y recordado en las últimas prácticas, un array es una colección de variables del mismo tipo y colocadas en memoria de forma consecutiva. Esta última característica permite que, para tener bajo control toda la información de un array, sea suficiente con conocer (1) la posición del primero de sus elementos y (2) la cantidad de elementos del array. Ya verá que la forma en que el compilador logra conocer la ubicación del primer elemento del array es mediante un puntero. Y verá también que tener las distintas variables del array ubicadas de forma consecutiva no es ni baladí ni casual: es el hecho de que sus posiciones sean consecutivas lo que hace posible recorrer el array sin más información que un índice: porque al estar ubicados los elementos del array de forma consecutiva, se tiene que conocemos la posición relativa de cada uno respecto al primer elemento; y como conocemos la posición absoluta en memoria del primero, podemos ubicar también de forma absoluta al resto de elementos del array.

Supongamos que tenemos un array (con nombre v) de 5 variables de tipo **long** y un puntero a variable de tipo también **long** (con nombre pv) (cfr. Código 10.3.).

Con, por ejemplo, la sentencia marcada con `/*01*/`, podemos asignar al puntero pv la dirección de memoria de uno cualquiera de los elementos del array: en este caso, la dirección del elemento `v[2]`.

Para desplazarnos en memoria por las distintas posiciones correspondientes a los elementos del vector podemos hacer uso de los operadores suma y resta de enteros. Al sumar un entero (por ejemplo **`int`** `d = 1;`) a un puntero (por ejemplo, el puntero pv de tipo **long** que acabamos de

crear en las líneas de código anteriores) obtenemos la dirección de memoria de la variable ubicada d posiciones más a la derecha que la apuntada por el puntero al que se le ha sumado el entero d. Sumar o restar un entero d a un puntero da como resultado la dirección de una variable que está ubicada d posiciones a la derecha o a la izquierda del puntero al que se le ha sumado el entero. Y si asignamos al puntero pv la posición del elemento 2 del array pv, entonces pv + d (siendo d = 1) el resultado será la posición del elemento 3 del array (cfr. línea /*03*/ de Código 10.3.).

Código 10.3. Punteros y arrays

```
            //array de variables de tipo long
            long v[5] = {1, 2, 3, 4, 5};

            //puntero a variable de tipo long
            long *pv;

 /*01*/     pv = &v[2];
 /*02*/     pv = pv + 1;            // ahora pv apunta a v[3]
```

También están definidos, para los tipos de dato puntero, los operadores incremento (++) y decremento (--). En el caso de ser aplicados sobre una variable de tipo puntero, la correspondiente dirección de memoria será incrementada o decrementada en tantos bytes como bytes ocupe en memoria cualquier variable del tipo de dato que tiene asociado el puntero.

Código 10.4. Implementación correspondiente al Ejemplo 10.2.

```
      #include <stdio.h>
      int main(void)
      {
            //array de variables de tipo long
            long v[3] = {1, 2, 3};
            //puntero a variable de tipo long
            long *pv;

            pv = &v[0];
            printf("v[0]: Valor --> %ld; Memoria --> %p\n", v[0], pv);
            printf("v[1]: Valor --> %ld; Memoria --> %p\n", v[1], ++pv);
            printf("v[2]: Valor --> %ld; Memoria --> %p\n", v[2], ++pv);
            printf("\n\n");

            //Las siguientes lineas son equivalentes a las anteriores
            printf("v[2]: Valor --> %ld; Memoria --> %p\n",*pv, pv);
            printf("v[1]: Valor --> %ld; Memoria --> %p\n",*(--pv), pv);
            printf("v[0]: Valor --> %ld; Memoria --> %p\n",*(--pv), pv);

            return 0;
      }
```

Con objeto de comprender el funcionamiento de los operadores suma y resta de enteros a punteros, y por tanto también de los operadores incremento y decremento, podemos detenernos en el ejemplo propuesto en Código 10.4., donde, si ejecuta el código propuesto, obtendrá una salida semejante a la que aquí se muestra (probablemente con valores de las direcciones distintos).

```
v[0]: Valor --> 1; Memoria --> 0028FF00
v[0]: Valor --> 2; Memoria --> 0028FF04
v[0]: Valor --> 3; Memoria --> 0028FF08

v[0]: Valor --> 1; Memoria --> 0028FF00
v[0]: Valor --> 2; Memoria --> 0028FF04
v[0]: Valor --> 3; Memoria --> 0028FF08
```

***NOTA**: De hecho, cuando se declara un array, además de crearse tantas variables del tipo del array como indique la dimensión, también se crea una variable muy especial… ¡de tipo puntero! Al declarar:*

```
double vector[10];
```

se crean 10 variables de tipo **double** (`vector[0]`, `vector[1]`, … , `vector[9]`) *y el puntero a* **double** *de nombre* `vector`*. Este puntero tiene tres características singulares:*

(1) Su valor es el de la dirección del primer elemento del array: `vector` *vale* `vector[0]`*.*
(2) Es un puntero constante: no se le pueda cambiar su valor. No puede ser Lvalue en una asignación, ni puede aplicarse sobre él los operadores incremento o decremento.
(3) Si hace un programa que muestre por pantalla su valor, le mostrará la dirección de la primera variable del array (`vector[0]`*); y si hace un programa que muestre la dirección de este puntero (*`printf("Dirección del puntero vector: %p\n", &vector);`*)… ¡también le mostrará la dirección de la variable* `vector[0]`*!*

Quizá no es importante entender dónde está realmente guardada la dirección del primer elemento del array. Posiblemente no exista realmente esa variable puntero. Pero podemos hacer como si existiera… siempre y cuando no pretendamos hacer otra cosa más que leer su valor. Es muy útil.

Así, podemos hablar del valor de `vector[i]`*, y podemos hablar igualmente del valor en la posición* `vector + i`*; es decir,* `vector[i]` *es equivalente a* `*(vector + i)`*; y* `(vector + i)` *es equivalente a* `&vector[i]`*. Al manejar un array, pues, podemos trabajar con operatoria de punteros, y podemos trabajar también con operatoria de índices.*

Ejemplo 10.3. PUNTEROSFUNCIONES.

Hasta ahora, los ejemplos de funciones que se han implementado han hecho directamente uso de variables sencillas para los parámetros de entrada. Sin embargo, no hemos pasado matrices o vectores como parámetros de entrada y además en todas ellas sólo se ha devuelto a lo sumo un

único dato. Si deseamos pasar un array o una matriz como parámetro, o si necesitamos recoger más de un valor de retorno de la función invocada, se hace necesario, en C, el uso de punteros.

Supongamos que en la función principal nos encontramos con las siguientes líneas de código:

```
short x = 5;
long F = factorial(x);
```

y que la función factorial tiene la siguiente definición:

```
long factorial(short n)
{
      long f = 1;
      while(n) f *= n--;
      return f;
}
```

Lo que se produce en esta llamada es que el valor de la variable `x` —variable en el ámbito de la función `main`— copia su valor en la variable `n` —variable en el ámbito de la función `factorial()`—. En ese momento el programa abandona el ámbito de la variable `x` y crea un nuevo ámbito para la función `factorial` y sus variables `n` y `f`. Qué cosas haga la función `factorial()` con el valor pasado es cosa que a la función `main` no le importa ni le afecta en absoluto. Al final, la función `factorial()` devolverá un valor (será el valor 120) a la función `main` y las variables `f` y `n` dejarán de estar vivas: el ámbito creado para ellas queda eliminado.

Este procedimiento es conocido como **LLAMADA POR VALOR**: la función invocada recibe como entrada unos valores que le facilita la función que invoca. La función invocada no puede hacer nada con las variables de la función que llama porque trabajamos en ámbitos diferentes y exclusivos: la función invocada desconoce la existencia de las variables que le han pasado sus valores desde la función que invoca, y la función que invoca desconoce la existencia de las variables, parámetros de entrada, de la función invocada que recogen los valores que se le han pasado al llamarla.

Y supongamos ahora la función declarada y definida en Código 10.5., e invocada con el siguiente código desde de la función principal:

```
short x = 5 , y = 2; intercambiarValores(&x, &y);    // llamada desde main()
```

En este caso lo que hace la función `main` al llamar a la función `intercambiarValores()` no es pasarle los valores de las variables `x` e `y`: si eso hiciera, entonces todos los intercambios de valores que se realizan en la función llamada no afectarían a la función `main`. Lo que recibe la función invocada son las direcciones de esas dos variables; ahora las operaciones hechas sobre los contenidos apuntados por los punteros `a` y `b` de la función `intercambiarValores()` sí afectan a la

función `main`. Mientras se ejecutan la sentencias de la función `intercambiarValores()` estamos fuera del ámbito de las variables `x` e `y`; pero podemos acceder a ellas y manipularlas, porque la función invocada dispone de sus direcciones en memoria.

Código 10.5. Función `intercambiarValores()`

```
// Declaracion
      void intercambiarValores(short *a, short *b);

// Definicion
      void intercambiarValores(short *a, short *b)
      {
            *a ^= *b;
            *b ^= *a;
            *a ^= *b;
            return;
      }
// Otra posible definicion
void intercambiarValores(short *a, short *b)
{
      short aux = *a;
      *a = *b;
      *b = aux;
      return;
}
```

Este procedimiento es conocido como **LLAMADA POR REFERENCIA**: la función invocada recibe como entrada direcciones de unas variables que están fuera de su ámbito, pero que al disponer de su ubicación en memoria, podrá acceder a sus valores e incluso podrá cambiar esos valores. La función invocada puede acceder a esas variables fuera de su ámbito, y la función que invoca ha permitido esa posibilidad porque ha facilitado, como valores para los parámetros de entrada de la función invocada, las direcciones de sus variables sobre las que desea que la función invocada actúe.

El paso de variables por referencia también facilita la manipulación de arrays. Para pasar un array a una función invocada es suficiente con pasar como parámetro la dirección del primer elemento del vector. También suele ser necesario pasar, con otro u otros parámetros, la dimensión o dimensiones de este vector o matriz.

En Codigo 10.6. se define una función denominada `printArray()` que imprime por pantalla los elementos de un vector cuya dirección y dimensión recibe como parámetros de entrada. En este ejemplo concreto se ha utilizado la función `printArray()` para imprimir los valores diarios de humedad medidos durante una semana.

En la línea `/*01*/` la vemos cómo se le pasa a la función `printArray` la dirección del primer elemento del array (`&v[0]`). El código escrito es correcto, y si lo compila y ejecuta verá que, efectivamente, nada anda mal. Pero es más habitual, al pasar la dirección de un array, dar simplemente el nombre del array. Hemos dejado comentada, en esa línea `/*01*/` de Código 10.6.,

la forma habitual de hacer esa llamada: la sentencia `printArray(v, 7)`. Así pues, también así tenemos una llamada por referencia.

Código 10.6. Programa para imprimir un vector numérico a través de la función `printArray()`

```
     #include <stdio.h>

     void printArray(short *, short);

     int main(void)
     {
           short v[7]={68, 72, 73, 76, 78, 77, 75};
           printf("Los datos de humedad son:\n");
/*01*/     printArray(&v[0] , 7);       // printfArray(v , 7);
           return 0;
     }

     void printArray(short *a, short d) {
           short i;
           for(i = 0; i < d; i++){
                 printf("%hd\t", a[i]);
           }
           return;
     }
```

El ejemplo del Código 10.6. no es extrapolable a las matrices: su manejo tiene otra complejidad. En esta práctica, cuando se necesite pasar por referencia una matriz, lo haremos siempre declarando como parámetro no un puntero si una matriz de las dimensiones que reciba en otros parámetros.

Por ejemplo, si se desea implementar una función que reciba como parámetro (entre otros) una matriz, le daremos la siguiente declaración de prototipo (la definimos, por ejemplo, de tipo **double**):

```
double función(short filas, short columnas, long MTR[filas][columnas]);
```

Así, en la función podremos manejar la matriz `MTR` recibida por referencia como tercer parámetro. Es importante que los valores de las dimensiones de `filas` y `columnas` precedan a la declaración de la matriz `MTR`: las declaraciones de los parámetros se producen de izquierda a derecha, y si hubiéramos escrito la declaración en otro orden (primero la matriz `MTR` y luego sus dimensiones) el compilador daría un error, porque en el momento de la declaración de la matriz desconocería el significado de los nombres de las dos variables (`filas` y `columnas`) que aún no habían sido creadas.

10.2. PROYECTOS PROPUESTOS

Proyecto 10.1. PUNTEROS_INTERCAMBIO

***NOTA**: Este ejercicio ya ha sido resuelto en la primera parte de esta práctica, pero no lo ha podido comprobar. Ahora es el momento de verificar cómo realmente trabaja la función de intercambio de valores al pasar las referencias de las variables cuyos valores deben ser intercambiados.*

Creamos un proyecto denominado **PUNTEROS_INTERCAMBIO**. En el mismo, tenemos que desarrollar una función denominada `intercambio()` que permita intercambiar los valores de dos variables de tipo **`short`** haciendo uso de llamadas por referencia. El prototipo de esta función debe ser el indicado en Código 10.7.:

Código 10.7. Prototipo para la función a implementar en el Proyecto 10.1.

```
void intercambio(short *a, short *b);
```

donde se intercambien los valores de las variables a y b. Para probar el correcto funcionamiento de la función implementada se le sugiere la función `main` de Código 10.8.

Código 10.8. Implementación para la función principal del Proyecto 10.1.

```
#include <stdio.h>

void intercambio(short*, short*);

int main(void){
      short int a, b;

      printf("Introduzca el valor para a....");      scanf(" %hd", &a);
      printf("Introduzca el valor para b....");      scanf(" %hd", &b);

      printf("\nIntercambio de valores....\n");
      intercambio(&a, &b);

      printf("El valor de a es....%hd\n", a);
      printf("El valor de b es....%hd", b);

      return 0;
}
```

Proyecto 10.2. PUNTEROS_BASICO_VECTORES

Cree un nuevo proyecto denominado **PUNTEROS_BASICO_VECTORES** donde desarrollaremos una función denominada copyVector() que copie todos los elementos de un vector de enteros cortos a otro vector. El prototipo de esta función debe ser como el indicado en Código 10.9.

Código 10.9. Prototipo para la función a implementar en el Proyecto 10.2.

```
void copyVector(short *source, short *target, short d);
```

A modo de ejemplo, se puede emplear la función main que se recoge en Código 10.10. (que incluye la función printArray explicada en un ejemplo anterior) para probar el correcto funcionamiento de la función implementada:

Código 10.10. Implementación para la función principal del Proyecto 10.2.

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

void copyVector(short*, short*, short);
void printArray(short *, short);

int main(void)   {
      short i; //contador
      short v1[_D], v2[_D]; //vectores de enteros cortos

      //introducir enteros por teclado
      for(i = 0; i < _D; i++){
            printf("Introduzca el valor para el elemento %hd....", i);
            scanf(" %hd", &v1[i]);
      }
      //copiar de v1 a v2
      copyVector(v1, v2, _D);

      //Imprimir arrays
      printf("\n Se ha copiado el vector de enteros....\n");
      printf("\nVector V1:\t");
      printArray(v1,_D);
      printf("\nVector V2:\t");
      printArray(v2,_D);

      return 0;
}
```

Código 10.10. (Cont.)

```
void printArray(short *a, short d){
     short i;
     for(i = 0; i < d; i++){
           printf("%hd\t", a[i]);
     }
}
```

***NOTA**: Observe cómo al pasarle a las funciones* `copyVector()` *o* `printArray()` *el primero de los parámetros (el array) se escribe el nombre del array, y nada más. Podríamos invocar a la función con* `&v1[0]`*; pero ya ha quedado explicado que al indicar el nombre del array expresamos, precisamente, esa primera posición del array.*

Proyecto 10.3. PUNTEROS_BASICO_CADENAS

Debemos crear un proyecto denominado **PUNTEROS_BASICO_CADENAS**. El objetivo de este proyecto es desarrollar una función denominada `reverseString()` que permita copiar todos los caracteres de una cadena `s1` a otra cadena `s2` en sentido inverso. El prototipo debe ser como el indicado en Código 10.11.

Código 10.11. Prototipo para la función a implementar en el Proyecto 10.3.

```
     void reverseString(char *s1, char *s2);
```

NOTA: Para este ejercicio en concreto se le pide que haga uso, únicamente, de las funciones disponibles en la biblioteca `<stdio.h>`.

***NOTA**: Como ve, se puede dar nombre a los parámetros en la misma declaración de la función. Es correcto darles nombre; y es correcto también no dárselos en la propia declaración. Sí es necesario, evidentemente, dar nombre a los parámetros en la definición de la función.*

A modo de ejemplo, se puede emplear el código sugerido en Código 10.12. para probar el correcto funcionamiento de la función implementada.

Código 10.12. Implementación para la función principal del Proyecto 10.3.

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

     #define _D 100

     void reverseString(char*, char*);

     int main(void)   {
          short i;               // contador
          char c1[_D], c2[_D];   // cadenas de caracteres (strings)

     //introducir cadena de texto
          printf("Introduzca la cadena de texto (maximo %d)....",_D);
/*01*/     // gets(c1);      // recuerde que esta función no es recomendada
/*02*/     // gets_s(c1 , _D);
/*03*/     fgets(c1 , _D , stdin);
/*04*/     if(strlen(c1) < _D – 1) c1[strlen(c1) - 1] = 0;

     //copiar de c1 a c2 en sentido inverso
          reverseString(c1, c2);

          //Imprimir arrays
          printf("\nSe ha copiado el texto en sentido inverso....\n");
          printf("\nCadena c1:\t %s", c1);
          printf("\nCadena c2:\t %s", c2);

          return 0;
     }
```

Las líneas /*01*/ a /*04*/ exigen alguna aclaración: es cuestión ya trabajada en la práctica 8. Está recomendado NO hacer uso de la función gets() (línea marcada con /*01*/ en Código 10.12). La pega de esta función es que con facilidad puede causar violación de memoria. La ventaja es que es muy cómoda de usar, y en general los programadores noveles la prefieren a otras posibles. El estándar C11 recomienda la nueva función gets_s()(línea marcada con /*02*/ en Código 10.12), también cómoda de usar. La pega de esta función es que posiblemente no esté disponible todavía en muchos compiladores. Hasta el nuevo estándar, se recomendaba el uso de la función fgets()(línea marcada con /*03*/ en Código 10.12): la verdad es que es tan cómoda de usar como las otras dos; además limita la longitud de la entrada y así evita la violación de memoria (cosa, ésa, que también hace la función gets_s()). La pega es que con frecuencia inserta el carácter fin de línea como uno más dentro de la cadena. Por eso implementamos esa línea marcada como /*04*/, que elimina ese carácter de la entrada por teclado en el caso habitual de que la longitud de la entrada sea menor que el tamaño del array de la cadena.

Las líneas de código /*03*/ y /*04*/ podrían haberse reducido a una sola, teniendo en cuenta que la función fgets() tiene como valor de retorno la misma cadena de entrada recibida y almacenada en el primer parámetro de la función. Podríamos haber escrito:

```
if(strlen(fgets(c1 , _D , stdin)) < _D – 1) c1[strlen(c1) - 1] = 0;
```

Pero lo dejamos como está en Código 10.12 para no hacer el código un tanto indescifrable.

Proyecto 10.4. PUNTEROS_OPERACIONES_VECTORES

Creamos un proyecto denominado **PUNTEROS_OPERACIONES_VECTORES**.

➔ *P10.4. Ejercicio 1.*

Tenemos que definir una función denominada `randomizeVector()` que genere una colección de números aleatorios enteros y los asigne a las distintas posiciones de un vector que recibe por referencia como parámetro de entrada. La función recibirá también la dimensión del vector y los valores (a y b) que marcan el intervalo que definen el rango de los valores que se les deben asignar a las posiciones del vector. Esta función debe devolver un valor falso (0) si el rango no es correcto (a tiene que ser menor b) y un valor verdadero (por ejemplo, 1) si la función consigue rellenar con éxito el vector de números enteros aleatorios. El prototipo de esta función será el indicado en Código 10.13.

Código 10.13. Prototipo para la función a implementar en el P10.4. Ejercicio 1.

```
short randomizeVector(short *v, short d, short a, short b);
```

donde `v` es un vector de dimensión `d` que es inicializado aleatoriamente con números enteros comprendidos entre a y b.

➔ *P10.4. Ejercicio 2.*

A partir del código anterior, tenemos que implementar una nueva función denominada `statsVector()` que calcule una serie de valores estadísticos relativos a un vector `v` de valores enteros que recibe como entrada. En particular, esta función recibe como parámetros de entrada:

- Un puntero al vector cuyos elementos son números enteros cortos: `short *v;`
- La dimensión de dicho vector: `short d;`
- Un puntero a un vector de cuatro elementos para valores reales: `double *s`.

Esta función debe calcular los siguientes valores estadísticos y almacenarlos en los correspondientes elementos del vector `s`:

- `s[0]` : el mínimo del vector v
- `s[1]` : el máximo del vector v
- `s[2]` : la media del vector v
- `s[3]` : la desviación estándar muestral del vector v

El prototipo de esta función deberá ser la indicada en Código 10.14.

Código 10.14. Prototipo para la función a implementar en el P10.4. Ejercicio 2.

```
void statsVector(short *v, short d, double *s);
```

Proyecto 10.5. PUNTEROS_DETERMINANTE

Creamos un proyecto denominado **PUNTEROS_DETERMINANTE**. Tenemos que desarrollar una función denominada `determinant3()` que devuelva el valor del determinante de una matriz cuadrada de dimensiones 3x3. Para ello, se debe pasar la matriz mediante llamada por referencia. El prototipo de esta función será el indicado en Código 10.15.

Código 10.15. Prototipo para la función a implementar en el Proyecto 10.5.

```
double determinant3(short f, short c, double m[f][c]);
```

Se puede emplear el código propuesto en Código 10.16. para probar el correcto funcionamiento de la función implementada:

Código 10.16. Implementación para la función principal del Proyecto 10.5.

```
#include <stdio.h>

double determinant3(short f, short c, double m[f][c]);

int main(void)
{
      short i, j;       //contadores
      double M[3][3];   //matriz
      double det;       //valor del determinante

//introducir valores para matriz por teclado
      for(i = 0;i < 3; i++) {
            for(j = 0;j < 3; j++) {
                  printf("Valor para M[%hd][%hd]... ", i, j);
                  scanf(" %lf", &M[i][j]);
            }
      }
      //calcular determinante
      det = determinant3(3, 3, M);
```

Código 10.16. (Cont.)

```
        //Imprimir resultado
        printf("\nSu determinante es....%.3lf", det);

        return 0;
    }
```

Proyecto 10.6. PUNTEROS_DNI

Creamos un proyecto denominado **PUNTEROS_DNI**. Como sabe, la letra del Documento Nacional de Identidad (DNI) no se pone al azar. A cada número del DNI se le asigna una de entre una lista de 23 letras según un algoritmo muy simple: se calcula el resto entre el número del DNI y el entero 23; y según cuál sea ese resto, se le asigna una letra u otra, de acuerdo con la tabla que se recoge a continuación.

RESTO	0	1	2	3	4	5	6	7	8	9	10	11
LETRA	T	R	W	A	G	M	Y	F	P	D	X	B

RESTO	12	13	14	15	16	17	18	19	20	21	22
LETRA	N	J	Z	S	Q	V	H	L	C	K	E

Con todo lo anterior, tenemos que programar una función `checkDNI()` que reciba como entrada una cadena de caracteres con el DNI seguido de la letra y devuelva un 1 si la letra es correcta para ese número de DNI y 0 en caso contrario. Para ello, se debe pasar la cadena de caracteres mediante llamada por referencia. El prototipo de esta función será el sugerido en Código 10.17.

Código 10.17. Prototipo para la función a implementar en el Proyecto 10.6.

```
short checkDNI(char *dni);
```

En Código 10.18 se le ofrece un código de ayuda para desarrollar la función.

Tenga en cuenta que el DNI estará almacenado en una cadena de caracteres de dimensión 9: los 8 primeros elementos corresponden a los distintos dígitos del número y el último es la letra del DNI. Así por ejemplo: `char dni[9] = "12345678A";`

A la hora de probar el correcto funcionamiento de nuestra función, se puede hacer uso del programa mostrado en Código 10.19. No hace falta que le sugiramos entradas: puede probar usted ahora si su documento es o no es correcto. Por si usted no goza de nacionalidad española, o todavía no tiene la edad exigida para estar correctamente documentado, puede crear entradas correctas

ejecutando el programa corto que le proponemos en Código 10.20. Introduzca valores enteros de no más de 8 dígitos y obtendrá como salida la letra correcta para un posible dni con ese número de entrada. Cuando quiera terminar la ejecución del programa introduzca el valor 0. Además, seguro que el programa de Código 10.20 le ofrece alguna pista sobre cómo crear la función `checkDNI()`.

Código 10.18. Ayuda para definir la función `checkDNI()`

```
short checkDNI(char *d) {
char L[] = {'t', 'r', 'w', 'a', 'g', 'm', 'y', 'f', 'p', 'd', 'x',
            'b', 'n', 'j', 'z', 's', 'q', 'v', 'h', 'l', 'c', 'k', 'e'};

      // Aquí se debe insertar el código de la función

}
```

Código 10.19. Implementación para la función principal del Proyecto 10.5.

```
#include <stdio.h>
#include <ctype.h>
#include <conio.h>
#include <math.h>

short checkDNI(char*);

int main(void) {
      char dni[9]; //cadena de caracteres (string) para el DNI (12345678A)
      char c;
      short i = 0;
      short correcto;

      printf("COMPROBAR DNI");
//introducir DNI
      do {
            printf("\nIntroduzca el digito %hd de su DNI....", i+1);
            c = getch();
            if(isdigit(c)){
                  printf("%c", c);
                  dni[i++] = c;
            }
      } while(i < 8);

      do {
            printf("\nIntroduzca la letra de su DNI....", i+1);
            c = getch();
      } while(isalpha(c) == 0);

      printf("%c", c);
      dni[i++] = c;
```

Código 10.19. (Cont.)

```
//verificar letra del DNI
      correcto = checkDNI(dni);

      if(correcto == 1){
            printf("\nLa letra introducida es correcta");
      } else {
            printf("\nLa letra introducida no es correcta");
      }

      return 0;
}
```

NOTA**: Advierta que aunque el array* `dni` *es de tipo* **char, no se pretende en este caso manejar ese array como una cadena de texto, sino como un array de valores independientes. Por eso, en este ejemplo, no nos ocupamos de cerrar la cadena con el carácter de fin de cadena.*

Código 10.20. Programa para obtener la letra correcta de un número de dni.

```
#include <stdio.h>
int main(void)
{
      unsigned long dni;
      char L[] = {'t', 'r', 'w', 'a', 'g', 'm', 'y', 'f', 'p', 'd', 'x',
            'b', 'n', 'j', 'z', 's', 'q', 'v', 'h', 'l', 'c', 'k', 'e'};

      do
      {
            printf("dni ... ");     scanf(" %lu", &dni);
            printf("letra: %c\n", L[dni % 23]);
      }while(dni);

      return 0;
}
```

Proyecto 10.7. PUNTEROS_ORDENACION [EJERCICIO ADICIONAL]

Creamos un proyecto denominado **PUNTEROS_ORDENACION**. A partir del código desarrollado en la práctica de vectores y matrices, hay que programar una nueva función que permita ordenar los elementos de un vector utilizando el método del algoritmo del intercambio. El prototipo de esta función es el sugerido en Código 10.21.

Código 10.21. Prototipo para la función a implementar en el Proyecto 10.7.

```
void ordenacion(short *a, short d);
```

Los parámetros de entrada para esta función son un vector a de enteros cortos (es pasado por referencia utilizando punteros) y la dimensión del mismo (d).

Proyecto 10.8. PUNTEROS_RIEL [EJERCICIO ADICIONAL]

Creamos un proyecto denominado **PUNTEROS_RIEL**. Debemos implementar una nueva función que permita cifrar una cadena de texto empleando la transposición de Riel. Esta trasposición obtiene una cadena a partir de una inicial, formada primero por todos los caracteres ubicados en las posiciones pares de la cadena inicial, seguidos luego por todos los caracteres ubicados en las posiciones impares.

Por ejemplo, del texto `"ESTO ES UN EJEMPLO DE FRASE A CODIFICAR"` los caracteres ubicados en posición par son: `"ET SU JML EFAEACDFCR"`; y los ubicados en posición impar son: `"SOE NEEPOD RS OIIA".` Y concatenados forman el siguiente anagrama: `"ET SU JML EFAEACDFCRSOE NEEPOD RS OIIA"`.

El prototipo de esta función puede ser el propuesto en Código 10.22.

Código 10.22. Prototipo para la función a implementar en el Proyecto 10.8.

```
void rielCoding(char *source , char *target);
```

donde `source` es la cadena de caracteres a codificar utilizando el algoritmo de transposición de Riel, y `target` la cadena codificada según el algoritmo de Riel.

SOLUCIONES A LOS EJERCICIOS DE LOS PROYECTOS PROPUESTOS

Código 10.23. Solución para Proyecto 10.1.

```
#include <stdio.h>

void intercambio(short*, short*);

int main(void) {
      short a, b;

      printf("Introduzca el valor para a....");      scanf(" %hd", &a);
      printf("Introduzca el valor para b....");      scanf(" %hd", &b);

      printf("\nIntercambio de valores....\n");
      intercambio(&a, &b);

      printf("El valor de a es....%hd\n", a);
      printf("El valor de b es....%hd\n\n", b);

      return 0;
}

void intercambio(short *a, short *b) {
      short aux = *a;
      *a = *b;
      *b = aux;
      return;     // opcional
}
```

Código 10.24. Solución para Proyecto 10.2.

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

void copyVector(short*, short*, short);
void printArray(short*, short);

int main(void)
{
      short v1[_D], v2[_D]; //vectores de enteros cortos

//introducir enteros por teclado
      for(i = 0; i < _D; i++){
            printf("Introduzca el valor para el elemento %hd....",i);
            scanf(" %hd", &v1[i]);
      }
```

Código 10.24. (Cont.)

```
	//copiar de v1 a v2
	copyVector(v1, v2, _D);

	//Imprimir arrays
	printf("\n Se ha copiado el vector de enteros....\n");
	printf("\nVector V1:\t");
	printArray(v1,_D);
	printf("\nVector V2:\t");
	printArray(v2,_D);

	return 0;
}

void printArray(short *a, short d){
	short i;
	for(i = 0; i < d; i++){
		printf("%hd\t", a[i]);
	}

	return;		// opcional
}

void copyVector(short *v, short *w, short d){
	short i;
	for(i = 0; i < d; i++){
		w[i] = v[i];
	}

	return;		// opcional
}
```

Código 10.25. Solución para Proyecto 10.3.

```
#include <stdio.h>

#define _D 100

void reverseString(char*, char*);

int main(void){
	short i; //contador
	char c1[_D], c2[_D]; //cadenas de caracteres (strings)

//introducir cadena de texto
	printf("Introduzca la cadena de texto (maximo %d)....",_D);
	if(strlen(fgets(c1 , _D , stdin)) < _D – 1) c1[strlen(c1) - 1] = 0;
```

Código 10.25. (Cont.)

```
//copiar de c1 a c2 en sentido inverso
      reverseString(c1,c2);

//Imprimir arrays
      printf("\nSe ha copiado la cadena de texto en sentido inverso.\n");
      printf("\nCadena c1:\t %s",c1);
      printf("\nCadena c2:\t %s",c2);

      return 0;
}

void reverseString(char *s1, char *s2){
      short i;
      short L;    // Sobre ella calculamos la longitud de la cadena

      for(L = 0 ; s1[L] ; L++);     // Así L es la longitude de s1.

      for(i = 0 ; s1[i] ; i++){
            s2[i] = s1[L- i - 1];
}
      s2[i]='\0';
}
```

Código 10.26. Solución para P10.4. Ejercicio 1.

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

short randomizeVector(short *v, short d, short a, short b)
{
      if(a < b)
      {
            short i;    // local al bloque del if.
            srand(time(NULL));

            for(i = 0 ; i < d ; i++ , srand(rand()))
            {
                  v[i] = rand() % (b – a + 1) + a; //generar aleatorio
            }

            return 1;
      }     // Vea: aquí la variable i ya no existe.

      return 0;   // Si a no es menor que b
}
```

Código 10.27. Solución para P10.4. Ejercicio 2.

```
void statsVector(short *v, short d, double *s)
{
	int i;
	//s[0]: minimo de v;		s[1]: maximo de v;
	//s[2]: media de v;		s[3]: desviacion estándar de v

	for(i = 1 , s[0] = s[1] = s[2] = v[0] ; i < d ; i++)
	{
		if(s[0] > v[i])	s[0] = v[i];
		if(s[1] < v[i])	s[1] = v[i];
		s[2] += v[i];
	}
	s[2] = s[2]/d; //media de v

	for(i = 0 , s[3] = 0 ; i < d ; i++) {
		s[3] += (v[i] - s[2]) * (v[i] - s[2]);
	}
	s[3] = sqrt(s[3]/(d – 1.0)); //desv. estandar muestral de v

	return 0;
}
```

Código 10.28. Solución para Proyecto 10.5.

```
#include <stdio.h>

double determinant3(short f, short c, double *m[f][c]);

int main(void){
	short i,j; //contadores
	double M[3][3]; //matriz
	double det; //valor del determinante

//introducir valores para matriz por teclado
	for(i = 0; i < 3; i++){
		for(j = 0; j < 3; j++){
			printf("Valor para M[%hd][%hd]....", i, j);
			scanf(" %lf", &M[i][j]);
		}
	}

//calcular determinante
	det = determinant3(3, 3, M);
//Imprimir arrays
	printf("\nSu determinante es....%.3lf",det);
	return 0;
}
```

Código 10.28. (Cont.)

```
      double determinant3(short f, short c, double m[f][c]) {
            double D = 0;

            D += *(*(m + 0) + 0) * *(*(m+1)+1) * *(*(m + 2) + 2);
            D += *(*(m + 0) + 1) * *(*(m + 1) + 2) * *(*(m + 2) + 0);
            D += *(*(m + 0) + 2) * *(*(m + 1) + 0) * *(*(m + 2) + 1);
            D -= *(*(m + 0) + 2) * *(*(m + 1) + 1) * *(*(m + 2) + 0);
            D -= *(*(m + 0) + 1) * *(*(m + 1) + 0) * *(*(m + 2) + 2);
            D -= *(*(m + 0) + 0) * *(*(m + 1) + 2) * *(*(m + 2) + 1);

            return D;
      }

 /*
 Otra forma de definir la funcion, utilizando operatoria de indices y no de
 punteros, es la siguiente:

      double determinant3(short f, short c, double v[f][c]) {
            double D = 0;

            D += v[0][0] * v[1][1] * v[2][2];
            D += v[0][1] * v[1][2] * v[2][0];
            D += v[0][2] * v[1][0] * v[2][1];
            D -= v[0][2] * v[1][1] * v[2][0];
            D -= v[0][1] * v[1][0] * v[2][2];
            D -= v[0][0] * v[1][2] * v[2][1];

            return D;
      }
 */
```

Código 10.29. Solución para Proyecto 10.6.

```
      #include <stdio.h>
      #include <ctype.h>
      #include <conio.h>
      #include <math.h>

      short checkDNI(char*);

      int main(void)
      {
            char dni[9]; //cadena de caracteres (string) para el DNI (12345678A)
            char c;
            short i = 0;
```

Código 10.29. (Cont.)

```
        printf("COMPROBAR DNI");
        do
        { //introducir DNI
              printf("\nIntroduzca el digito %hd de su DNI....", i+1);
              c = getch();
              if(isdigit(c)) {
                    printf("%c", c);
                    dni[i++] = c;
              }
        } while(i < 8);

        do
        {
              printf("\nIntroduzca la letra de su DNI....", i+1);
              c = getch();
        } while(isalpha(c) == 0);

        printf("%c", c);
        dni[i++] = c;

  //verificar letra del DNI

        if(checkDNI(dni))
        {
              printf("\nLa letra introducida es correcta");
        }
        else
        {
              printf("\nLa letra introducida no es correcta");
        }

        return 0;
  }

  short checkDNI(char *d)
  {
        char letras[]=   {'t', 'r', 'w', 'a', 'g', 'm', 'y', 'f', 'p', 'd',
                          'x', 'b', 'n', 'j', 'z', 's', 'q', 'v', 'h', 'l',
                          'c', 'k', 'e'};
        short i;
        char cadena[9];

        for(i = 0; i < 9 && isdigit(cadena[i]) ; i++)  cadena[i] = d[i];
        cadena[i] = '\0';

        if(d[i] == letras[atoi(cadena) % 23])    return 1;
        return 0;
  }
```

Código 10.30. Solución para Proyecto 10.7.

```
     void ordenacion(short *a, short d) {
           short i, j;

           for(i = 0 ; i < d ; i++) {
                 for(j = i + 1 ; j < d ; j++) {
                       if(*(a + i) > *(a + j))
                             intercambio (a + i , a + j);
                 }
           }
     }
/*Donde la funcion intercambio() también deberia ser declarada y definida, y
que ya ha sido mostrada en el Proyecto 10.1.*/
```

Código 10.31. Solución para Proyecto 10.8.

```
     void rielCoding(char *source , char *target)
     {
           short i, j, longitud = 0;

           for(longitud = 0 ; source[longitud] ; longitud++);

           for(i = 0 , j = 0 ; i < longitud ; i += 2 , j++)     // Los pares
                 target[j] = source[i];
           for(i = 1 ; i < longitud ; i += 2 , j++)             // Los impares
                 target[j] = source[i];
           target[i] = 0;

           return;
     }
```