Depto. de Matemáticas del área industrial

Universidad Politécnica de Madrid

Introducción a la programación en C

Grado en Ingeniería Mecánica y Grado en Diseño Industrial

(curso 2016/17)

JAVIER SANGUINO

Presentación

Con este documento se cubren exhaustivamente los contenidos de la asignatura de Introducción a la programación en C, correspondientes al Grado de Ingeniería Mecánica y al Grado de Diseño Industrial de la ***Escuela Técnica Superior de Ingeniería y Diseño Industrial (*****ETSIDI*****)*** de la *Universidad Politécnica de Madrid.*

El objetivo de este curso es que el alumno aprenda las herramientas básicas de programación a través del **lenguaje C**. Por tanto, los contenidos comienzan desde un nivel elemental y se van desarrollando *progresivamente* hasta poder realizar programas de cierta complejidad.

Puede resultar un tanto sorprendente la elección de **lenguaje C** como vehículo para generar códigos, siendo el objetivo principal el aprendizaje de la programación. Hay lenguajes más adecuados para esta finalidad, como Pascal o Módula–2. La razón es que el **lenguaje C** viene impuesto en el Plan de Estudios, con la idea de buscar una uniformidad con las otras asignaturas de Informática, que se imparten en los restantes Grados de la (ETSIDI). Es cierto, que la flexibilidad del **lenguaje C** puede transformarse en una dificultad durante el proceso de aprendizaje, pero también es cierto que el **C** ofrece unas posibilidades muy amplias, que permiten al alumno tener una perspectiva muy clara de la amplitud y potencia de este lenguaje. Opino que es valioso que alumno conozca la existencia de conceptos y posibilidades, de los que otros lenguajes puedan carecer, como por ejemplo, el acceso a direcciones de memoria mediante *punteros*, o la idea de *recursividad*. Así mismo, constituye la puerta de entrada para el estudio del **C++**, que es una herramienta destacada para abordar una *programación orientada a objetos*, tan importante en multitud de áreas de la ingeniería industrial.

Con la idea de cumplir los objetivos del curso, este documento tiene la particularidad de ser *fundamentalmente práctico*. No he querido describir ni plantear explícitamente conceptos teóricos de programación, como tampoco transformarlo en un manual de **C**. En este texto, he pretendido conjugar ambos aspectos a través de los elementos que ofrece el **lenguaje C**. Por ello, aunque no resulte muy *ortodoxo*, he considerado un capítulo inicial (**Primeros programas**) en los que he planteado algunas herramientas básicas de la programación, que traducidas al **lenguaje C** (`if, if-else, while`), permite al alumno escribir en **C** una gran cantidad de algoritmos básicos, desde el comienzo.

En mi opinión, el alumno que quiera aprender a programar **tiene** que programar. Por este hecho, me he cuidado mucho de que el documento tuviese gran cantidad de ejemplos comentados (que el alumno puede reproducir) así como, una variedad de ejercicios al final de cada capítulo, relacionados directamente con lo visto en ese capítulo o bien, en capítulos anteriores. Estos ejercicios se han tratado de ordenar con una dificultad gradual y no se han escogido al azar, sino que se han buscado de forma que contribuyan a clarificar conceptos importantes. Es imprescindible que alumno se enfrente a ellos y se equivoque. Se aprende mucho de los propios errores. Con este compromiso, no he querido acompañar los ejercicios con las respectivas soluciones, cuando además, hay una gran diversidad de ejemplos resueltos en cada capítulo.

En cuanto a las características propias del **C**, he seguido las que ofrece el compilador del entorno Dev-C++ ver. 5.2. Es un compilador que se aproxima a las características del **C99**. En este documento sólo hago referencia al **C**, y no a las diferentes versiones. Creo que de otra forma, sería un error. Cada vez más compila-

dores se ajustan a las características de los estándares **C99** o **C11**, con lo cual no debe ser problemático asumir características propias de estas versiones como propias del **C**. En este contexto, es importante señalar que en este documento he optado por utilizar el *dimensionamiento dinámico* (*Variable-Length Arrays*), en lugar de acceder a esta idea a través del *direccionamiento dinámico* mediante la función **`malloc`**. No obstante, ambos procedimientos están presentes en este texto, pero como experiencia docente, este último procedimiento crea una mayor confusión al alumno.

El documento se ha realizado mediante LaTeX 2_ε, y para mostrar los programas y sus resultados se ha utilizado la configuración ofrecida por David Villa a través de Crysol (`http://crysol.org/node/909`).

Desde aquí quiero agradecer a mi compañera Carmen García–Miguel, la paciencia y gentileza de leer versiones anteriores y cuyas críticas constructivas, sugerencias y correcciones, me ha permitido mejorar enormemente este documento. Así mismo, los comentarios de Pedro Galán me han resultado muy oportunos, muchas gracias.

Madrid

enero de 2017

Índice general

CAPÍTULO 1

Elementos de un ordenador

1.1. Introducción

Ya se ha comentado en la Presentación de este documento, que aunque el **lenguaje C** dispone de las estructuras típicas de los lenguajes de *alto nivel*, a su vez dispone de comandos u órdenes que permiten un control a bajo nivel. Esta característica hace que antes de comenzar con el aprendizaje del **lenguaje C**, sea importante plantear una serie de conceptos relacionados con el funcionamiento de un *ordenador*, que faciliten este estudio y permitan utilizar correctamente algunas de las herramientas que ofrece este lenguaje, para realizar códigos eficientes y flexibles.

1.1.1. Breve apunte histórico

A lo largo de la historia no existe una fecha que represente el momento de la invención del ordenador, tal y como lo conocemos en nuestros días. Se ha producido una evolución constante, desde el simple ábaco, hasta los microprocesadores, responsables del funcionamiento de los ordenadores contemporáneos (véase Cantone [3, pág. 59 y sig.]). Sin embargo, desde mi punto de vista, históricamente ha habido dos contribuciones que fueron claves para disponer de ordenadores con las características de los actuales. Por una parte Alan M. Turing y por otra John von Neumann.

- **John von Neumann**. Nació en Budapest en 1903. Es uno de los matemáticos más brillantes de la historia (recomiendo consultar Wikipedia). Hizo numerosas y significativas contribuciones en diferentes áreas de las matemáticas y física, desde la rama pura a la más aplicada. Trabajó en el *Instituto de Estudios Avanzados* de Princeton, junto con otros científicos de la época (entre ellos Albert Einstein). Fue así mismo, un científico destacado en el *Proyecto Manhatann* dirigido por J.R. Openheimmer. En aquella época era habitual subestimar los peligros de la radiación y como consecuencia de las sufridas durante los ensayos de la Bomba Atómica, murió en 1957 en Washington D.C.

  Destacó en las *Ciencias de la computación* contribuyendo en cómo debía ser la configuración física de un ordenador y dando lugar a la llamada *arquitectura von Neumann*, que es utilizada en casi todos los computadores actuales. Su concepto se basa en tres pilares:

  1. Una ***unidad de control*** que se encargaría de organizar el trabajo de la máquina: regular las instrucciones, su cálculo e intercambio de información con el exterior.

2. Una ***memoria*** importante que fuera capaz de almacenar no sólo datos sino también instrucciones.
3. Un ***programa*** que indicase a la máquina los pasos a seguir sin necesidad de esperar las sucesivas órdenes desde el exterior, como se venía haciendo desde entonces.

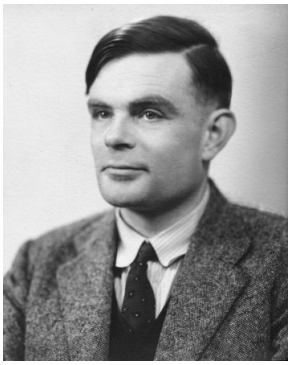
En 1949 construyó un ordenador con estas características: EDVAC. El primer ordenador comercial fabricado con esta configuración fue el UNIVAC I en 1951.

- **Alan Turing**. Nació en Londres en 1912. Fue matemático y es considerado uno de los padres de la ciencia de la computación siendo el precursor de la informática moderna. Proporcionó una influyente formalización de los conceptos de algoritmo y computación: la máquina de Turing. En cierta manera estableció los pilares teóricos sobre los conceptos de algoritmo y programa, considerando que la resolución de ciertos problemas podían ser descompuestos en tareas más simples abordables de manera automática.

  Tuvo una aportación clave en la II SEGUNDA GUERRA MUNDIAL, al contribuir de manera significativa en el descubrimiento de la codificación de la máquina *Enigma* y permitir a los aliados anticipar los ataques y movimientos militares nazis.

  En 1952 Turing reconoció en un juicio su homosexualidad, con lo que se le imputaron los cargos de *indecencia grave y perversión sexual* (los actos de homosexualidad eran ilegales en el Reino Unido en esa época). Convencido de que no tenía de qué disculparse, no se defendió de los cargos y fue condenado. Dos años después del juicio, en 1954, Turing se suicidó mediante la ingestión de una manzana contaminada con cianuro.

(a) John von Neumann



(b) Alan Turing

Figura 1.1 – *Iniciadores de la informática actual.*

1.2. Componentes de un ordenador

En este curso de Informática, el modelo de estructura de ordenador que se va a seguir es la llamada *arquitectura von Neumann*, correspondiente a los primeros computadores, pero que sigue siendo conceptualmente válido hoy en día. En general un computador sencillo se compone de los siguientes elementos tal y como se observa en la figura 1.2:

Unidades de entrada. Consisten en dispositivos que permiten introducir datos e instrucciones en el ordenador. En estas unidades se transforma la información, en impulsos eléctricos de tipo binario. Un computador puede tener diferentes dispositivos de entrada: teclado, *ratón*, escáner de imágenes, lectora de tarjetas, *etc.*

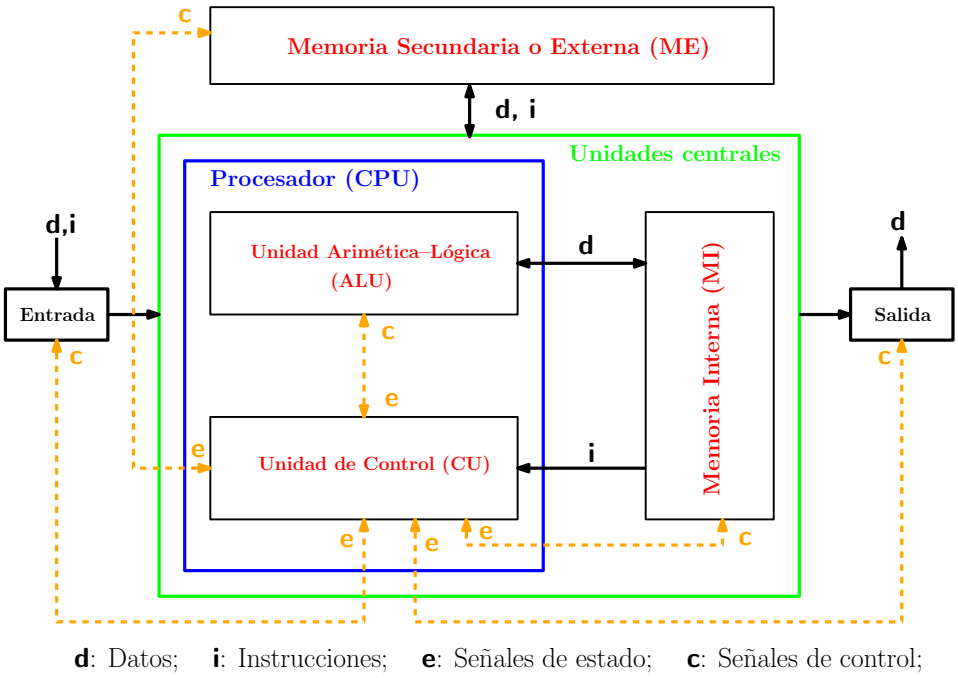


Figura 1.2 – *Elementos básicos de un ordenador (arquitectura von Neumann)*

Unidades de salida. Son los instrumentos por el que se muestran los resultados de los programas ejecutados en el ordenador. Estos dispositivos transforman las señales binarias en información inteligible para el usuario: pantalla, altavoz, impresora, *etc.*

Unidad central de proceso (CPU, Central Processing Unit). Se trata de un microprocesador que constituye el *cerebro* del ordenador (véase figura 1.2). Permite sincronizar las diferentes instrucciones que deben ejecutarse en cada instante y además proporciona señales de temporización y de control que posibilitan la introducción y eliminación de las instrucciones y datos, según se vayan realizando. Por tanto, se trata de un dispositivo que procesa instrucciones básicas que son necesarias tanto para el correcto funcionamiento del ordenador, como para la ejecución adecuada de los programas.

Alguno de los elementos básicos que constituyen la CPU son

- ***Unidad de control* (CU, Control Unit).** Detecta las las *señales eléctricas de estado* procedentes de las distintas unidades, indicando su situación o condición de funcionamiento (véase figura 1.2). También recibe secuencialmente de la memoria las instrucciones del programa en ejecución, y de acuerdo con el código de operación de la instrucción captada y con las *señales de estado* procedentes de los distintos dispositivos del ordenador, genera ***señales de control*** dirigidas a todas las unidades, ordenando las operaciones que implican la ejecución de una instrucción.

  La *Unidad de control* contiene un ***reloj***, que es un *generador de impulsos eléctricos* que sincroniza todas las operaciones elementales del ordenador. El período de esta señal se denomina *tiempo de ciclo* y está comprendido aproximadamente entre décimas de nanosegundos y microsegundos, dependiendo del procesador. La ***frecuencia del reloj*** (inverso del tiempo de ciclo) suele expresarse en millones de *ciclos/segundo* (*Megahercios* o **MHz**) o miles de millones de *ciclos/segundo* (*Gigahercios* o **GHz**). La ejecución de cada instrucción supone la realización de un conjunto de operaciones elementales consumiendo un número predeterminado de ciclos, de forma que las instrucciones más complejas utilizan un número mayor de ciclos que las más simples (véase Cantone [3, pág. 46]).

- ***Unidad aritmético-lógica*** **(ALU, Arithmetic Logic–Unit).** Contiene los circuitos electrónicos con los que se hacen las operaciones del tipo aritmético (sumas, restas, *etc*) y de tipo lógico (comparar dos números, operaciones del álgebra de Boole binaria, *etc.*). Está directamente relacionada con ciertos *registros* que están en el interior de la CPU y que almacenan temporalmente datos e instrucciones, con el objetivo de operar con ellos.
- ***Registros.*** Se llama ***registro*** a una pequeña memoria diseñada para almacenar un dato, instrucción o dirección de memoria. Están constituidos por un conjunto de *bits* (ya se definirá más adelante), normalmente 32 ó 64 *bits*. Existen diferentes tipos de registro dentro de la CPU
  - *Registro de Contador de Programa.* Se trata de un registro que almacena la dirección de memoria de la siguiente instrucción a ejecutar.
  - *Registro de Instrucciones.* Se almacena la instrucción que en ese momento ejecuta la *Unidad de Control* (CU).
  - *Registro de dato de memoria.* Almacena la información (instrucción o dato de instrucción) y cuya dirección se encuentra en el *Registro de dirección de memoria*.
  - *Registro de dirección de memoria.* Almacena la dirección de la instrucción o dato que se encuentra en el *Registro de dato de memoria*.
  - *Matriz de Registros.* Consiste en una disposición de registros de propósito general, que se emplean en el proceso intermedio de ejecución de instrucciones. Una de sus utilidades consiste en almacenar los datos y direcciones de memoria, que serán necesarios en los procesos siguientes. La ALU puede acceder muy rápidamente a estos registros, lo que permite que el programa se ejecute de manera más eficiente.

Memoria interna (MI). Es la unidad que almacena tanto los datos, como las instrucciones durante la ejecución de los programas. La *memoria interna* (también se denomina ***memoria central*** o ***memoria principal***) actúa con una gran velocidad y está ligada directamente a las unidades más rápidas del computador: *unidad de control* y *unidad aritmético–lógica*. Para que un programa se ejecute debe estar almacenado (o *cargado*) o al menos parte, en la *memoria principal*.

Normalmente hay una zona de la memoria que sólo se puede leer, llamada ***ROM (Read Only Memory)*** y que es permanente (se mantiene aunque se desconecte el ordenador) y otra en la que se puede leer y escribir, llamada ***RAM (Random Access Memory)*** pero que es volátil.

La memoria ROM suele venir grabada de fábrica y contiene un conjunto de programas llamados ***BIOS*** (*Basic Input/Output System*) encargados de hacer posible inicialmente, el acceso y la gestión básica de algunos dispositivos, como el disco duro y los puertos (conexiones externas de entrada/salida). Contiene subrutinas de inicialización de parámetros y gestiona determinadas solicitudes procedentes de los periféricos, denominadas ***interrupciones***. Así mismo, posee las instrucciones para buscar y cargar el Sistema Operativo del dispositivo de almacenamiento secundario, en el momento de encender el ordenador.

Por otra parte, la memoria RAM constituye el espacio de trabajo de la CPU. En ella se almacenan los programas y también reside el Sistema Operativo cuando está activado el ordenador. Se divide a su vez, en dos tipos

- **RAM Dinámica (DRAM).** En los ordenadores actuales esta memoria está formada por capacitores, dispuestos en circuitos electrónicos integrados, llamados ***chips*** (véase Cantone [3, pág.50]). Debido a que tiende a descargarse es necesario refrescarlas periódicamente.
- **RAM Estática (SRAM).** Los datos se almacenan de manera fija, por lo que no necesitan refresco. Son más rápidas que las memorias DRAM y más caras. Estás memorias también se conocen como ***memorias Caché***. Su objetivo es contener información que utiliza frecuentemente el procesador,

de esta manera no es necesario descargarlas de la DRAM constantemente y aumentan la rapidez de ejecución. Existen diferentes niveles:

- L_1. Es una memoria que va desde los 32 KB a los 64 KB[1]. Tiene la particularidad de estar dentro del procesador, por eso este tamaño *reducido*, pero por contra es muy rápida.
- L_2. Es de mayor tamaño que la anterior (puede llegar a un 512 KB). No está situada dentro del microprocesador pero también puede acceder a ella muy rápidamente.
- L_3. Análoga a la L_2, pero de tamaño mayor a ésta (puede llegar a 2 ó 3 MB.)

La aparición de procesadores (CPU'S) de más de un ***núcleo***[2] ha replanteado la estructura de la *memoria Caché*. En este sentido, la compañía *Intel* ha desarrollado una memoria llamada ***Smart Caché*** (que anuncia como característica *Caché* en sus microprocesadores) y que *grosso modo*, consiste en que cada núcleo posea su *Caché* del nivel L_1, pero junta los niveles L_2 y L_3, que son compartidos por los núcleos del microprocesador (véase figura 1.3). De esta manera, la memoria *Smart Caché* puede llegar a tener un tamaño de 3 Mb.

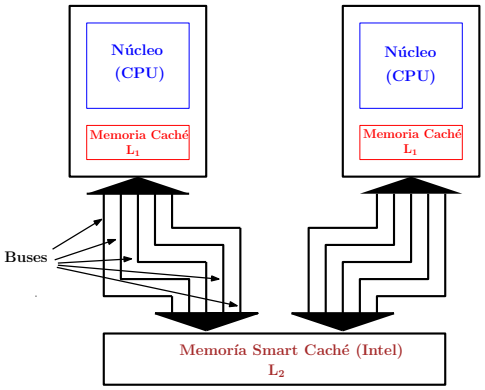


Figura 1.3 – *Esquema de la Memoria Smart Caché desarrollada por* Intel

Memoria externa (ME). La zona RAM de la memoria interna es muy rápida, pero no tiene gran capacidad y es volátil (esto quiere decir que cuando se apaga el ordenador, se pierde la información almacenada en ella). Para guardar masivamente información de manera permanente, se utilizan otros tipos de memoria, que están dispuestos en *discos magnéticos*, *discos ópticos* y *cintas magnéticas*. Actualmente se utilizan también discos duros externos con conexiones USB. El acceso a los datos es mucho más lento (alrededor de un millón de veces), pero es un modelo permanente y la capacidad de almacenamiento es mucho mayor (alrededor de mil veces). El conjunto de estas unidades se llama ***memoria externa, secundaria o auxiliar***. Usualmente los datos y programas se graban en la memoria externa, de esta forma cuando se ejecuten varias veces, no es necesario introducirlo por los *dispositivos de entrada*. La información almacenada permanece en el dispositivo indefinidamente, hasta que se borre expresamente por el usuario.

Buses. La transmisión de la información a través de los distintos dispositivos del ordenador se realiza a través de un conjunto de hilos, líneas o pistas de conductores eléctricos, llamados ***buses***. Sirven para interconectar dos o más componentes funcionales de un sistema o de varios sistemas distintos. Estas conexiones suelen llevar en un instante dado la información completa de una instrucción, un dato o una dirección. El ***ancho de bus*** es el número de hilos que contiene, o número de bits que transmite simultáneamente en paralelo (véase Floyd [5, pág. 786] o bien Cantone [3, pág. 49]). Por tanto, un *bus* se caracteriza por la cantidad de información que se transmite a través de estos hilos, de forma simultánea. Este volumen se

[1] Más adelante, en el cuadro 1.1 se presentan las medidas de la memoria de un ordenador.
[2] Se entiende por *núcleo* (o *core*) al conjunto formado por una ALU y una CU

expresa en *bits* y corresponde al número de líneas físicas (*hilos*) mediante las cuales se envía la información en forma simultánea. Un cable plano de 32 hilos permite la transmisión de 32 bits en paralelo. Así pues, el propósito de los *buses* es poder transmitir información de manera simultánea y reducir el tiempo de acceso a la información entre los distintos componentes, al realizar las comunicaciones a través de un sólo canal de datos. Ésta es la razón por la que, a veces, se utiliza la metáfora *autopista de datos*. En el caso en que sólo dos componentes de hardware se comuniquen a través de la línea, podemos hablar de ***puerto hardware*** (*puerto serial o paralelo*).

Por otra parte, la *velocidad del bus* se define a través de la ***frecuencia*** (medida en Hercios), es decir el número de paquetes de datos que pueden ser enviados o recibidos por segundo. Cada vez que se envían o reciben estos datos podemos hablar de ciclo. Este proceso está directamente relacionado con la velocidad de la *Unidad de Control*.

Así pues, es posible hallar la ***velocidad de transferencia*** máxima del bus (la cantidad de datos que puede transportar por unidad de tiempo) al multiplicar su ancho por la frecuencia. Por lo tanto, un bus con un ancho de 16 *bits* y una frecuencia de 133 *MHz*, tiene una velocidad de transferencia de:

$$16 * 133 \cdot 10^6 = 2128 \cdot 10^6\, bits/sg \approx 266 MB/sg$$

En realidad, cada bus se halla generalmente constituido por hilos eléctricos cuyo número suelen ser potencias de 2 ($2^5 = 32, 2^6 = 64, 2^7 = 128$, *etc*). Estos hilos se dividen a su vez en tres subconjuntos:

- El **bus de direcciones** (o *bus de memoria*) transporta las direcciones de memoria al que el procesador desea acceder, para leer o escribir datos. Se trata de un bus *unidireccional*.
- El **bus de datos** transfiere tanto las instrucciones que provienen del procesador como las que se dirigen hacia él. Se trata de un bus *bidireccional*.
- El **bus de control** transporta las órdenes y las señales de sincronización que provienen de la unidad de control y viajan hacia los distintos componentes de hardware. Se trata de un bus *bidireccional* en la medida en que también transmite señales de respuesta del hardware.

En algunas ocasiones se habla de ***buses multiplexados*** que utilizan líneas eléctricas *multiplexadas* para el *bus de direcciones* y el *bus de datos*, por ejemplo. Esto significa que un mismo conjunto de líneas eléctricas se comportan unas veces como bus de direcciones y otras veces como bus de datos, pero nunca al mismo tiempo. Una línea de control permite discernir cual de las dos funciones está activa.

1.3. Disposición de la memoria

1.3.1. Introducción

Como ya se ha comentado, una de las particularidades del lenguaje **C** es la posibilidad de acceder a diferentes posiciones de memoria, lo que permite generar códigos muy eficientes y flexibles. Es por ello, que resulta conveniente tener una idea clara de la disposición y organización de la memoria dentro de un ordenador.

1.3.2. Nomenclatura

Desde un punto de vista electrónico no es difícil confeccionar un dispositivo que tenga únicamente dos estados: *encendido o apagado* (véase Cantone [3, pág. 44]). A este tipo de dispositivo se le denomina bit (BInary digiT). Es decir, se trata de una posición, celda o variable que toma el valor 0 o 1. Resulta obvio que la información que se puede almacenar en un *bit*, resulta muy pobre. Para aumentar esta capacidad de información se define el ***byte***, que es un conjunto de 8 *bits*. Esto aumenta claramente la información, pues se pasa de 2, a

Símbolo	Equivalencia	Nomenclatura
Byte (B)	8 *bits*	Byte
Kbyte (KB)	1024 *bytes* = 2^{10} *bytes*	Kilobyte
Mbyte (MB)	1024 *Kbytes* = 2^{20} *bytes*	Megabyte
Gbyte (GB)	1024 *Mbytes* = 2^{30} *bytes*	Gigabyte
Tbyte (TB)	1024 *Gbytes* = 2^{40} *bytes*	Terabyte
Pbyte (PB)	1024 *Tbytes* = 2^{50} *bytes*	Petabyte
Ebyte (EB)	1024 *Pbytes* = 2^{60} *bytes*	Exabyte
Zbyte (ZB)	1024 *Ebytes* = 2^{70} *bytes*	Zettabyte
Ybyte (YB)	1024 *Zbytes* = 2^{80} *bytes*	Yottabyte

Cuadro 1.1 – *Medidas de almacenamiento de la memoria interna*

$2^8 = 256$ maneras diferentes de representación. Así pues, un *byte* es una disposición de 8 *bits*, que puede tomar 256 valores (todos en forma de 0's y 1's). Es habitual considerar el *byte* como medida de *bits*. Así pues, se habla 3 *bytes* en lugar de 24 *bits*.

Puesto que los *bytes* (en particular los *bits*) se pueden utilizar para almacenar información, el tamaño de la memoria de un ordenador se mide mediante la cantidad de *bytes* disponibles o a los que puede acceder la CPU. Con el paso de los años la tecnología se va perfeccionado y la capacidad de los ordenadores va en aumento. Es habitual utilizar la nomenclatura del cuadro 1.1, para medir el tamaño de la memoria. Como puede observarse, se utilizan factores de $2^{10} = 1024$, en lugar de 10^3 para definir las diferentes unidades.

1.3.3. Disposición

Para que el acceso a la memoria por parte de la CPU sea lo más eficiente posible, la memoria se organiza en grupos de *celdas* (cada *celda* correspondería a un *bit*) y a cada grupo de celdas se le asigna un número (entero sin signo) que se llama ***dirección***. Por lo general (principalmente en los ordenadores domésticos), a cada 8 celdas es decir a cada *byte*, se le asigna una dirección, que suele ser el tamaño asociado a una variable de tipo *carácter* (por tanto en un *byte* se pueden almacenar 256 tipos de *caracteres*).

La siguiente cuestión es cuántos *bytes* son direccionables. Este hecho depende del procesador y del *sistema operativo*. En lo que se refiere al procesador, los primeros (por ejemplo el *Intel* 8086 en 1979) utilizaban dos *bytes* para el *bus de direcciones*, con lo cual sólo podía direccionar 2^{16} *bits* $=$ 64 *KB* de memoria. En la actualidad se tienen procesadores con 32 *bits* con lo cual pueden direccionar 2^{32} *bits* $=$ 4 *GB* y los últimos ya utilizan direccionamiento de 64 *bits* (2^{64} *bits* $=$ 16 *EB*). Una representación de la disposición de la memoria interna en un ordenador puede verse en la figura 1.4.

Un concepto importante relacionado con el direccionamiento de la memoria en un ordenador es el término ***palabra*** (*de memoria*). Consiste sencillamente, en el tamaño de la información que se graba en cada una de las posiciones especificadas a través del *bus de direcciones*. Esta definición puede ser imprecisa, ya que normalmente la memoria se suele organizar en módulos que actúan en paralelo y pueden direccionarse datos de distinto tamaño o longitud. Por ejemplo, los ordenadores de 32 *bits* (o que utilizan 32 *bits* para direccionar la memoria), el direccionamiento a memoria se realiza por *bytes* y es posible **acceder directamente** a *bytes*, *medias palabras* (16 *bits*) y *palabras* (32 *bits*); incluso permiten acceder a *dobles palabras* (64 *bits*) y *dobles*

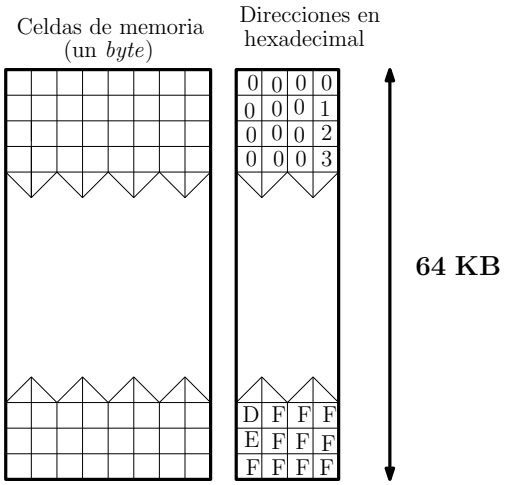


Figura 1.4 – *Mapa de memoria, para un* bus de direcciones *de* 16 bits

palabras grandes (96 *bits*). En este caso, el concepto de *palabra* se asocia a grupos de 32 *bits*, que es el tamaño del *bus de direcciones*.

Otros autores prefieren definir la idea de ***palabra*** como *la mínima unidad de memoria direccionable*. Por tanto, con arreglo a esta propuesta, la *palabra* equivaldría a un *byte*. Para añadir más confusión al término *palabra*, también se emplea para medir el tamaño de los *registros* que utilizan los microprocesadores internamente y que están directamente relacionados con el tamaño del *bus de direcciones* (actualmente son de 32 o 64 *bits*, dependiendo del procesador). Con lo cual, la utilización de este concepto, puede ser confuso si no se tiene claro cuál es la situación que se plantea. En este documento se utilizará el concepto de *palabra* como el número de *bits* que se utilizan para direccionar memoria a través del Sistema Operativo[3].

1.4. Sistemas Operativos

Un ***Sistema Operativo (SO)*** es un programa o conjunto de programas que sirven de intermediario entre el usuario de un ordenador y el conjunto de dispositivos que constituyen una computadora. El *Sistema Operativo* asigna, planifica y distribuye los distintos tareas y recursos de un ordenador. Por *recurso* se entiende el uso de la CPU, la memoria RAM, los dispositivos de almacenamiento secundario, los periféricos conectados al ordenador y la red. Así pues,

- Transfiere programas de cualquier dispositivo de almacenamiento secundario a la memoria RAM y viceversa.
- Ordena a la CPU que ejecute un programa cargado en la RAM.
- Gestiona las porciones de memoria de la RAM, demandados por programas en ejecución.
- Organiza los datos de los dispositivos de almacenamiento secundario (sistema de ficheros) de una manera particular. No lo hacen de la misma forma Linux, Windows o IOS.
- Determina el orden en el que las tareas deben ejecutarse en la CPU y el tiempo que ésta dedica a la ejecución de cada una de ellas. En este caso, un sistema operativo puede ofrecer *multiproceso* o paralelización si la CPU dispone de varios *núcleos*.

[3] Cada vez es más frecuente, que los Sistemas Operativos sean capaces de direccionar 64 *bits*.

- Gestiona la conexión de una posible red de ordenadores, incluso permite que el ordenador se pueda transformar en un *servidor* de otros posibles ordenadores conectados.

En la actualidad es posible encontrar SO de 32 *bits* (cada vez menos) o de 64 *bits*. Si un SO es de 32 *bits*, quiere decir que únicamente puede direccionar memoria hasta 2^{32} *bits* $= 4\,Gb$. Los SO suelen ser programas muy flexibles y aunque un ordenador tenga un procesador que permita direccionar hasta 64 *bits* de memoria, es posible instalar un SO de 32 *bits* y que el ordenador funcione con normalidad. Sin embargo, resulta mucho más eficiente instalar un SO cuyo direccionamiento de memoria se ajuste al direccionamiento de memoria del procesador. Análogamente sucede con las aplicaciones (*procesadores de texto, hojas de cálculo, compiladores, programas de cálculo,...*), no es lo mismo ejecutar un programa que es de 64 *bits* (esto es, es capaz de gestionar direcciones de memoria de 64 *bits*) sobre un SO de 32 *bits*, que en uno de 64 *bits*. Siempre hay que tratar que el tamaño de direccionamiento de memoria de los procesadores, SO y programas coincidan para obtener un mayor rendimiento.

Hay diferentes SO en el mercado. Los más populares son Windows, Linux e IOS, cada uno realiza la gestión entre usuario y computadora de manera distinta por lo general, no compatible entre ellos. En la actualidad todos disponen de versiones de 64 *bits*, con lo cual sobre procesadores capaces de gestionar direcciones de memoria de 64 *bits*, hacen que la computadora permita gestionar y ejecutar de manera eficiente un gran número de potentes programas (siempre que sean a su vez, de 64 *bits*).

LLegado a este punto se puede describir de manera aproximada, el proceso de arranque de una computadora:

1. La CPU se dirige a la ROM y carga la ***BIOS***. Seguidamente realiza un autotest para comprobar todos los componentes principales y la memoria.

2. La ***BIOS*** proporciona información acerca del almacenamiento secundario, la secuencia de arranque y carga en la RAM el SO.

3. El SO reconoce la estructura de ficheros en la memoria secundaria, gestiona la conexión a internet y/o reconoce y direcciona los diferentes dispositivos conectados a la máquina.

4. El ordenador queda listo para ejecutar las órdenes del usuario.

1.5. Compilación

Para que un ordenador ejecute un programa (generalmente identificado como ***programa o código fuente***), es necesario que este código esté *traducido* a un lenguaje que el ordenador sepa entender (llamado ***código máquina***). Existen dos categorías de *traductores*: ***intérpretes*** y ***compiladores***.

Intérpretes. Se trata de un *traductor* que toma el *programa fuente*, almacenado en un ***fichero fuente*** y lo traduce y lo ejecuta, *línea por línea*. Los primeros lenguajes (por ejemplo el BASIC) eran *interpretados*. Generalmente son lentos, debido a que deben volver a traducir cualquier código que se repita, pues una vez traducida al *código máquina* y ejecutada una línea, se olvida. Actualmente la mayoría de los intérpretes modernos, como los del *lenguaje* PYTHON, traducen el programa entero en un *lenguaje intermedio*, que es ejecutable por un *intérprete* mucho más rápido. Tal y como comenta Eckel [1, Cap. 2], los límites entre los compiladores y los intérpretes tienden a ser difusos, especialmente con PYTHON, que tiene muchas de las características y el poder de un *lenguaje compilado*, pero también tiene parte de las ventajas de los *lenguajes interpretados*. Otro ejemplo actual de un *lenguaje interpretado* es MATLAB.

Compiladores. Un *compilador* consiste en un programa que *traduce* los programas almacenados en ***ficheros fuente*** a un lenguaje *máquina*. El resultado final suele ser un fichero que contiene el *código máquina* y

recibe el nombre de ***fichero ejecutable***, que se identifica por la extensión `.exe`. El **lenguaje C** utiliza la *compilación* para traducir los programas a *lenguaje máquina*.

A diferencia de los *intérpretes*, los programas generados por un *compilador* habitualmente requieren mucho menos espacio para ser ejecutados y se realizan mucho más rápido. Adicionalmente, el **C** está diseñado para admitir trozos de programas compilados independientemente. Estas partes se combinan mediante una herramienta llamada ***enlazador*** (*linker*), para generar el *fichero ejecutable*. La forma de realizar la traducción de un *código fuente* mediante un compilador suele constar de varias etapas:

1. se empieza la *compilación* ejecutando un *preprocesador* sobre el *código fuente*. Este *preprocesador* es un programa sencillo que traduce los patrones definidos por el usuario mediante las *directivas del preprocesador* (véase sección 3.5, pág. 56). El código del *preprocesado* se almacena en un fichero intermedio. Seguidamente analiza sintácticamente el código generado en este fichero y en una segunda pasada crea un *código máquina*, que suele referirse como *módulo objeto* y que se almacena en fichero con extensión `.o` u `.obj` que se llama ***fichero objeto***. Puede darse la situación, que el usuario haya generado diferentes funciones en distintos ficheros fuentes. Todos estos programas deben compilarse, dando lugar a diferentes *ficheros objeto*, que habrá que *enlazar*.
2. el *enlazador (linker)* une los distintos *ficheros objeto* generados, en un *programa ejecutable* que el *Sistema Operativo* puede cargar y ejecutar. Cuando una función en un *fichero objeto* hace referencia a una función o variable en otro *fichero objeto*, el *enlazador* resuelve estas referencias; se asegura que todas las funciones y datos solicitados durante el proceso anterior de compilación, están realmente definidas. El *enlazador* puede, así mismo, buscar en las *librerias (bibliotecas)* definidas previamente para resolver todas las posibles referencias. Una vez terminado, se genera el ***fichero ejecutable***, que tiene la extensión `.exe`.

Hay que mencionar que existen muchos ***compiladores de C*** en el mercado. En este curso se utilizará el *compilador* TDM-GCC 4.9.2 que utiliza el entorno de compilación DEV-C++ 5.11 que es de *libre distribución* y que puede descargarse de la dirección `http://orwelldevcpp.blogspot.com.es/`. En este caso, los *ficheros fuente* (aquellos que contienen el programa que se quiere ejecutar) deben tener la extensión `.c`. Automáticamente, este compilador genera el *fichero ejecutable*, con el mismo nombre que el *fichero fuente*, pero con la extensión `.exe`. La generación del *fichero objeto* o los *ficheros objeto* y el proceso de *enlazado* se hace automáticamente y pasa de forma inadvertida para el usuario.

1.6. Programación

La programación es el *procedimiento* por el cual ciertos problemas son transformados de manera que puedan ser resueltos mediante una computadora. Este *procedimiento* puede descomponerse, *grosso modo*, en dos partes: *algoritmo* y *lenguaje de programación*. Para realizar un programa que resuelva un problema mediante un procedimiento computacional es importante que se tenga claro primeramente, cuál es el proceso esquemático que debe seguirse para resolverlo y segundo, la implementación de dicho algoritmo en un lenguaje que sea *entendible* por el ordenador.

Un concepto más abstracto y formal de programa se puede encontrar en Wirth [19]. Expresa que los programas deben estar apoyados en algoritmos eficientes y en un adecuado diseño de datos para poder dar lugar resoluciones óptimas de los problemas planteados. El diseño de los datos capaces de representar adecuadamente las ideas del algoritmo desembocan en las *estructuras de datos*, esto conlleva a considerar *lenguajes* capaces de implementar estas estructuras.

El concepto de algoritmo como proceso para resolver problemas a través de procesos computacionales surge con el trabajo de Turing y Church (véase Gil y Quetglás [6, pág. 41]). A partir de este momento la idea algoritmo se desarrolla de manera más formal en el mundo de los procesos computacionales (véase Knuth [10]).

El *algoritmo* es el primer paso que debe considerarse en la resolución de un problema mediante medios computacionales. Su descripción debe ser independiente del lenguaje que se utilice para su posterior codificación. Mediante instrucciones simples o sentencias se deben establecer los pasos necesarios para llegar correctamente a la solución. Estas sentencias deben estar constituidas por las estructuras de datos abstractas, previamente establecidas. Pueden existir diferentes algoritmos para un mismo problema, como en matemáticas pueden existir diferentes soluciones para un mismo problema. Pero al igual que en matemáticas, siempre se intentará buscar el algoritmo óptimo. No obstante, la idea de *algoritmo óptimo* genera cierta controversia en el mundo de la computación. Puede haber algoritmos expresados de forma compacta y sencillos de entender, pero que su ejecución no sea óptima desde el punto de vista computacional. Por ejemplo, los *algoritmos recursivos* (que se estudiarán más adelante) son ejemplos de algoritmos que permiten resolver ciertos problemas de manera muy compacta, pero que en ciertas ocasiones las versiones *iterativas* de esos mismos problemas, no son tan compactos ni tan claros, pero mucho más eficientes desde el punto de vista computacional.

Siguiendo la idea de Wirth [19], una vez planteado el algoritmo y con unas estructuras de datos determinadas se plantea la construcción del programa. La idea es ajustar de manera *eficiente* las sentencias del algoritmo con las estructuras de datos establecidas.

Durante la segunda mitad de los años 50 y comienzos de los 60 ya comenzaba a desarrollarse de manera significativa los procesos computacionales. Se tenía claro el concepto de *algoritmo*, pero no tanto las ideas sobre las *estructuras de datos* necesarias para implementar estos algoritmos. Así pues, en los primeros lenguajes de programación de *ámbito* popular (COBOL, FORTRAN) las codificaciones resultaban oscuras y complicadas, sobre todo cuando diferentes programadores debían trabajar sobre una misma codificación. Resulta clave la claridad en la implementación del algoritmo. Esto es una realidad para todo aquel que comience en el mundo de la Informática. Es fundamental que el programa que se realice sea claro, no sólo para la persona que lo programa, sino para todo aquel que en cierto momento necesite consultarlo. Así pues, en este sentido para eliminar estos inconvenientes aparecieron nuevas estructuras de datos que a su vez, generó nuevos planteamientos y teorías sobre la metodología de programación.

En este curso se tratará de seguir el modelo de ***programación estructurada***. Es deseable que todo programa que se realice cumpla con los siguientes objetivos (véase Gil y Quetglás [6, pág. 46])

1. Debe ser fácil de leer y de comprender leyendo el propio código.
2. Debe ser fácil de depurar (localizar errores).
3. Debe ser fácil de mantener (ampliar con nuevas especificaciones o modificar las ya existentes).
4. Debe permitir el trabajo en equipo sobre un mismo programa.

La idea de seguir una metodología estructurada es precisamente conseguir los objetivos anteriores. Se trata de plantear programas *jerárquicos* o con metodología *top–down*. Para ello nuestro objetivo será dividir el programa en *subprogramas* más sencillos y que el flujo del programa no implique volver a sentencias anteriores. Por ello comandos del tipo `goto` que existían en las primeras versiones de los lenguajes de programación y permitían desviar el flujo de la ejecución a cualquier parte del código, quedan *desterradas* y prohibidas.

Para implementar esta metodología es necesario disponer de lenguajes que soporten estas especificaciones. Con el desarrollo de los métodos computacionales las versiones de los lenguajes existentes y los que fueron apareciendo se adaptaron a estos requisitos. En la actualidad, prácticamente casi todos los lenguajes se adecúan a esta forma de *programación estructurada*, en particular, el **`lenguaje C`** que es el objeto de este curso de programación.

No obstante, hay que destacar que con el desarrollo de los procedimientos computacionales han surgido nuevos lenguajes capaces de abordar problemas cada vez más globales. Los lenguajes C, FORTRAN, COBOL, PASCAL son ejemplos de lenguajes *procedimentales*. Es decir, cada sentencia o instrucción obliga al compilador que realice alguna tarea: obtén un dato, escribe esta frase, divide dos números, ... En resumen, puede

decirse que un programa en un lenguaje procedimental es un conjunto de instrucciones o sentencias. Esto no es óbice, para que en caso necesario se puedan considerar subprogramas o funciones que simplifiquen el procedimiento de resolución. En cualquier caso, para problemas no muy complejos, estos lenguajes tienen unos principios de organización (llamado *paradigma*) que son eficientes: el programador sólo tiene que crear una lista de instrucciones en este tipo de lenguajes, compilar el programa en el ordenador que ejecutará las órdenes establecidas.

Cuando los problemas se vuelven más complejos y los programas más grandes, la necesidad de modularizar es clara. Es obligado, como ya se ha comentado, definir funciones, procedimientos, subrutinas o subprogramas que simplifiquen la presentación y sirvan para otros programas. La mayor parte de los lenguajes procedimentales actuales se ajustan a esta necesidad (en particular el **C**). No obstante, el principio sigue siendo el mismo: agrupar componentes que ejecutan listas de de instrucciones o sentencias. Esta característica hace que a medida que los programas se hacen más complejos y grandes, el paradigma estructurado comienza a no ser adecuado, resultando difícil terminar los programas de manera eficiente. Existen al menos, dos razones por las que la metodología estructurada se muestra insuficiente para abordar problemas complejos. Primeramente, las funciones tienen acceso limitado a los datos globales y segundo, las funciones inconexas y datos, fundamentos del paradigma procedimental, proporcionan un modelo pobre del mundo real.

En este sentido surgen los lenguajes ***orientado a objetos***, que siguen un paradigma de programación más adecuado para la resolución de problemas complejos más ajustados a la realidad. La idea, entre otras muchas, es diseñar formatos de datos que se correspondan con las características esenciales del problema. Se trata de combinar en una única unidad o módulo, tanto datos como las funciones que operan con estos datos, hablándose de ***objeto***. Las funciones de un *objeto* se llaman ***funciones miembro*** o ***métodos*** y son el único medio para acceder a sus datos. Los datos de un objeto también se conocen como ***atributos*** o ***variables de instancia***. No se puede acceder a los datos directamente. Los datos son ocultos, de modo que están protegidos de alteraciones accidentales. Los datos y funciones se dicen que están *encapsuladas en una única entidad*. El *encapsulamiento de datos* y la *ocultación* de los datos son términos clave en la descripción de los lenguajes orientados a objetos. Otras características son

- ***Abstracción de los tipos de datos. Clases.***
- ***Herencia.***
- ***Polimorfismo.***

El **C++**, **Java** son ejemplos de lenguajes *orientado a objetos*. El **C++** es más flexible porque tiene capacidades *procedimentales* que coinciden con el **C**, por ello se dice que el lenguaje **C** es un subconjunto del **C++**.

CAPÍTULO 2

Primeros programas

2.1. Introducción

En este capítulo se introducen algunas herramientas básicas del **lenguaje C** a través de tres programas. Con ello se pretende proporcionar algunos comandos esenciales que permitan poder desarrollar programas sencillos de manera autónoma. En los siguientes capítulos se irán facilitando nuevos elementos que permitan ir generando programas cada vez más sofisticados.

2.2. Primer programa

Siguiendo el libro de Kernighan y Ritchie [9], es muy sencillo escribir un programa en **C**:

Código 2.1 – *Primer código en C*

```
#include <stdio.h>
/* Este es mi primer programa en C */
int main(void)
{
    printf("Hola mundo ! \n");
    printf("Este es mi primer programa en C \n");

    return 0;
}
```

y cuyo resultado en la pantalla, una vez *compilado* y *ejecutado* el *fichero fuente* (llamado `saludos.c`), es el siguiente:

```
Hola mundo!
Este es mi primer programa en C

--------------------------------------
Process exited after 0.4095 seconds with return value 0
Presione una tecla para continuar . . .
```

Veamos cada una de las sentencias de este código.

Línea 1. Esta primera sentencia está diciendo que se debe incluir el fichero `stdio.h`, que contiene unas definiciones de funciones de entrada salida (en particular `printf()`), de ahí su nombre: **st***andar***d i***nput* **o***utput* **h***eader*. El símbolo `#` la identifica como línea a ser manipulada por el *preprocesador* **C**. Tal y como se infiere por el nombre, el *preprocesador* realiza algunas tareas antes de comenzar la compilación.

Línea 2. Se trata de un comentario. Todo aquello que esté entre `/*` y `*/` lo ignora el compilador. El texto entre estos dos símbolos puede ocupar, incluso varias líneas. Los comentarios son notas que se introducen para hacer más claro el programa.

Línea 3. Indica que `main` es una función de *tipo entero* (ya se estudiará más adelante qué quiere decir esto) y que no tiene argumentos, por eso aparecen dentro del paréntesis: `(void)`. Los programas en **C** se suelen componer de una o más funciones. Esta declaración expresa que se trata de la función *principal*. Todo programa en **C** comienza su ejecución siempre con la instrucción que lleva el nombre de `main`. Todas las demás funciones podrán llevar el nombre que elijamos nosotros, pero la existencia de la función `main` es **obligatoria**. Todo aquello que queda antes de la función `main` se le denomina ***cabecera*** del programa.

Línea 4. La llave `{` indica que comienza la definición de la función (en este caso, de la función `main`). La definición acaba en la LÍNEA 9 con el carácter `}`.

Línea 5. Se trata de otra función. En este caso de *salida* de información, que por defecto interpreta que es la pantalla del ordenador. Imprime la frase comprendida entre comillas, que es lo que constituye su argumento. Dentro de las comillas está el símbolo `\n` que simplemente es la instrucción de: *comienza una nueva línea ajustándote al margen izquierdo*. Forma parte de los comandos denominados *secuencias de escape*.

Un detalle fundamental es el punto y coma `;` al final de la instrucción. Es la manera de indicar al **C** que el instrucción ha terminado. Es un símbolo que es parte de la sentencia y no simplemente un separador, como puede ocurrir en otros lenguajes. Por tanto, su utilización al final de cada *sentencia ejecutable* es **obligatoria**.

Línea 6. Otra sentencia ejecutable para que imprima en la pantalla.

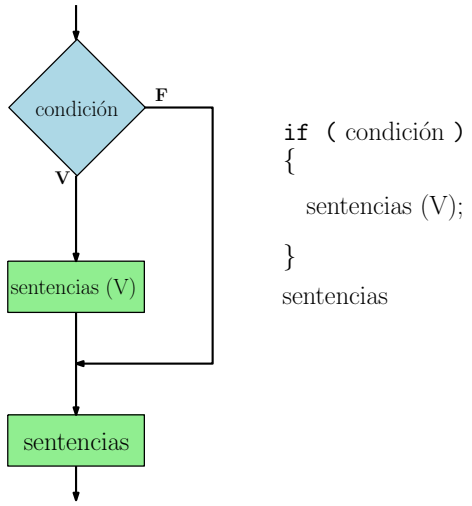
Línea 8. Le indica al compilador que ha terminado la descripción de la función `main` y por tanto, debe terminar la ejecución del programa devolviendo un valor 0, como resultado de una terminación *normal*. Este valor aparece después, en la pantalla de salida de ejecución del programa cuando pone: [...] `with return value 0`. No obstante, en este documento no se prestará mucha atención al resultado de la ejecución que aparece después de la línea discontinua y por tanto **NO** se mostrará en los resultados de ejecución de los programas que se realicen.

2.3. Segundo programa

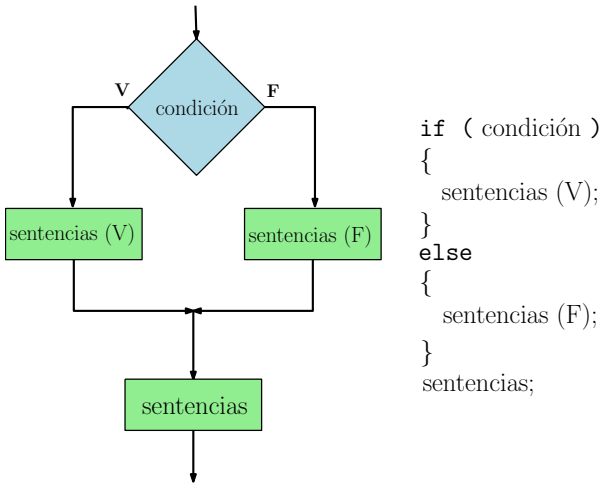
En el siguiente código se pretende aumentar un poco más la complejidad. Para ello se introduce el concepto de variable entera y la utilización de operaciones aritméticas elementales. Además, se presenta la sentencia de asignación `=` y las sentencias de control: **`if`** e **`if-else`**, que permiten cambiar el flujo del programa (véase figura 2.1).

Con este objetivo, se considera un programa que lee 3 números enteros del teclado y que después obtiene:

1. la suma,
2. el producto,



(a) **Bloque `if`**



(b) **Bloque `if-else`**

Figura 2.1 – *Bloques `if` y `if-else`*

3. el promedio. Para ello, la suma se divide por 3. Si este resultado no es entero se calcula el cociente entero y su resto. Por último
4. una disposición ordenada de los números introducidos, de menor a mayor.

Una posible solución se muestra en el código 2.2.

Código 2.2 – *Segundo código en C*

```c
#include <stdio.h>
/* Segundo  programa en  C */
int main(void)
{
    // Declaración de variables
    int entero1, entero2, entero3;
    int suma, producto, promedio, resto;
    int aux = 100;

    suma = producto = promedio = resto = aux; //Asignación múltiple

    //Entrada de datos
    printf("Introduce tres número enteros -> ");
    scanf("%i  %i  %i", &entero1, &entero2, &entero3);
    printf("\n \n Los números introducidos son: %i,  %i,  %i \n\n\n", entero1,
       entero2, entero3);
    printf("\n \n suma = %i, producto = %i, ", suma, producto);
    printf("promedio = %i y resto = %i\n\n\n",  promedio, resto);

    /* Algoritmo para
                la resolución del problema */
    suma = entero1 + entero2 + entero3;
    printf("La suma de los números enteros es  %i \n", suma);

    producto = entero1 * entero2 * entero3;
    printf("El producto de los números enteros es  %i \n", producto);

```

```
27      promedio = suma/3;
28      resto = suma % 3;
29      if( resto != 0 ){
30        printf("El promedio es %i y su resto %i\n", promedio, resto);
31      } else {
32        printf("El promedio es %i \n", promedio);
33      }
34
35      if(entero1 > entero2){
36         aux = entero1;
37         entero1 = entero2;
38         entero2 = aux;
39      }
40      if(entero2 > entero3){
41         aux = entero2;
42         entero2 = entero3;
43         entero3 = aux;
44      }
45      if(entero1 > entero2){
46         aux = entero1;
47         entero1 = entero2;
48         entero2 = aux;
49      }
50
51      printf("\n Los números enteros ordenados son: %i, %i, %i\n\n", entero1,
           entero2, entero3);
52
53      return 0;
54  }
```

El resultado en pantalla si se introducen los números enteros: 17 -4 9 sería:

```
Introduce tres números enteros -> 17 -4 9

 Los números introducidos son: 17, -4, 9


 suma = 100, producto = 100, promedio = 100 y resto = 100

La suma de los números enteros es 22
El producto de los números enteros es -612
El promedio es 7 y su resto 1

 Los números enteros ordenados son: -4, 9, 17
```

2.3.1. Análisis del programa

Línea 5. Es otra forma de introducir un comentario en el programa. Todo lo que haya a la derecha de `//` y en la *misma línea*, el lenguaje **C** no lo procesa. Otros ejemplos pueden observarse en las LÍNEAS 12 y 10. En este último caso, se sitúa después de una sentencia ejecutable.

Líneas 6-8. Antes de comenzar es importante explicar qué son las ***variables*** desde un punto de vista informático. Una primera definición sería: *una porción de la memoria dinámica en la que se puede almacenar un dato, cuyo valor podría cambiar durante la ejecución del programa*. El hecho de poner **`int`** delante de cada sentencia, indica que, por ejemplo en la LÍNEA 6: `entero1, entero2, entero3` serán

tres porciones de memoria que almacenarán *valores enteros* en un tamaño de 4 *bytes*[1]. Esta es la forma que tiene **C** para expresar qué tipo de *variable* se va a utilizar y cuánto espacio de memoria se utilizará. Se llama sentencia ***declarativa*** o simplemente ***declaración*** de variables (en este caso *enteras*). En **C** es ***obligatorio*** *declarar* **todas** las variables que se vayan a utilizar en el programa.

En la LÍNEA 8, también se ha inicializado el valor de la variable `aux` con el valor de 100. Es lo que se llama una *declaración* con *inicialización* de la variable. En la inicialización se utiliza la sentencia de asignación `=`, para introducir el valor de 100 en la posición de memoria identificada por `aux`. Se trata de un *operador* que asigna el valor de su derecha, a la variable que está a su izquierda. Por tanto, aunque por su representación pudiera parecerlo, el símbolo de asignación `=`, no tiene nada que ver con la igualdad matemática.

Línea 10. Se trata de una sentencia de asignación múltiple, en la que se introduce el valor que contiene `aux` a las distintas variables que están a su izquierda. Así pues, a las variables: `suma, producto, promedio` y `resto` se les asigna el valor 100 que es lo que contenía la variable `aux`. Como ya se ha dicho, a continuación de la sentencia se ha puesto un comentario, mediante el símbolo `//`.

Línea 14. El comando **`scanf`** permite leer los tres valores enteros desde el teclado. Su argumento (aquello que está entre paréntesis) se divide en dos partes: *cadena de control* y *lista de argumentos*. La cadena de control está formado por *especificaciones de formato* que se ponen entre comillas (`" "`). En este caso: `%i %i %i`, indican que se van a introducir tres números enteros[2]. La lista de argumentos coincide con el nombre de las respectivas variables, separadas por comas y precedido cada una de ellas, por el símbolo `&`. Esto indica a la función **`scanf`** la *dirección* de las respectivas variables. Este asunto se desarrollará posteriormente (véase subsección 3.2.3, pág. 38).

Líneas 15-17. Se genera un *eco* de los datos de entrada. El motivo es comprobar que la lectura de los datos introducidos se ha realizado correctamente. La función **`printf`**, al igual que sucedía con el comando **`scanf`**, se compone de una *cadena de control* y una *lista de argumentos*. En la cadena de control se pone entre comillas (`" "`) el texto que aparecerá en la pantalla, con los especificadores de formato necesarios, correspondientes al tipo de variables que constituyen la lista de argumentos, cuyas variables, de nuevo, deben estar separadas por comas. En el texto del ejemplo, se introducen, de nuevo, los especificadores `%i` debido a que en la lista de argumentos, sólo hay variables enteras. En este caso, únicamente se especifica el nombre de la variable **sin** estar precedido del símbolo `&`.

Líneas 21, 24, 27 y 28. En estas líneas se realizan operaciones aritméticas elementales *enteras*. Aunque la descripción de todas ellas es obvia se tratarán más adelante (véase subsección 3.4.2, pág. 48). Sin embargo, merece la pena resaltar la palabra *entera*. Hace referencia a que los datos utilizados para operar son *enteros* y por tanto, el resultado también será también *entero*. Así pues, en la división

```
promedio = suma/3;
```

puesto que `suma` es una variable entera y 3 es una constante entera, la operación se realiza con enteros y el resultado es por tanto, entero. Esto quiere decir que si `suma` no almacena un múltiplo de 3, entonces el resultado que se almacene en la variable `promedio` será únicamente, la parte entera del resultado. Por último, señalar que la operación

```
resto = suma % 3;
```

[1] es el tamaño por *defecto* para este tipo de compilador. Más adelante en el documento se aprenderá a modificar este tamaño de *bytes*.

[2] También es posible utilizar el formato `%d` para valores enteros. Este formato era obligatorio para los valores enteros en versiones clásicas de **C**. Sin embargo, resultaba poco *intuitivo*, por ello ahora es posible utilizar el formato `%i`, para el *tipo* **`int`**.

hace referencia al resto de la división entera.

Líneas 29-33. Se trata de un bloque **`if-else`** que permite cambiar el flujo del programa, de acuerdo con el resultado de la *condición*, que se encuentra entre los paréntesis que siguen a la instrucción **`if`**. En este caso, se pregunta si `resto` es **distinto** a 0 (`resto` $\neq$ 0), que en **C** se escribe como

```
resto != 0
```

En caso de ser ***cierta***, se ejecutan todas las sentencias del *bloque* **`if`**, esto es, todas las sentencias que están a continuación del comando **`if`**, que están entre llaves. En este programa, sólo existe una sentencia:

```
printf("El promedio es %i y su resto %i\n", promedio, resto);
```

Una vez ejecutada, el flujo del programa sigue con la siguiente sentencia ejecutable, que se encuentra a continuación de la llave que cierra el *bloque* **`else`**. En el ejemplo, iría a la LÍNEA 35 del programa. Si la evaluación de la *condición* es ***falsa***, entonces el flujo del programa se *salta* el *bloque* `if` y realiza el *bloque* **`else`**. Esto quiere decir, que ejecuta todas las sentencias que están a continuación de la instrucción **`else`** y están entre llaves. En este ejemplo, sólo se ejecutaría la sentencia:

```
printf("El promedio es %i \n", promedio);
```

Una vez terminado, el flujo del programa sigue en la sentencia que se encuentra después de la llave que cierra el *bloque* **`else`**. En el programa anterior, sería la LÍNEA 35.

Para construir las expresiones que constituyen las *condiciones*, que deben ser evaluadas para determinar la dirección del flujo del programa, se suelen construir mediante los *operadores de relación* y los *operadores lógicos*. Para conocer distintos tipos de estos operadores y sus propiedades, puede consultarse las subsecciones 3.4.5 (pág. 54) y 3.4.6 (pág. 55), respectivamente.

Líneas 35-39. Se tratan de un *bloque* **`if`**. Su descripción es similar al caso anterior, del *bloque* **`if-else`**. Se evalúa la *condición*, que está a continuación de **`if`** y entre paréntesis. Si es ***cierta***, entonces se ejecutan todas las sentencias que están entre las llaves. En este programa, en caso de ser *cierta* la *condición*, se ejecutarían las sentencias:

```
aux = entero1;
entero1 = entero2;
entero2 = aux;
```

Una vez terminado, el flujo del programa continúa con la siguiente sentencia después de la última llave. En este caso, seguiría en la LÍNEA 40. Si la *condición* es ***falsa***, entonces el flujo del programa, se salta el *bloque* de instrucciones y continúa con la siguiente sentencia ejecutable que sigue a la última llave del *bloque*. En este ejemplo, la LÍNEA 45.

Básicamente estos algoritmos describen un intercambio de valores entre dos variables, que puede verse con más detalle en la sección 3.2 (pág. 25).

Líneas 40-44, 45-49. De nuevo se tratan de *bloques* **`if`** y su interpretación es similar al caso anterior ya descrito.

Línea 51. Se muestra la salida del programa en pantalla.

2.4. Tercer programa

En este apartado se trata de ilustrar la utilización de una sentencia de *repetición*. Se trata de la instrucción **`while`** (véase figura 2.2), que permite repetir un conjunto de sentencias, siempre que se cumpla cierta *condición* (de manera similar al caso de **`if`**).

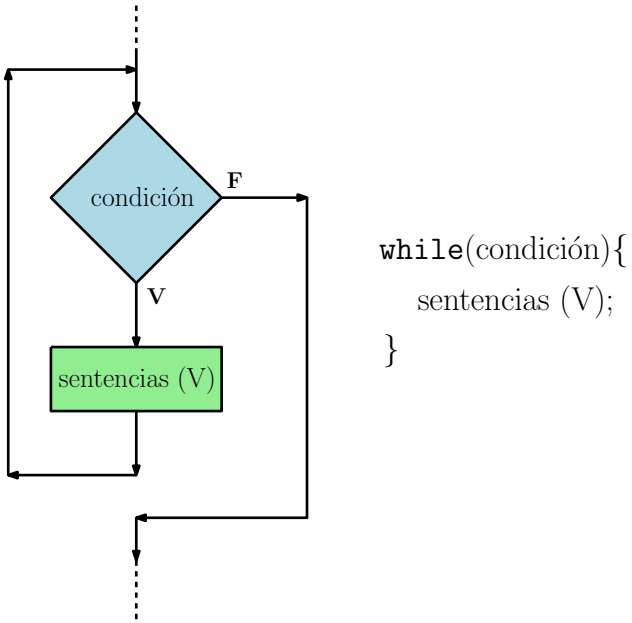


Figura 2.2 – *Bucle* ***while***

El objetivo es realizar un programa que sume los n primeros números naturales, siendo n un valor introducido por el usuario. Una vez sumados, el programa debe comprobar que se cumple la igualdad

$$1 + 2 + 3 + \cdots + n = \frac{n \cdot (n+1)}{2}$$

Una solución al ejercicio planteado se describe en el código 2.3, a continuación

Código 2.3 – *Tercer código en* ***C***

```c
#include <stdio.h>
/* Otro  programa en  C */
int main(void){

    //Declaración de variables
    int n;
    int cont, suma = 0, formula = 0;

    //Entrada de datos
    printf("Introduce un número entero positivo -> ");
    scanf("%i", &n);
    printf("\n\n");

    //Algoritmo
    cont = 1;
    while (cont <= n){
```

```
17          suma = suma + cont;
18          cont = cont + 1;
19      }
20       formula = n * (n+1) / 2;
21       if(formula == suma){
22           printf("La suma de los %i primeros número es correcta y es %i\n\n", n,
                 suma);
23       } else {
24           printf("La suma de los %i primeros números es %i pero NO es correcta\n\n
                 ", n, suma);
25       }
26       return 0;
27  }
```

La solución que aparece en la pantalla si se introduce el número $n = 20$ es

```
Introduce un número entero positivo -> 20

La suma de los 20 primeros números es correcta y es 210
```

2.4.1. Análisis del programa

Líneas 15-19. En esta parte del código se muestra la instrucción **`while`**. En la LÍNEA 15 se inicializa la variable entera `cont` a 1. Con este valor el flujo del programa entra en el *bloque* **`while`**. Lo primero que se hace es evaluar la *condición* que se encuentra entre paréntesis. En este caso, como `cont = 1` y el valor de `n = 20` (este valor se ha introducido previamente por el teclado), la condición es ***cierta*** y se ejecutan las sentencias que se encuentran dentro del *bloque*, es decir las que están entre las dos llaves. Una vez que el flujo del programa encuentra la última llave, vuelve a la instrucción **`while`** (LÍNEA 16) y repite la evaluación de la *condición*. Si es ***cierta***, de nuevo se ejecutan las instrucciones del *bloque* y así sucesivamente, hasta que la evaluación de la *condición* sea ***falsa***. En este caso el flujo del programa se desplaza hasta la siguiente sentencia después de la última llave: en el código anterior, la LÍNEA 20.

Líneas 21-25. Se trata de un *bloque* **`if-else`**, similar al ya comentado anteriormente. Sin embargo, en la *condición* aparece el *operador de relación* `==`. Esta instrucción comprueba si los valores que están en sus extremos son iguales o no. Es importante no confundir este operador con el de *asignación*: `=`, que como ya se ha comentado, introduce el valor que está a su derecha, en la variable que se encuentra a su izquierda.

El resto de las sentencias son muy similares a las ya estudiadas en los programas anteriores y no se vuelven a describir.

2.5. Problemas

Todos ejercicios que se plantean pueden resolverse a partir de los tres programas analizados en este capítulo, utilizándolos como modelo.

1. Identifica y corrige los errores si existen, en los siguientes apartados

a) `printf("El valor de la variable es %d \\n", &entero)`

b) `scanf("%i %i" &valor1 valor2);`

c) Con el siguiente trozo de programa, la salida que se obtiene es "`El valor de entero es 12`"

```
entero = 12;
if(entero = 7){
    printf("El valor de entero es 7);
} else {
    printf("El valor de entero es 12);
}
```

d) Si se sustituye el *bloque* **`while`**

```
cont = 1;
while (cont <= n){
   cont = cont + 1;
   suma = suma + cont;
}
```

por el existente en el del código 2.3, se obtiene el mismo resultado.

2. Dada la sentencia

```
resultado = a * b * c;
```

Escribe un programa que pida al usuario tres números enteros, utilice la sentencia anterior y escriba por pantalla el resultado.

3. Escribe un programa que pida un número entero y determine si es par o impar. Una ejecución típica debería ser:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 12345
**************************

Introduce un número entero -> -243
El número -243 es IMPAR
```

Ahora haz otro programa que determine si el número entero es múltiplo de tres. Si no lo es, que determine su resto. Una ejecución típica del programa debe ser:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 12345
**************************

Introduce un número entero -> 25
El número 25 NO es un múltiplo de 3. Su resto es 1.
```

4. Siguiendo los problemas anteriores, ahora se pide que escribas un programa que pida un número entero y a continuación otro entero. Que compruebe si éste último es un divisor del primero. En caso contrario que determine el resto, de la división entera. Una ejecución típica del programa debe ser:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 12345
**************************

Introduce un número entero -> 25
Introduce un segundo entero (posible divisor del anterior) -> 7

 El número 25 NO es un múltiplo de 7. Su resto es 4.
```

5. Escribe un programa que intercambie los respectivos valores que contengan dos variables. Esto es, si la variable $a = -12$ y la variable $b = 7$, entonces el código tiene que ser capaz de obtener la salida: $a = 7$ y $b = -12$. Una posible ejecución podría ser:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 12345
**************************

Introduce un valor para la primera variable a = 27
Introduce un valor para la segunda variable b = -53

Después del intercambio:

El valor de la variable a = -53 y el de la variable b = 27
```

6. Describe qué hace el siguiente *bloque* **`while`**:

```
var1 = 1;
var3 = 1;
var2 = 25;
while (var1 < var2){
   var1 = var1 + 1;
   var3 = var3 * var1;
}
```

7. Escribe un programa que calcule el factorial de número natural, dado. Una posible ejecución sería:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 12345
**************************

Introduce el número natural -> 7

 El factorial de 7 es 7! = 5040
```

8. Escribe un programa que calcule la potencia entera positiva de un número entero. Esto es, si $a \in \mathbb{Z}$ es la base y $b \in \mathbb{N}$ el exponente, se pide a^b. Una posible ejecución sería:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 12345
**************************

Introduce un número entero (base de la potencia) -> -2
Introduce ahora el exponente (número natural) -> 5

 La potencia 5 de -2 es -32
```

9. Escribe un programa que sume los dígitos de un número natural. Esto es, dado el número 7435 la suma de sus dígitos es 19. La idea es ir obteniendo el resto de la división entera por 10. Por ejemplo, siguiendo con el número anterior: $7435 \,\%\, 10 = 5$ y $7435/10 = 743$. Seguidamente se hace el mismo proceso pero ahora con 743. Es decir: $743 \,\%\, 10 = 3$, que será otro dígito que se suma al anterior y se opera: $743/10 = 74$. A continuación se sigue una pauta análoga con 74 y así sucesivamente, hasta que la parte entera sea cero. Una posible ejecución del programa sería:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 12345
**************************

Introduce un número natural -> 45207

La suma de los dígitos de 45207 es 18
```

10. A partir del programa anterior, escribe un código que escriba el reverso de un número natural. Esto es, si se introduce el número 45178, la salida debe ser 87154. A continuación que compruebe si es *capicúa* o no. Una posible ejecución del programa sería

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 12345
**************************

Introduce un número natural -> 45207

El reverso de 45207 es 70254

En número 45207 NO es capicúa
```

11. Escribe un programa que determine si un número es *perfecto*. Se dice que un número natural es *perfecto* si su valor coincide con la suma de sus divisores. Por ejemplo, el número 6 es *perfecto* ya que $6 = 1+2+3$. También lo son 28 y 496. La idea del algoritmo es muy sencilla: basta con tomar todos los números menores que el introducido y comprobar si alguno es divisor. Si lo es, entonces se suma con los anteriores

que han sido divisores. Al final la suma total y el número natural inicial deben ser iguales, si el número es *perfecto*.

Utilizando el código anterior, haz un programa que dado un número natural, saque por pantalla todos los *números perfectos* menores que él.

12. Haz un programa que calcule el *Máximo común divisor (m.c.d.)* y el *Mínimo común múltiplo (m.c.m.)* de dos números naturales.

NOTA:

Para calcular el *máximo común divisor (m.c.d.)* de dos números enteros positivos, puede aplicarse el *Algoritmo de Euclides*. Este proceso se basa en que dados dos números D (*dividendo*) y d (*divisor*) con $D > d$, la división entre ambos números se expresa de la forma:

$$D = C \cdot d + r$$

donde C es el *cociente* y r el *resto*. Si k es un divisor de D y d, entonces estos valores serán de la forma: $D = T \cdot k$ y $d = s \cdot k$ y como consecuencia, k también debe ser un divisor de r, ya que:

$$T \cdot k = C \cdot s \cdot k + r \iff k \cdot (T - C \cdot s) = r$$

Teniendo en cuenta este hecho, se hace a continuación, la división entre d y r, pero ahora d como *dividendo* y r como *divisor*, obteniéndose:

$$d = C_2 \cdot r + r_2$$

Si $r_2 \neq 0$, entonces de nuevo, si k es divisor de d y r, también lo será de r_2. Con lo cual se vuelve a repetir el mismo procedimiento, dividiendo r entre r_2 y así sucesivamente hasta que el resto de las sucesivas divisiones sea cero, para obtener el *m.c.d.*

Basándonos en este hecho se plantea un proceso iterativo, llamado el *Algoritmo de Euclides*, cuyo desarrollo se presenta en el siguiente seudocódigo:

Seudocódigo: Algoritmo de Euclides

```
Dados a y b (a > b)
Calcula el m.c.d.(a,b)

D <- a; d <- b;
Mientras r != 0 haz lo siguiente
    r <- mod(D,d)  //resto de la división entera
    D <- d
    d <- r
fin
el resultado es: D
```

Como ejemplo y antes de resolver el problema aplica este algoritmo, para determinar el *m.c.d.*(35, 25).

CAPÍTULO 3

Introducción al lenguaje C

3.1. Introducción

En este capítulo se introducen nuevas herramientas y conceptos básicos del **lenguaje C**, que permitan desarrollar unos programas más completos y eficientes.

3.2. Datos

En un programa se gestionan dos modelos de datos: *constantes* y *variables*. Ambos se caracterizan por almacenar valores en la memoria de un ordenador. La diferencia consiste en que el valor de las *constantes* no puede modificarse durante la ejecución de programa y el de las *variables* sí. Así pues, como ya se comentó en el capítulo anterior (véase subsección 2.3.1, pág. 16), una variable en Informática se identifica por una posición (*dirección*) en la memoria, que puede tomar valores diferentes a lo largo de la ejecución de un programa. El hecho que pueda tomar diferentes valores hace que se asemeje al concepto de variable en Matemáticas. Pero la cosa cambia, cuando por ejemplo, se quieren intercambiar los valores de dos variables. Mientras que matemáticamente el proceso es inmediato, desde un punto de vista informático, ya no lo es. Esto es debido a que las variables ocupan unas posiciones de memoria (*direcciones*) determinadas y para poder modificar su contenido es necesario un proceso de *asignación de valores*. Como consecuencia, si se quiere intercambiar el valor de dos variables, es necesario involucrar una tercera variable auxiliar (véase figura 3.1) Esta idea ya se utilizó en el capítulo anterior (véase el código 2.2, pág. 15). A la vista de este ejemplo, es claro que la *variable informática*, tiene cierta similitud con la *variable matemática*, pero no es lo mismo.

Es básico insistir en que las variables en Informática ocupan un espacio físico en la memoria, identificado por una *dirección* (que suele ser un número). Para que el programa pueda realizar de manera eficaz el acceso a los valores de una variable, la memoria se organiza en grupos de *bytes* que son *direccionables* (esto es, cada *byte* tiene un número), tal y como se- especificó en la sección 1.3 (pág. 6). Con este planteamiento, para utilizar una variable en **C** en un programa, es necesario

1. hacer saber al programa que quieres utilizar una porción de su memoria para almacenar un valor (en principio nos da igual en qué dirección de memoria) y además indicar qué ***tipo*** de valor vas a utilizar: si es ***tipo*** *carácter*, de ***tipo*** *entero* o *real*[1] (hay algunos *tipos* de variables un poco más sofisticados, pero se

[1] Hay que advertir, que en Informática los *números reales*, coinciden más con la idea de *números racionales* ($\mathbb{Q}$) estudiados en Matemáticas. En Informática no existen los *números reales* ($\mathbb{R}$) tal y como se entienden en Matemáticas, aunque se hable de ellos.

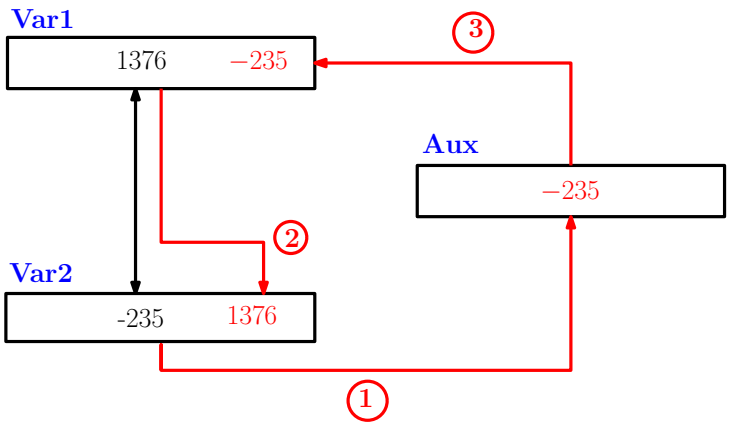


Figura 3.1 – *Intercambio de los valores de dos variables. En **negro** se muestra la idea de intercambio, en **rojo** el procedimiento de intercambio de variables en Informática: en los círculos se determina el orden de las etapas.*

verán más adelante)

2. así mismo, se le puede comunicar que nos interesa introducir un valor en esa dirección de memoria, que ha reservado anteriormente (esto no es imprescindible, pero es una buena costumbre a la hora de programar).

Estos dos pasos se llaman ***declaración*** e ***inicialización*** de variables, respectivamente. Veamos cada uno de ellos más detenidamente

Declaración. Este paso es ***obligatorio*** en **C** y se le dice al ordenador, *qué tipo* de variable se va a emplear y *cuánto* espacio de memoria se necesita, junto con la manera de identificarla. El *tipo* y el *espacio* se realiza en una misma etapa, de esta forma para realizar una sentencia de *declaración* se utilizan las palabras claves que se muestran en el cuadro 3.1. Para identificar la variable se utiliza un nombre *alfanumérico* (con alguna restricción que se verá más adelante). A continuación el programa asocia una dirección de memoria a ese *nombre* elegido por el usuario.

Ejemplos de sentencias de declaración

```
char carac, ter;
int entero;
unsigned int entero_sin;
float flotador, radio;
double real_largo;
```

Los tamaños (el número de *bytes*) asignado a cada *tipo* de variable, depende del modelo del compilador, o si funciona bajo 32 o 64 *bits*. Para poder determinar el tamaño para cada caso, el **C** ofrece la función **sizeof**, que da el tamaño en *bytes* del *tipo* de variable. De esta manera, para obtener el cuadro 3.1 se ha hecho un programa con sentencias parecidas a:

```
int Tam_ent, Tam_car, Tam_float, Tam_double;
..........
..........
Tam_ent = sizeof(int);
Tam_car = sizeof(char);
```

Tipos	Declaración	Tamaño	Valor mínimo	Valor máximo
Caracteres	`char`	1 *byte*	−128	127
	`unsigned char`	1 *byte*	0	255
Enteros	`short int`	2 *bytes*	-32.768	32.767
	`int`	4 *bytes*	-2.147.483.648	2.147.483.647
	`long int`	4 *bytes*	-2.147.483.648	2.147.483.647
	`long long int`	8 *bytes*	-9.223.372.036.854.775.808	9.223.372.036.854.775.807
	`unsigned int`	4 *bytes*	0	4.294.967.295
Reales	`float`	4 *bytes*	[±] 1.17e–38	[±] 3.4e38
	`double`	8 *bytes*	[±] 2.225e–308	[±] 1.797e308

Cuadro 3.1 – *Algunos tipos básicos de variables en* **C**. *Los tamaños que se muestran se han realizado con el compilador Dev–C++ Ver. 5.11*

```
Tam_float = sizeof(float);
Tam_double = sizeof(double);
```

Existe la posibilidad de declarar una variable como **`long double`**, aunque ANSI C no garantiza el *rango* y *precisión* mayores que las de **`double`**. Generalmente la posibilidad de utilización depende del compilador y del tipo de procesador (sea 32 o 64 *bits*). La declaración de estas variables es similar a los casos anteriores

```
long double real_enorme
```

El compilador DEV-C++ asigna un tamaño de 12 o 16 *bytes* a este tipo de variables, según se trabaje en 32 o 64 *bits*, respectivamente. En el presente curso, no se utilizará este tipo de variables.

Inicialización. Este paso es opcional, aunque muy conveniente a la hora de programar. La *inicialización* de una variable se realiza mediante una orden llamada ***asignación***. Simplemente consiste en que se *asigna o deposita* en el espacio de memoria que tiene la variable, un valor. En **C** este proceso se realiza mediante el símbolo de *igualdad*: [=]. Esta notación es un tanto confusa, porque no tiene nada que ver con el concepto de igualdad en Matemáticas. De hecho, hay lenguajes utilizan otra notación más *gráfica* para distinguirlo, por ejemplo: '<−' o '**:**='. En cualquier caso, en **C** quiere decir que el valor que hay a la derecha de [=], se *asigna* a la variable que hay a la izquierda. Esto lleva a que tenga perfecto sentido, sentencias del tipo:

```
Contador = Contador + 1;
```

Simplemente indica que se sume una unidad al valor que tiene almacenada la variable **`Contador`**, y el resultado se coloque en la misma posición de memoria, obteniéndose el aumento en una unidad, el valor de la variable **`Contador`**.

Seguidamente se muestran unos ejemplos de *inicialización y declaración* de variables

Ejemplos de sentencias de declaración e inicialización

```
char carac, ter;
int entero = 15;

float flotador = 8.123, area;
double real_doble = 12234.1234;

const double Pi = 3.14159265358979323846264338327795; /* Constante */

area = 15.734562;
carac = 'a';
ter = 70;  /* Código ASCII de la letra F */
```

Veamos el resultado de algunas de estas instrucciones en un sencillo programa

Código 3.1 – *Ejemplo de programa con las sentencia de asignación e inicialización*

```
#include <stdio.h>
/* Salida de algunos tipos de variables y constantes */
int main(void)
{
        char carac, ter;
        int entero = 15;
        float flotador = 8.123, area;
        float real = 345.1234567;
        double realDoble = 345.1234567;

        const double Pi = 3.14159265358979323846264338327795; /* Constante */

        area = 15.734562;
        carac = 'a';
        ter = 70;  /* Código ASCII de la letra F */


        printf("El valor de carac es -> %c. \n", carac);
        printf("El valor de ter es -> %c. \n", ter);
        printf("El valor de entero es -> %d. \n", entero);
        printf("El valor de flotador es -> %f \n", flotador);
        printf("El valor de area es %f -> \n", area);
        printf("El valor de Pi es -> %f \n", Pi);
        printf("El valor de Pi es -> %.20f \n", Pi);
        printf("El valor de real es -> %.8f \n", real);
        printf("El valor de realDoble es -> %.8f \n", realDoble);

        return 0;
}
```

y cuya salida en la pantalla sería la siguiente

```
El valor de carac es -> a
El valor de ter es -> F
El valor de entero es -> 15
El valor de flotador es -> 8.123000
El valor de area es -> 15.734562
```

```
El valor de Pi es -> 3.141593
El valor de Pi es -> 3.14159265358979310000
El valor de real es -> 345.12344360
El valor de realDoble es -> 345.12345670
```

En la LÍNEA 11 se ha utilizado el ***cualificador*** **`const`**. Esto implica que esta variable no puede cambiar de valor durante la ejecución del programa. De hecho, se genera un error de compilación[2] si se detecta un comando del tipo

```
Pi = Pi + 1;
```

Por tanto **`Pi`** constituye una ***constante simbólica*** para **C**.

3.2.1. Tipos

En este apartado se pretende analizar más detenidamente los *tipos* básicos que ofrece el **C**. En la declaración de *tipos*, como ya se ha comentado, se establece de manera implícita, el tamaño de *bytes* que ocupará cada variable en la memoria.

Tipo `char`

Aunque por su denominación y uso hace referencia a caracteres (véase el código 3.1), la realidad es que **C** los trata como variables enteras de 8 *bits* (como puede comprobarse mediante el comando **`sizeof(char)`**, anteriormente comentado). El hecho de que se pueda manipular caracteres con este tipo de variables, no quiere decir que almacene los caracteres *físicamente*. Realmente almacena un número, que corresponde al código ASCII[3] de ese carácter. Con 8 *bits*, una variable del tipo **`char`** tiene capacidad para almacenar 256 caracteres ($2^8 = 256$). En el programa 3.1 se utiliza este tipo de declaración:

Línea 5. Se declaran `carac` y `ter`, como variables carácter.

Líneas 14-15. Se inicializan las variables. Mientras que para la variable `carac` se realiza explícitamente con el carácter `a`, para la variable `ter` se utiliza el código ASCII, de la letra `F`.

Líneas 18-19. Se imprime en pantalla los caracteres `a` y `F` que son los valores que contienen las variables `carac` y `ter`, respectivamente. Como se observa se utiliza el *especificador de formato*: `%c` para imprimir los caracteres. Se podía haber utilizado un especificador de formato entero: `%i` o `%d`. En este caso, el resultado hubiese sido el código ASCII del carácter.

En **C** existe otro tipo de variable carácter. Se declara como

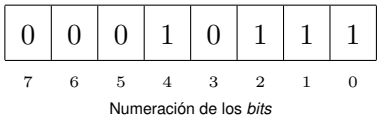
```
unsigned char car_sinsigno;
```

Se trata de una variable carácter *sin signo*. Pero ¿qué quiere decir esto?. Como ya se ha comentado, el lenguaje **C** trata a las variables carácter como variables enteras de 8 *bits*. Que sea una variable *carácter sin signo* significa que almacena números enteros del 0 al 255 y por supuesto que estos valores los asigna a signos ASCII. Sin embargo, una variable declarada como **`char`** se asocia a una variable entera con signo. Esto quiere decir que su rango va desde -128 a 127, pero también asigna *unívocamente* todos estos valores a los signos del código ASCII. A la vista de esta situación, veamos cómo realmente se representan este tipo de variables internamente:

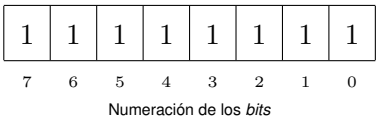
[2] Con el compilador DEV–C++

[3] Es el acrónimo de *American Standard Code for Information Interchange* que corresponde a una tabla con símbolos estandarizados utilizados por los computadores.

`unsigned char` Para este tipo de variables se dedican 8 *bits* de memoria, por tanto se pueden representar $256 = 2^8$ números que van desde el 0 al 255. La representación interna se realiza en base 2. De esta forma, el número representado en la figura 3.2(a), sería el número 23 en base decimal



(a) Esquema de almacenamiento interno de un `unsigned char` de 1 *byte*



(b) Mayor número de un `unsigned char`

Figura 3.2 – *Modelo de almacenamiento de `unsigned char`*

$$0 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 1 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 23$$

El mayor número sería el representado en la figura 3.2(b) y cuyo valor decimal viene dado por[4]

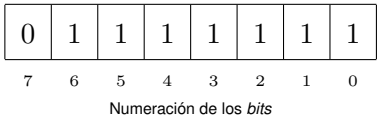
$$1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = \sum_{i=0}^{7} 2^i = \frac{1-2^8}{1-2} = 2^8 - 1 = 255$$

El menor sería cuando todos los *bits* fuesen cero, que obviamente correspondería al valor decimal, nulo. Por tanto, el rango de las variables declaradas como **`unsigned char`** sería $[0, 255]$, como puede verse en el cuadro 3.1 (pág. 27).

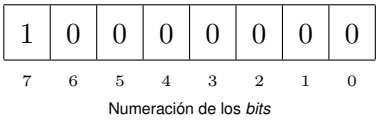
`char` Al igual que en el caso anterior, se dedican 8 *bits* para la representación del valor de la variable. Sin embargo, en este caso hay un *bit* que se dedica al signo. Habitualmente el más *significativo* (el que está más a la izquierda). Si es 0, se toma como positivo y si es 1, se toma como negativo. A partir de este criterio, existen diferentes maneras de representación de los números enteros con signo. El más habitual y eficiente es el llamado *complemento a dos*, que trata el *bit* del signo como un *bit* de magnitud. El criterio que establece el modelo de *complemento a dos*, es que la representación decimal debe ser:

$$\gamma \cdot 2^7 + \sum_{i=0}^{6} a_i 2^i$$

donde $\gamma \in \{0, 1\}$ y $a_i \in \{0, 1\}$ con $i = 0, \ldots, 6$. Si el número es *positivo*, el valor γ será 0. Esto implica que el mayor número positivo verifica que $\gamma = 0$ y sería el representado en la figura 3.3(a) con lo cual su



(a) Mayor número de un `char`



(b) Menor número de un `char`

Figura 3.3 – *Modelo de almacenamiento de `char`*

valor decimal vendría dado por

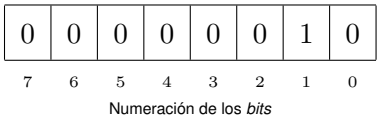
$$0 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = \sum_{i=0}^{6} 2^i = \frac{1-2^7}{1-2} = 2^7 - 1 = 127$$

[4] Utilizando la expresión de la suma de una progresión geométrica

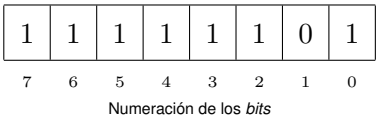
Con esta regla de *complemento a dos*, el número negativo más pequeño, sería el representado por la figura 3.3(b) cuya expresión decimal sería:

$$1 \cdot 2^7 + 0 \cdot 2^6 + 0 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = (-)2^7 = -128$$

que es negativo debido a que $\gamma = 1$. Para obtener la expresión binaria de los números negativos, el método *complemento a dos* toma el valor positivo en binario, calcula su opuesto (intercambia ceros por unos y *viceversa*) y suma una unidad. Por ejemplo, para determinar el valor -2 en binario, se tomaría el valor 2 en binario, que se representa en la figura 3.4(a) y se determina su opuesto, como muestra la



(a) Valor 2 en binario para una variable `char`



(b) Opuesto binario de 2 para una variable `char`

Figura 3.4 – *Modelo de almacenamiento de un número negativo para* `char`

figura 3.4(b) y por último se le sumaría 1, con lo que obtendría el valor -2 en binario, que se muestra en figura 3.5

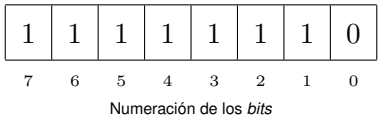


Figura 3.5 – *Valor en binario de* -2 *con criterio* complemento a dos *para una variable* `char`

pero que en decimal sería:

$$1 \cdot 2^7 + 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 = \sum_{i=1}^{7} 2^i = 2^8 - 2 = 254 \equiv -2$$

Así pues, el rango de las variables de tipo **`char`** estaría en el intervalo $[-2^7, 2^7 - 1]$, como muestra el cuadro 3.1 (pág. 27).

El **lenguaje C** proporciona instrucciones para determinar el rango de este tipo de variables *carácter* de manera directa, tal y como muestra el código 3.2. En este caso es necesario añadir un nuevo fichero en el *encabezamiento*: `limits.h`

Código 3.2 – *Programa que muestra el rango de los tipos* **`char`**

```
1  #include <stdio.h>
2  #include <limits.h> //Fichero para determinar los rangos de los enteros
3
4  int main(void)
5  {
6    printf("El rango de char está entre %i y %i\n", CHAR_MIN, CHAR_MAX);
7    printf("El rango de unsigned char está entre 0 y %u\n", UCHAR_MAX);
8
9    return 0;
10 }
```

y cuya salida en la pantalla sería la siguiente

```
El rango de char está entre -128 y 127
El rango de unsigned char está entre 0 y 255
```

Un ejemplo interesante para visualizar la consecuencia de almacenar un carácter en una variable tipo **unsigned char** o en una de tipo **char**, lo muestra el siguiente programa. El objetivo consiste en imprimir los códigos ASCII de las variables carácter.

Inicialmente se imprime el código ASCII de un carácter almacenado en una variable **unsigned char**[5]:

Código 3.3 – *Programa que muestra el código ASCII de un carácter*

```
1  #include <stdio.h>
2  /* Ejemplo de codificación ASCII caracteres */
3  int main(void)
4  {
5  unsigned char car;
6
7  printf("Introduce un caracter -> ");
8  scanf("%c", &car);
9
10 printf("El caracter es %c <-> %i <-> %X <-> %o\n\n", car, car, car, car);
11
12 return 0;
13 }
```

En este caso, si se introduce el carácter ñ el resultado sería

```
Introduce un caracter -> ñ
El caracter es ñ <-> 164 <-> A4 <-> 244
```

Sin embargo, si se cambia el tipo de la variable que almacena el carácter, por un **char**, entonces se obtendría

```
Introduce un caracter -> ñ
El caracter es ñ <-> -92 <-> FFFFFFA4 <-> 37777777644
```

La razón es que el código ASCII de ñ es 164 y una variable **char** tiene un rango, como se ha visto anteriormente en el intervalo $[-128, 127]$, con lo cual $164 \notin [-128, 127]$, pero sí está incluido en el intervalo $[0, 255]$ de las variables del tipo **unsigned char**.

Tipo int

Esta declaración ya se ha utilizado no sólo en el programa 3.1, sino también en los códigos 2.2 (pág. 15) y 2.3 (pág. 19). Hace referencia a una variable entera con signo, que ocupará 4 *bytes*. De manera similar a lo que sucedía con el tipo **char** hay una opción *sin* signo: **unsigned int**. La forma de almacenar internamente estas variables es análoga al caso de las variables **unsigned char** y **char** ya vistas. La diferencia es que ahora se utilizan 4 *bytes* para almacenar los datos, en lugar de 1 *byte*, empleado en el caso anterior. Por tanto para variables del tipo **int** el rango de variación sería $[-2^{31}, 2^{31} - 1]$ y en el caso de **unsigned int**: $[0, 2^{32} - 1]$, como puede comprobarse en el cuadro 3.1 (pág. 27).

Existen otros tipos de variables enteras, que se emplean según el número de *bytes* que se necesiten en un programa. Así el tipo **short int** y **unsigned short int** utilizan 2 *bytes*, **long int** y **unsigned**

[5]Este código utiliza funciones y sentencias que se explicarán más adelante. El objetivo es fijarse en el resultado que se obtiene.

long int 4 *bytes*, los mismos que **int** y **unsigned int**. Por último, **long long int** y **unsigned long long int** reservan 8 *bytes*. Los rangos de algunos de ellos pueden consultarse en el cuadro 3.1 (pág. 27).

Análogamente al caso anterior, a través de instrucciones del **C** puede obtenerse el rango de variación de los diferentes tipos, como se observa en el código 3.4

Código 3.4 – *Programa que muestra el rango de los tipos* **int**

```
1  #include <stdio.h>
2  #include <limits.h>       // Fichero para determinar los rangos de los enteros
3
4  int main(void)
5  {
6    printf("El rango de short está entre %i y %i\n", SHRT_MIN, SHRT_MAX);
7    printf("El rango de unsigned short está entre 0 y %u\n\n", USHRT_MAX);
8
9    printf("El rango de int está entre %i y %i\n", INT_MIN,  INT_MAX);
10   printf("El rango de unsigned int está entre 0 y %u\n\n", UINT_MAX);
11
12   printf("El rango de long está entre %li y %li\n", LONG_MIN, LONG_MAX);
13   printf("El rango de unsigned long está entre 0 y %lu\n\n", ULONG_MAX);
14
15   printf("El rango de long long está entre %lli y %lli\n",LLONG_MIN, LLONG_MAX);
16   printf("El rango de unsigned long long está entre 0 y %llu\n", ULLONG_MAX);
17
18   return 0;
19 }
```

la salida en la pantalla sería la siguiente

```
El rango de short está entre -32768 y 32767
El rango de unsigned short está entre 0 y 65535

El rango de int están entre -2147483648 y 2147483647
El rango de unsigned int está entre 0 y 4294967295

El rango de long están entre -2147483648 y 2147483647
El rango de unsigned long está entre 0 y 4294967295

El rango de long long están entre -9223372036854775808 y 9223372036854775807
El rango de unsigned long long está entre 0 y 18446744073709551615
```

que coincide con lo que muestra el cuadro 3.1 (pág. 27).

Tipo `float`

Las variables afectadas con este tipo de declaración permiten almacenar valores numéricos con decimales. Un ejemplo puede observarse en la LÍNEA 7 del programa 3.1 (pág. 28). Las variables `flotador` y `area`, se declaran como **float**, incluso la primera de ellas es inicializada. Para visualizar el valor de este tipo de variables se utiliza el *especificador de formato*: `%f`, como puede apreciarse en las LÍNEAS 21-22, del mismo programa.

En el *mundillo* de la Informática a estas variables se les identifica de manera general con el nombre de números *reales*. Pero como ya se ha comentado, están más cerca de lo que se entiende en Matemáticas por números Racionales ($\mathbb{Q}$), que de los números Reales ($\mathbb{R}$). La cuestión es que el compilador no almacena de la

misma forma un número entero, que un número dotado de parte decimal, aunque tengan el mismo número de *bytes* asignados. Para los números *reales* se suele utilizar el *formato estándar de representación (IEEE Storage Format)* (véase libro Infante *et al* [8, pág. 22]).

El hecho de utilizar **`float`** para declarar variables indica, no sólo que se almacenarán valores decimales, sino que además debe reservarse un tamaño de 4 *bytes*. En efecto, el compilador reserva 4 *bytes* que equivalen a 32 *bits* y en estas posiciones se almacena el número en binario, con una representación *científica*. Esto implica que se necesita espacio para almacenar el ***signo***, la ***mantisa*** y el ***exponente con su signo***. La regla que se considera es que de estos 32 *bits*, el primer *bit* lo utiliza para el signo (0 si es ***positivo*** y 1 si es ***negativo***), los 8 *bits* siguientes los emplea para almacenar el ***exponente*** y los 23 *bits* últimos para la ***mantisa***, tal y como se representa en la figura 3.6 y que se suele representar de manera *normalizada*[6], de la forma:

$$(-1)^s \times 1.m \times 2^{E_d - 127} \tag{3.1}$$

donde s es el signo, m es la *mantisa* (únicamente contiene 0's y 1's) y E_d es la expresión decimal del número binario guardado en los 8 *bits* dedicados al exponente. Esta manera de representación, permite que el $\boxed{1}$ más significativo de la representación *normalizada* (3.1) (esto es, el que está a la izquierda del punto), no sea necesario almacenarlo en m y así, dejar espacio para un *bit* más (aumentando la precisión).

Por otra parte, para evitar tener que utilizar un *bit* adicional para el signo del exponente, se establece el siguiente criterio[7]: para almacenar el valor (decimal) correspondiente al exponente y su signo, se suma 127 al exponente y se utilizan 8 *bits* para almacenarlo; de manera inversa, el exponente (en decimal) se obtendrá al restar 127 a la conversión decimal del exponente almacenado (que está en binario). Puesto que hay 8 *bits*, se pueden almacenar $2^8 = 256$ números (de 0 a 255). Sin embargo, se eliminan el 0 y el 255 (para números *especiales*), con lo que quedan 254 números. La representación en base binaria de los valores extremos del exponente serían:

- $-126 \Longrightarrow -126 + 127 = 1_{(10)} \Longrightarrow 0\,0\,0\,0\,0\,0\,0\,1_{(2)}$
- $127 \Longrightarrow 127 + 127 = 254_{(10)} \Longrightarrow 1\,1\,1\,1\,1\,1\,1\,0_{(2)}$

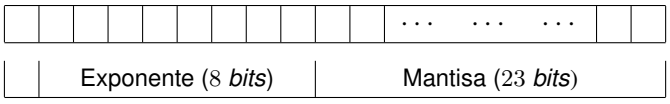


Figura 3.6 – *Esquema de almacenamiento interno de un float de* 4 bytes

Existen dos conceptos importantes que son ***precisión*** y ***rango***, que caracterizan a los distintos tamaños dedicados para almacenar números reales.

Precisión. Hace referencia al número de cifras con los que se representa la ***mantisa***. Para el caso del *tipo float* se dedican 23 *bits*, lo que quiere decir que el mayor número que puede representarse (teniendo en cuenta la suma de una progresión geométrica) es

$$1 \cdot 2^0 + 1 \cdot 2^1 + \cdots + 1 \cdot 2^{22} = \sum_{i=0}^{22} 2^i = \frac{1 - 2^{23}}{1 - 2} = 2^{23} - 1 = 8.388.607_{(10)}$$

Esto quiere decir que se pueden representar todos los números decimales de 6 cifras y la mayor parte (aunque no todos) de los de 7 cifras (por ejemplo el número 9.231.543 no se puede representar con 23 *bits*). Este número de cifras también suelen llamarse ***cifras significativas***. Por tanto, se dice que la *precisión* de esta representación está entre 6 y 7 *cifras significativas*.

[6] También llamada *representación Normal* del número binario

[7] Es el llamado criterio de *representación sesgada* o *codificación en exceso*, con sesgo $2^{k-1} - 1$. En este caso $k = 128$.

Rango. Se refiere a los valores que puede ser representados (con la limitación del tamaño de la mantisa), limitados por la cota superior e inferior (en valor absoluto). Para los números llamados *normales* (aquellos que tienen una representación de la forma (3.1)) los valores extremos serían:

- **Extremo superior.** Corresponde a un número positivo almacenado como muestra la figura 3.7

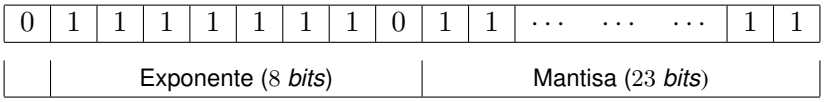


Figura 3.7 – *Esquema de almacenamiento interno del máximo número* normal *positivo de* 4 bytes

y cuya expresión en binario *normalizada* sería

$$\alpha_{(2)} = 1.\underbrace{11\ldots1}_{23} \times 2^{127} \equiv \underbrace{1}_{bit.sig.}\ \overbrace{\underbrace{11\ldots1}_{23}\underbrace{00\ldots0}_{104}}^{127}$$

La expresión en decimal de este número sería:

$$\alpha_{(10)} = 1 \times 2^{127} + 1 \times 2^{126} + \cdots + 1 \times 2^{104} + 0 \times 2^{103} + 0 \times 2^{102} + \cdots + 0 \times 2^{1} + 0 \times 2^{0}$$

Ahora bien, teniendo en cuenta que la suma para una progresión geométrica de razón 2 es

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1$$

resulta que

$$\sum_{i=0}^{127} 2^i = 2^{128} - 1$$

de esta forma:

$$\sum_{i=0}^{127} 2^i = 2^{128} - 1 = \sum_{i=0}^{103} 2^i + \underbrace{\sum_{i=104}^{127} 2^i}_{\alpha_{(10)}}$$

y despejando resulta

$$\alpha_{(10)} = \sum_{i=104}^{127} 2^i = \sum_{i=0}^{127} 2^i - \sum_{i=0}^{103} 2^i = \left(2^{128} - 1\right) - \left(2^{104} - 1\right) = 2^{128} - 2^{104} = 3.4028234 \times 10^{38}$$

que sería la *cota superior*.

- **Extremo inferior.** Corresponde a un número positivo almacenado como muestra la figura 3.8

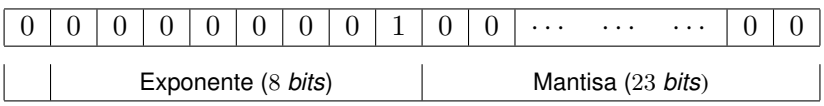


Figura 3.8 – *Esquema de almacenamiento interno del mínimo número* normal *positivo de* 4 bytes

y cuya expresión en binario sería

$$\alpha_{(2)} = 1.\underbrace{00\ldots0}_{23} \times 2^{-126} \equiv 0\,\underbrace{000\ldots000}_{125}\,1$$

y que en base decimal será

$$\alpha_{(10)} = 0 \times 2^0 + 0 \times 2^{-1} + \cdots + 0 \times 2^{-125} + 1 \times 2^{-126} = 1 \times 2^{-126} = 1.1754943 \times 10^{-38}$$

que representa la *cota inferior*.

Es importante insistir que **no** es posible representar en el computador **todos** los números x que cumplan

$$1.1754943 \times 10^{-38} \leqslant x \leqslant 3.4028234 \times 10^{38},$$

como consecuencia de la limitación física en el almacenamiento de la *mantisa*. Este resultado coincide con el que ya se mostraba en el cuadro 3.1 (pág. 27). Análogamente como en los casos anteriores, se puede determinar mediante instrucciones **C**, el rango y las cifras significativas, como muestra el código 3.5

Código 3.5 – *Programa que muestra el rango* **`float`**

```
1  #include <stdio.h>
2  #include <float.h>          // Límites para los tipos "punto decimal"
3
4  int main(void)
5  {
6    printf("El rango de un float es de %g a %g\n", FLT_MIN, FLT_MAX);
7    printf("El tipo float proporciona %u dígitos de precisión \n", FLT_DIG);
8
9    return 0;
10 }
```

con el siguiente resultado en pantalla:

```
El rango de un float es de 1.17549e-038 a 3.40282e+038
El tipo float proporciona 6 dígitos de precisión
```

que coincide con lo expuesto en el cuadro 3.1 (pág. 27).

Tipo `double`

Es similar al tipo **`float`**, la única diferencia es el número de *bytes* que se utilizarán para almacenar el valor numérico. En este caso, se emplean 8 *bytes*. Como es de suponer, la representación interna es análoga al caso anterior, únicamente cambia el tamaño para el almacenamiento del *exponente* y la *mantisa*. Se sigue empleando un *bit* para el signo del número, pero 11 *bits* para el *exponente* y 52 *bits* para la *mantisa*, tal y como muestra la figura 3.9

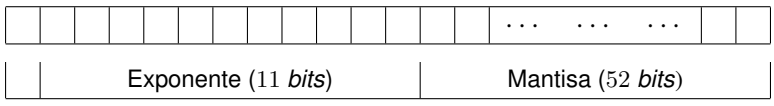


Figura 3.9 – *Esquema de almacenamiento interno de un double de* 8 bytes

En esta situación se tiene

Precisión. Puesto que la *mantisa* está constituida por 52 *bits* resulta que el mayor número representable será

$$\sum_{i=0}^{51} 2^i = 2^{52} - 1 = 4.503.599.627.370.495$$

por tanto tiene una *precisión* de entre 15 y 16 dígitos o *cifras significativas*.

Rango. Siguiendo un procedimiento análogo al establecido para el caso del *tipo float*, se llega a que el rango está entre

$$2.225073858507201 \times 10^{-308} \leqslant x \leqslant 1.797693134862316 \times 10^{308}$$

que como era de esperar, coincide con el que ya se mostraba en el cuadro 3.1.

Un ejemplo de utilización de este tipo de declaración e inicialización puede verse, de nuevo en el código 3.1 (pág. 28), en la LÍNEA 9. Para obtener una salida por pantalla se utiliza también el *especificador de formato*: `%f`, como puede apreciarse en la LÍNEA 26.

Es interesante además, resaltar la asignación que se hace a la variable **`Pi`** en la LÍNEA 11 del programa, con el resultado que se obtiene con los comandos de las LÍNEAS 23–24. El número de dígitos que se quieren mostrar en la salida, viene determinado por el ***formato***. En la LÍNEA 23 aparece el símbolo **`%f`**, que simplemente indica que esa posición se va a introducir un número *real*. El compilador toma *por defecto* el número de decimales. Sin embargo, en la LÍNEA 24 el *formato* es **`%.20f`** y con ello se le comunica al compilador de *manera explícita*, que deje 20 posiciones para la parte decimal.

Por último, en el programa 3.6 se muestran las instrucciones en **C** que permiten determinar el rango y la precisión (o *cifras significativas*) del tipo **`double`**.

Código 3.6 – *Programa que muestra el rango* **`double`**

```
1  #include <stdio.h>
2  #include <float.h>          // Límites para los tipos "punto decimal"
3
4  int main(void)
5  {
6    printf("El rango de un double es de %.15g a %.15g\n", DBL_MIN, DBL_MAX);
7    printf("El tipo double proporciona %u dígitos de precisión\n", DBL_DIG);
8
9    return 0;
10 }
```

con el siguiente resultado en pantalla:

```
El rango de un double es de 2.2250738585072e-308 a 1.79769313486232e+308
El tipo double proporciona 15 dígitos de precisión
```

que de nuevo, coincide con lo expuesto en el cuadro 3.1 (pág. 27).

3.2.2. Identificadores de variables

Como ya se ha comentado, en Informática un dato (ya sea una variable o constante simbólica) hace referencia a una posición de memoria que tiene asignado un número o ***dirección***. Esta es la manera más básica de referirse a la información que contiene. Es obvio, que referirse a una variable mediante una *dirección*, resulta poco práctico por la nula relación nemotécnica del número de la dirección con la información contenida en dicha posición. Además esta dirección puede cambiar en la propia ejecución de un programa. Es por ello que

C (al igual que los lenguajes de *alto nivel*), emplea un ***nombre simbólico*** para referirse a un dato, llamado ***identificador*** (algunos de ellos ya se han mostrado en programas anteriores). Los *identificadores* deben cumplir una serie de reglas:

- Un ***identificador*** se forma con una secuencia de caracteres alfanuméricos (*letras*: de la ***a*** a la ***z***; de la ***A*** a la ***Z*** y *números*: ***0*** a ***9***).
- El carácter ***subrayado***: '_', se considera como una *letra*.
- Un identificador no puede contener espacios en blanco ni caracteres especiales: ' %', '&', ';' , ',', '.', ':' ,'+', '-', *etc*
- El primer carácter de un identificador debe ser **obligatoriamente** una *letra* o el símbolo: '_'. No puede ser un dígito.
- Se hace distinción entre mayúsculas y minúsculas. De esta manera **`Velocidad`**, no es la mismo que **`velocidad`**, ni que **`VELOCIDAD`**.
- El estándar de **C** permite hasta 31 caracteres de longitud.

Es recomendable elegir *identificadores* o *nombres de variables* de forma que permitan conocer a simple vista, qué tipo de variable o constante representan, utilizando para ello tantos caracteres como sean necesarios. Esto simplifica enormemente la tarea de programación, de corrección y mantenimiento de programas. Por tanto, no se debe rehuir de nombres largos si son ilustrativos de lo que representan.

3.2.3. Dirección de las variables

Una de las características más destacables del **lenguaje C** es que puede acceder *explícitamente* a la dirección de memoria de los datos, mediante el ***operador de dirección*** `&`. En el programa 3.7 se muestra un código en donde se utiliza el operador `&` para determinar las direcciones de memoria que ocupa cada variable.

Código 3.7 – *Ejemplo de programa que presenta las direcciones que ocupan las variables en la memoria*

```
#include <stdio.h>
/* Salida de las direcciones que ocupan las */
/* variables en la memoria del ordenador */
int main(void)
{
        char carac;
        int entero = 15;
        float real = 345.1234567;
        double realDoble = 345.1234567;

        const double Pi = 3.141592653589793238;

        carac = 'a';

        printf("La direc. de carac es -> %u \n", &carac);
        printf("La direc. de entero es -> %u \n", &entero);
        printf("La direc. de real es -> %u \n", &real);
        printf("La direc. de realDoble -> %u  \n", &realDoble);
        printf("La direc. de Pi es -> %u \n", &Pi);

        return 0;
```

```
22 }
```

y cuya salida en el monitor es

```
La direc. de carac es -> 2293575
La direc. de entero es -> 2293568
La direc. de real es -> -> 2293564
La direc. de realDoble -> 2293552
La direc. de Pi es -> 2293544
```

A la vista de este código, el símbolo **`&realDoble`** por ejemplo, contiene un *entero sin signo* que representa la *dirección* de la variable **`realDoble`**.

Cada número representa la dirección del primer *byte* que ocupa la variable, si el *tipo de variable* declarado necesita más de uno. De esta manera, la variable **`real`** es del *tipo float* y utiliza 4 *bytes*, por tanto la dirección: 2.293.564 corresponde al primer *byte* de los cuatro reservados. O dicho de otro modo, esta variable ocupa los *bytes* que van desde la dirección 2.293.564 a la 2.293.567.

En la función de entrada **`scanf`**, que se explica más adelante, se utiliza este operador.

3.3. Funciones básicas de entrada–salida

El **lenguaje C** tiene dos funciones básicas para el manejo de entrada y salida de los datos: **`scanf()`** y **`printf()`**. Para poder utilizar estas órdenes de entrada–salida (como otras que se plantearán más adelante) es imprescindible incluir al comienzo del programa (en el *encabezamiento*) el fichero **`stdio.h`**, tal y como se comentaba en el capítulo anterior, en el programa 2.1 (pág. 13).

3.3.1. `printf()`

Imprime en la unidad de salida (el monitor suele ser la salida habitual), el texto y el valor de los datos que se indiquen. La expresión general de esta función toma la forma:

```
printf("cadena de control", var1, var2, ...)
```

donde `"cadena de control"` es un texto junto con una serie de ***formatos*** que depende del *tipo de variable*. El número de *formatos* tienen que coincidir con el número de variables y en el mismo orden. En los programas 3.1 y 3.7 se pueden observar diferentes ejemplos de utilización de la función **`printf()`**, con distintos *formatos*. El tipo de *formato* depende del *tipo de variable*. En el cuadro 3.2 se observan los *formatos* con sus respectivos *tipos de variables* asignadas. En el caso de variables enteras del tipo **`long long`**, el formato es el mismo que el tipo **`int`**, salvo que se debe añadir doble `l` antes que el descriptor. Esto es, `%lli` o `%lld`. Veamos a continuación un programa de ejemplo sobre el manejo de formatos

Código 3.8 – *Ejemplo de programa de manejo de los formatos con la función `printf()`*

```
1  #include <stdio.h>
2  /* Salida ejemplo de formatos */
3  int main(void)
4  {
5          char carac = 'r';
6          int entero = -1234567, i = 2, j = 3;
7          unsigned Ssigno = 396;
8          double realDoble = 345.1234567;
9
10         const double Pepi = 3141592.653589793;
```

Formatos	Tipos	Notas
%c	char	
%d %i	int	
%u	unsigned	Salida en sistema *decimal*
%x		Salida en sistema *hexadecimal*
%o		Salidas en sistema *octal*
%f	float	
	double	
%e	float	Notación exponencial
	double	
%g	float	Ajusta la salida numérica
	double	

Cuadro 3.2 – *Formatos*

```
11
12          printf("carac es -> %c <-> %d ASCII\n", carac, carac);
13          printf("entero es -> %d y sin signo -> %u\n", entero, entero);
14          printf("Ssigno es -> %d <-> %x hex <->%o oct\n", Ssigno, Ssigno, Ssigno);
15          printf("C(%d,%d) = %f\n", i,j,realDoble);
16          printf("realDoble es -> %f <-> %e\n", realDoble, realDoble);
17          printf("Pepi es -> %f <-> %e <-> %g \n", Pepi, Pepi, Pepi);
18          printf("Pepi es -> %.20f <-> %.20e <-> %.20g \n", Pepi, Pepi, Pepi);
19
20          return 0;
21  }
```

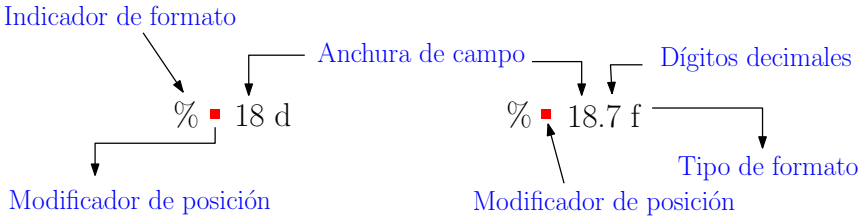
y cuyo resultado es

```
carac es -> r <-> 114 ASCII
entero es -> -1234567 y sin signo -> 429732729
Ssigno es -> 396 <-> 18c hex <-> 614oct
C(2,3) es -> 345.123457
realDoble es -> 345.123457 <-> 3.451235e+002
Pepi es -> 3141592.653590 <-> 3.141593e+006 <-> 3.14159e+006
Pepi es -> 3141592.65358979300000000000 <-> 3.14159265358979300000e+006 <-> 3141592.653589793
```

En la LÍNEA 13 se observa que a una variable *entera* se la asignado un formato `%u` (*entero sin signo*). Hay que tener cierta precaución porque esto no genera error de compilación, sino que ejecuta la sentencia y presenta un resultado, que en ocasiones (como el que se muestra) puede ser absurdo o impredecible. Así pues, es importante que los *formatos* se ajusten a los *tipos declarados* de las variables y constantes simbólicas.

En las LÍNEAS 17-18 se utiliza el formato `%g` que tiene la particularidad de ajustar la salida: puede ser en forma *exponencial* o no. Así mismo, tal y como se muestra en la LÍNEA 18, la salida numérica puede ser modificada variando los formatos (véase figura 3.10),

- `%`: es el indicador de formato.



(a) Formato para un número entero (b) Formato para un número real

Figura 3.10 – *Modelos de formatos*

- *Modificador de posición*: si existe un guión, indica que debe ajustarse a la izquierda.
- *Anchura de campo*: debe indicarse la cantidad de dígitos que ocupará el número. Si es real, hay que incluir tanto la parte entera como la decimal, incorporando un posible signo negativo. No se contabiliza el punto decimal. Si no aparece (como en la LÍNEA 18), se ajusta el número de dígitos de la parte entera.
- *Dígitos decimales*: indica el número de dígitos decimales que deben aparecer en la salida. Este es el factor importante en **C**. Puede modificar la *anchura de campo* para mostrar los *dígitos decimales* establecidos.
- *Tipo de formato*: `c,u,d,f,e,g`, depende del *tipo* de variable declarada.

Veamos un ejemplo

Código 3.9 – *Ejemplo para la variación de formatos en los números reales* `printf()`

```
1  #include<stdio.h>
2  /* Pruebas con formatos */
3  int main(void)
4  {
5          double Pope  = -3141592.653589793;
6
7          printf("/%d/\n", 12345);
8          printf("/%2d/\n", 12345);
9          printf("/%10d/\n", 12345);
10         printf("/%-10d/\n", 12345);
11         printf("\n\n");
12         printf("/%8.5f/\n", Pope);
13         printf("/%-20.5f/\n", Pope);
14         printf("/%15.5e/\n", Pope);
15         printf("/%15.12f/\n", Pope);
16
17         return 0;
18 }
```

cuya salida es

```
/12345/
/12345/
/    12345/
/12345    /


/-3141592.65359/
```

```
/-3141592.65359      /
/  -3.14159e+006/
/-3141592.653589793000/
```

La especificación de anchuras fijas resulta muy útil cuando se desean imprimir columnas de datos (por ejemplo, los coeficientes de una matriz). Al hacerse el ancho de campo, por defecto, equivalente a la anchura del propio número, el uso repetido un formato, por ejemplo

Repetición de formatos

```
printf("%d %d %d\n", num1, num2, num3);
```

produciría columnas desalineadas. Así la salida podría tener el siguiente aspecto

```
12 234 1222
4 5 23
22334 2322 10001
```

Por el contrario se puede conseguir una salida más nítida utilizando campos de anchura fija, lo suficientemente grandes. Así la sentencia

Repetición de formatos con anchura fija

```
printf("%9d %9d %9d\n", num1, num2, num3);
```

daría como resultado

```
       12       234      1222
        4         5        23
    22334      2322     10001
```

Secuencias de escape

Como se ha visto en los ejemplos anteriores, también es posible añadir en la *cadena de control* una serie de símbolos especiales como '\n'. Son las llamadas ***secuencias de escape***. Hay varias, cada una con un cometido distinto como pueden observarse en el cuadro 3.3. Veamos seguidamente un ejemplo sencillo de utilización de las *secuencias de escape*

Código 3.10 – *Ejemplo de utilización de alguna secuencia de escape*

```
#include <stdio.h>
/* Salida ejemplo de secuencias de escape */
int main(void)
{
      printf("Hola, voy a desparecer\r");
      printf("Esto es lo siguiente\n");
      printf("\t Veamos el tabulador\n\n");
      printf("Hola, ya no desaparezco\r\n");
      printf("Esto es siguiente\n");
      printf("\t Veamos el tabulador\n\n\n");
      printf("Voy a pitar \007 ...ya he pitado\n");

      return 0;
```

Secuencia	Significado
\n	Salto y comienzo de línea
\r	Comienzo de línea, sin salto
\t	Tabulador
\b	Retrocede un caracter
\'	Imprime comilla simple
\"	Imprime comilla doble
\\	Imprime *backslash*
\007	Pita (7 es el cód. ASCII para el pitido)

Cuadro 3.3 – *Algunas secuencias de escape*

```
14  }
```

cuya salida es

```
Esto es lo siguienteer
        Veamos el tabulador

Hola, ya no desaparezco
Esto es lo siguiente
        Veamos el tabulador

Voy a pitar ...ya he pitado
```

3.3.2. `scanf()`

Permite leer la entrada que teclea el usuario. Al igual que la función **`printf()`**, emplea una *cadena de control* y una lista de argumentos. La mayor diferencia es que es obligado la utilización del *operador dirección* '`&`' en los argumentos. Esto es, se introduce la *dirección de la variable*, en lugar de su nombre, simplemente. Así pues, se tiene la siguiente muestra

Ejemplo de la función `scanf()`

```
      printf("Introduce un número real ->  ");
      scanf("%f", &numero);
```

La función **`scanf()`** asigna el valor introducido por el teclado, a la dirección que apunta la variable **`numero`**. Dicho de otro modo, a la variable **`numero`**, se el asigna el número tecleado por el usuario. Veamos un ejemplo

Código 3.11 – *Ejemplo de utilización de la función `scanf()`*

```
1  #include <stdio.h>
2  /* Ejemplo para la función scanf() */
3  int main(void)
4  {
5        double peso, valor;
```

```
6
7         printf("Vale usted su peso en oro?\n\n");
8         printf("Por favor introduzca su peso en Kgr. -> ");
9         scanf("%lf", &peso);  // <- ¡¡¡ OJO !!!, lee una variable double
10
11        /* Supongamos que el kilo de oro está a 30.000 euros*/
12
13        valor = peso * 30000.;
14        printf("\n Usted vale %.2f euros\n", valor);
15        printf("\n\n Enhorabuena !!!\n");
16
17        return 0;
18  }
```

y cuyo resultado, una vez introducido el valor de 70 es

```
Vale usted su peso en oro?

Por favor introduzca su peso en Kgr. ->

 Usted vale 2100000.00 euros


 Enhorabuena !!!
```

Con la función **`scanf()`** hay que ser más cuidadoso con los formatos. Por ejemplo, con la función **`printf()`** se podía utilizar el especificador `%f` tanto para variables de *tipo* `float` como `double`. Sin embargo, aquí ya no, hay que tener cierta precaución. Veamos el cuál sería el resultado si se cambia el formato `%lf`, por el especificador `%f`, en la función **`scanf()`** de la LÍNEA 9. Esto es, si se sustituye la LÍNEA 9 por

Cambio de especificador de formato

```
        scanf("%f", &peso);
```

el resultado, para la misma entrada, sería

```
Vale usted su peso en oro?

Por favor introduzca su peso en Kgr. ->

 Usted vale 0.00 euros


 Enhorabuena !!!
Presione una tecla para continuar . . .
```

lo cual es absurdo. Así pues, es importante ajustar la especificación del formato, al *tipo declarado* para la variable o constante simbólica.

El **lenguaje C** declara la función **`scanf()`** como *entera*. Esto quiere decir que devuelve un valor entero cuando se ejecuta. Este valor entero, corresponde al número de *items* o *argumentos* que se leen correctamente. Por ejemplo, si se tiene una sentencia del tipo

Ejemplo de la función `scanf()`

```
        printf("Introduce dos números enteros ->  ");
        scanf("%i  %i", &ent1, &ent2);
```

la función **scanf()**, devolverá el valor entero $\boxed{2}$ si la lectura ha sido correcta, es decir, si se han introducido dos números enteros. Ahora bien, si primero se teclea un número entero y después un carácter (x, por ejemplo), entonces el valor que devuelve **scanf()** es $\boxed{1}$, indicando que sólo ha leído un argumento correctamente. Por otra parte, si el orden de la introducción de datos se hubiese realizado al revés (primero el entero y después el carácter), el valor de retorno hubiese sido $\boxed{0}$, debido a que no lee más datos, a partir de la lectura errónea. Así pues, **scanf()** devuelve el número de argumentos leídos *correctamente*. Esta característica se puede utilizar comprobar si los datos de entrada se introducen correctamente, en caso contrario, corregir esta introducción de datos, sin necesidad de interrumpir la ejecución del programa. Sin embargo, la función **scanf** es bastante *permisiva* y puede proporcionar resultados erróneos si no se tiene cierta precaución. En el siguiente código se lee una variable de tipo **unsigned int**, pero si se introduce *por error* un número real negativo, la función **scanf** no detecta ninguna anomalía. En efecto, sea el código

Código 3.12 – *Ejemplo de lectura errónea de la función `scanf()`*

```c
#include <stdio.h>
/* Lectura de enteros sin signo */
int main(void)
{
    unsigned int entS;
    int chk;

    printf("Introduce un entero sin signo -> ");
    chk = scanf("%i", &entS);
    printf("La salida de scanf es %i \n\n", chk);

    printf("El número introducido es %u <-> %i \n\n", entS, entS);

    return 0;
}
```

el resultado, si se introduce *por error*, el valor -34.67, sería:

```
Introduce un entero sin signo -> -34.67
La salida de scanf es 1

El número introducido es 4294967262 <-> -34
```

Como se observa, la salida de la función **scanf** (almacenada en la variable **chk**) toma el valor 1, lo que indica que no ha habido error en la lectura del valor de la variable. Pero claramente este valor no era, ni entero, ni *sin signo*, como se preveía en la codificación. De hecho era un real negativo. Sin embargo, no se ha detectado dicho error y además se ha almacenado la parte entera del número: -34. Por este motivo, se necesita tener cierta precaución al introducir los datos, ya que, como se ha visto, el **C** suele ser un lenguaje demasiado *tolerante*.

Otro situación interesante sobre el empleo de la función **scanf()** se describe a continuación:

Código 3.13 – *Ejemplo de utilización de la función `scanf()`*

```c
#include <stdio.h>
/* Ejemplo para la función scanf() */
int main(void)
{
      float var;
      char car;

```

```
8
9         printf("Introduzca un número -> ");
10        scanf("%f", &var);
11
12        printf("\n El número es -> %f\n\n", var);
13
14        printf("Ahora introduce un carácter -> ");
15        scanf("%c", &car);
16        printf("\n El carácter  es -> %c", car);
17
18        printf("\n \n Introduce ahora OTRO carácter -> ");
19        scanf("%c", &car);
20        printf("\n El carácter es -> %c;  \n", car);
21
22        return 0;
23  }
```

Se trata de un programa, que pide un número y lo imprime. A continuación hace lo mismo con un carácter y lo repite con otro carácter. Sin embargo, en la salida generada no se obtiene lo deseado. En efecto, el resultado, una vez introducido el número 45 es

```
Introduzca un número -> 45
 El número es -> 45.000000

Ahora introduce un carácter ->
 El carácter  es ->



 Introduce ahora OTRO carácter -> g
 El carácter es -> g;
```

Una vez tecleado el número 45 y presionada la tecla ENTER, inmediatamente se ejecuta la siguiente función y aparece en pantalla

```
Ahora introduce un carácter ->
 El carácter  es ->



 Introduce ahora OTRO carácter ->
```

y en este caso, se teclea g. Es decir, únicamente se ha podido introducir un carácter. ¿Qué ha sucedido? Resulta que el programa, en particular la función **`scanf()`**, no lee los datos a medida que se introducen por el teclado, sino que los lee de un *buffer*. La palabra ***buffer*** (palabra común en el argot de ordenadores) hace referencia a una zona de memoria que se utiliza para almacenamiento temporal de datos, generalmente de entrada y salida. Al pulsar la tecla ENTER, se da vía libre al programa para que capture el carácter o bloque de caracteres que se haya introducido en este *buffer*. También la propia tecla ENTER (o mejor dicho su código ASCII: 10) queda almacenada. Así pues, el *buffer* suele considerarse una especie de depósito, donde se almacenan los caracteres tecleados, junto con la 'tecla ENTER'[8].

[8]Esta situación es propia de los entornos *Windows*. En sistemas operativos como *Linux* o *Ios*, no se presenta. La razón se basa en que para *Windows*, el código asociado a la tecla ENTER, es considerado como un código de *caracter*, frente a otros sistemas operativos, que lo asocian a un código especial.

En el programa anterior 3.13, el valor 45 se introduce en la variable `var`, pero en el *buffer* queda almacenado el código de la tecla ENTER. De esta manera, la siguiente función **`scanf()`** en la LÍNEA 15, lee el código ASCII de la tecla ENTER que sigue en el *buffer*. Al ejecutar la siguiente sentencia en la LÍNEA 16, lo que hace es imprimir este carácter y por este motivo salta de línea, en la salida. Una vez que el *buffer* ya está vacío, el siguiente comando **`scanf()`** en la LÍNEA 19, lee correctamente el segundo carácter.

3.3.3. `getchar()` y `putchar()`

Para evitar la situación planteada con la función **`scanf()`** es necesario vaciar el *buffer*. Esto se puede hacer de varias maneras. Una de ellas (no muy ortodoxa ni elegante, pero que funciona) es leer el carácter de la tecla ENTER, con la función **`getchar()`**. Esta función carece de argumento (no se pone nada entre los paréntesis), simplemente captura el siguiente carácter disponible en el *buffer*. De esta manera, una posible solución se muestra en el siguiente programa

Código 3.14 – *Ejemplo de utilización de la función `scanf()`*

```
1  #include <stdio.h>
2  /* Ejemplo para la función scanf() */
3  int main(void)
4  {
5        float var;
6        char car;
7
8
9        printf("Introduzca un número -> ");
10       scanf("%f", &var);
11       getchar();
12
13       printf("\n El número es -> %f\n\n", var);
14
15       printf("Ahora introduce un carácter -> ");
16       scanf("%c", &car);
17       getchar();
18       printf("\n El carácter  es -> %c", car);
19
20       printf("\n \n Introduce ahora OTRO carácter -> ");
21       scanf("%c", &car);
22       getchar();
23       printf("\n El carácter es -> %c;  \n", car);
24
25       return 0;
26 }
```

Se ha introducido la función **`getchar()`** después de cada **`scanf()`**, con la misión de eliminar la 'tecla ENTER' del *buffer*. Sin embargo, existe una manera de asegurarse de que se *limpie* ***completamente*** el *buffer*, hasta encontrar la tecla ENTER. Esto se consigue mediante el operador `!=` (*distinto*), usado en el código 2.2 (pág. 15) y la instrucción **`while`**, empleada en el código 2.3 (pág. 19), con la sentencia

Sentencia para vaciar el buffer

```
while( getchar()!='\n' );
```

después de cada función **`scanf()`**. Únicamente, se obliga a leer caracteres del *buffer* hasta encontrar la tecla ENTER, que indicará que se ha terminado de introducir datos.

Por otra parte, existe una función que toma un carácter de programa y lo envía a la pantalla. Se denomina **`putchar()`**. En este caso, sí tiene argumento, que es el carácter que debe imprimirse en pantalla. Tienen sentido las siguientes sentencias:

```
putchar('R');    /* Las constantes de caracteres */
putchar('\t');   /* se escriben entre apóstrofos */
putchar(caracter);  /* caracter es una variable de tipo char */
putchar(getchar()); /* escribe lo que lee */
```

3.4. Operadores

Los ***operadores*** constituyen órdenes que pueden actuar sobre una variable (*operadores unarios*) o sobre dos variables (*operadores binarios*). En **C** hay diversos tipos de operadores: *aritméticos*, *lógicos*, *relacionales*, *incrementales*, *asignación* e incluso, *de bits*. Unos se utilizan más que otros, en esta sección se tratarán de los más básicos, que permitan abordar los aspectos fundamentales de la programación en el **lenguaje C**. No obstante, para obtener una información más detallada sobre los *operadores*, se puede consultar el libro de Rodríguez–Losada *et al* [18, pág. 23].

3.4.1. Operador de asignación

Ya se ha comentado (véase sección 3.2, pág. 25 en el apartado: *Inicialización de variables*), que el símbolo [=] no significa *igual a*. Se trata del más básico de los operadores de *asignación*. La sentencia,

Ejemplo de asignación

```
xyz = 2345;
```

introduce el valor 2345 en la variable 'xyz'. Por otra parte, la orden (como ya se ha planteado)

Ejemplo de asignación

```
i  = i + 1;
```

siendo muy común en programación, desde el punto de vista matemático, carece de sentido. Quiere decir: encuentra el valor de la variable **`i`**; a tal valor, súmale 1 y a continuación *asigna* este nuevo valor, a la variable de nombre **`i`**.

No tienen sentido en programación, por tanto, sentencias del tipo:

Sentencia carente de sentido

```
2.457 = valor;
```

si `valor` es una variable.

3.4.2. Operadores aritméticos

Se trata de operadores *binarios* que aceptan dos operandos de tipo numérico y devuelve un valor del ***mismo tipo*** que el de los operados. En el cuadro 3.4 se muestran los operadores aritméticos admitidos en **C**. Veamos

Símbolo	Significado
+	Suma
−	Resta
*	Producto
/	División
%	Módulo (o resto de la división entera)

Cuadro 3.4 – *Operadores aritméticos*

un ejemplo:

Código 3.15 – *Ejemplo de utilización de operaciones aritméticas*

```
#include <stdio.h>
/* Ejemplo de operaciones ariméticas */
int main(void)
{
      int entero1 = 37, entero2 = 15;
      int enteroresultado, restodivision;
      float real1 = 35.689, real2= 0.643;
      float realdivision;


      enteroresultado = entero1/entero2;
      restodivision = entero1  % entero2;

      realdivision = real1/real2;

      printf("Division entera -> %d; ", enteroresultado);
      printf("\t Resto ->  %d; \n\n", restodivision);
      printf("Division real -> %f\n", realdivision);

      return 0;
}
```

y cuyo resultado es

```
Division entera -> 2;    Resto ->  7;

Division real -> 55.503887
```

Conversión de tipo de variable

El **C**, a diferencia de otros lenguajes, permite la mezcla de *tipos* de variables durante la ejecución de una o varias operaciones aritméticas. En lugar de seguir la pista de un eventual error generado por esta combinación de *tipos*, utiliza una serie de reglas para efectuar automáticamente conversiones de *tipo*. Esta característica del **lenguaje C** puede ser muy útil en ocasiones, pero también puede producir más de un *quebradero de cabeza*,

cuando se mezclan *tipos* inadvertidamente. Es conveniente por tanto, tener una idea razonablemente clara del funcionamiento de las ***conversiones de tipo***. Las reglas básicas son:

1. El rango o categoría de los *tipos*, de mayor a menor, es el siguiente: **`double`**, **`float`**, **`long`**, **`int`**, **`short`**, **`char`**. Los *tipos* **`unsigned`** tiene el mismo rango que el *tipo* al que están referidos.
2. Una operación en que aparezcan dos tipos de variables de *rangos diferentes* se eleva la *categoría* del que tiene el *tipo de menor rango*, para igualarla a la del mayor. Este proceso se conoce con el nombre de *promoción*.
3. En una sentencia de asignación, el resultado final de los cálculos se reconvierte al tipo de la variable a que están siendo asignados. El proceso puede ser, una *promoción* o una *pérdida de rango*.

La *promoción* suele ser un proceso que pasa inadvertido fácilmente. La *pérdida de rango*, por el contrario puede generar auténticas catástrofes. Veamos un programa más amplio donde se comprueba el funcionamiento de éstas reglas.

Código 3.16 – *Ejemplo de utilización de operaciones aritméticas*

```
1  #include <stdio.h>
2  /* Ejemplo de operaciones aritméticas */
3  int main(void)
4  {
5        char caracter;
6        int entero;
7        float real;
8
9        real = entero = caracter = 'A';
10       printf("caracter = %c, entero = %d, real = %2.2f\n", caracter, entero,
             real);
11
12       caracter = caracter  + 1;
13       entero = real + 2*caracter;
14       real = 2.0 * caracter + entero;
15       printf("caracter = %c, entero = %d, real = %2.2f\n", caracter, entero,
             real);
16
17
18       return 0;
19 }
```

el resultado en pantalla es el siguiente:

```
caracter = A, entero = 65, real = 65.00
caracter = B, entero = 197, real = 329.00
```

En la LÍNEA 13 se produce un proceso de *promoción*: la variable `caracter` pasa *tipo* entero para multiplicarlo por 2; después el resultado se *promociona* a **`float`** para sumar la variable `real` y por último se transforma en entero al almacenarse en la variable `entero`, pudiéndose producir una pérdida de rango. Sin embargo, en la LÍNEA 14 el proceso es una *promoción*: la variable `caracter` se convierte en punto flotante para multiplicarla por 2.0; a continuación la variable `entero` también se pasa a **`float`** para sumarle el resultado anterior y por último se almacena en la variable `real`.

Es posible realizar una conversión de tipos de manera explícita con el operador *cast*. Consiste en poner entre paréntesis el *tipo* al cual quiere transformarse la variable que queda a su derecha. Por ejemplo, la operación

```
        numreal = entero / 28;
```

realiza una división *entera*, si `entero` es una variable de tipo **`int`**. Esto es, si `entero` no es un múltiplo de 28 entonces únicamente se almacena la parte entera de la división en `numreal`, aunque está sea de tipo *decimal* (es decir, **`float`** o **`double`**). Si se quiere que la división sea *real*, esto es, que almacene la parte decimal, implica que además de que `numreal` sea *decimal*, la división también lo deberá ser. Para conseguir esto, se *promociona explícitamente* a la variable `entero` a *decimal*, con el operador *cast*, como indica la operación

```
        numreal = (float) entero / 28;
```

Un ejemplo sobre las *conversiones de tipo*, se muestra en el código 3.17:

Código 3.17 – *Ejemplo de conversiones de tipo*

```
#include <stdio.h>

int main (void)
{

    float real1 = 3487.3095, real2;
    int ent1, ent2 = -2635;


    ent1 = real1; // conversión a entero, implícita
    printf ("%f se transforma en el entero %i\n\n", real1, ent1);

    real1 = ent2; // conversión a real, implícita
    printf ("%i se transforma en el real %f\n\n", ent2, real1);

    real1 = ent2 / 100; // división entera
    printf ("%i dividido por 100 es %f\n\n", ent2, real1);

    real2 = ent2 / 100.0; // división real
    // (entero dividido por real). Conversión implícita
    printf ("%i dividido por 100.0 es %f\n\n", ent2, real2);

    real2 = (float) ent2 / 100; // conversión explícita
    printf ("(float) %i dividido por 100 es %f\n\n", ent2, real2);

    return 0;
}
```

y cuya salida es

```
3487.309570 se transforma en el entero 3487

-2635 se transforma en el real -2635.000000

-2635 dividido por 100 es -26.000000

-2635 dividido por 100.0 es -26.350000

(float) -2635 dividido por 100 es -26.350000
```

Símbolo	Ejemplo	Interpretación
+=	**entre** += **salga**	**entre** = **entre** + **salga**
−=	**pinto** −= **nada**	**pinto** = **pinto** − **nada**
⋆=	**producto** ⋆= 25	**producto** = **producto** ⋆ 25
/=	**valde** /= 8.34	**valde** = **valde** / 8.34
%=	**moro** %= 3	**moro** = **moro** % 3

Cuadro 3.5 – *Operaciones de asignación compuestas*

Resulta muy ilustrativo a analizar los resultados de este programa.

Existe otro proceso de conversión que no se ha mencionado y que realiza internamente el compilador de **C**. En efecto, con el fin de conservar al máximo, la precisión numérica, **todas** las variables y constantes **`float`**, se transforman en **`double`**, cuando se realizan los cálculos aritméticos con ellas; así se reduce enormemente el error de redondeo. La respuesta final se reconvierte en **`float`**, si ese era el *tipo* declarado inicialmente.

3.4.3. Operadores de asignación compuestos

Son operadores que conjugan la *asignación* [=] con *operadores aritméticos*. Realmente son operadores de *actualización* de variables. Todos ellos emplean un nombre de variable a su izquierda y una expresión a su derecha. La variable queda asignada a un nuevo valor igual a su antiguo valor operado por el valor de la expresión de su derecha. La operación concreta a que se somete la variable depende del operador aritmético. Estas asignaciones se pueden ver en el cuadro 3.5. En la parte de la derecha del operador se pueden utilizar expresiones más sofisticadas:

```
X *= 3*Y + 12;
```

que es equivalente a

```
X = X * (3*Y + 12);
```

3.4.4. Operadores incrementales

El operador *incremento*: ++ (respec. *decremento*: −−) realiza una tarea muy simple: incrementa (respec. decrementa) el valor de su operando en una unidad. En el **lenguaje C** se ofrecen dos variedades en este tipo de operadores. La primera de ellas se denomina *sufijo*, en el que operador ++ (o bien, −−) se encuentra a la derecha de la variable; mientras que la otra posibilidad: *prefijo*, está justamente a la izquierda. Esto implica dos modalidades a la hora de modificar el valor de la variable. En el procedimiento *sufijo*, primero se opera sobre la variable y después se incrementa (respec. disminuye). Sin embargo, para la variedad de *prefijo*, la situación es al revés: primero se incrementa (respec. decrementa) y luego opera sobre la variable. Veamos un ejemplo con el operador ++ (análogamente sería para −−).

Código 3.18 – *Operador incremental*

```
1 #include <stdio.h>
2 /* Ejemplo de operaciones incrementales */
```

```
3  int main(void)
4  {
5        int incre1, incre2, ent = 4;
6
7
8        incre1 = 2 * ++ent;
9        ent = 4;
10       incre2 = 2 * ent++;
11       printf("incre1 = %5d, incre2 = %5d\n", incre1, incre2);
12       printf("ent -> %d\n", ent);
13
14
15       return 0;
16 }
```

la salida en pantalla es

```
incre1 =    10, incre2 =     8
ent -> 5
```

El proceso es el siguiente. En la LÍNEA 8 el operador está en la modalidad *prefijo*. Esto implica que la variable `ent` aumenta en una unidad, antes de operar. Con lo cual `ent` toma el valor 5. A continuación opera y el valor almacenado en `incre1` es 10, como se observa en la salida. Sin embargo, en la LÍNEA 10, se tiene la modalidad *sufijo*. Así pues, se opera con el valor `ent` asignado. Es decir, `ent = 4` y a continuación se opera. Con lo cual la variable `incre2` almacena el valor 8, como se muestra en la salida. Una vez terminadas las operaciones, se incrementa en una unidad el valor de `ent`, cuyo resultado es 5.

Una consecuencia importante de esta manera de proceder, es que estos operadores tiene una elevada ***precedencia*** (el operador `++` o `--` únicamente afecta a la variable que acompaña); tan sólo superados por el paréntesis. Por ello

```
X * Y++;
```

significa

```
(X) * (Y++);
```

y **NO**

```
(X * Y)++;
```

Ahora bien, la *precedencia* no hay que confundirla con el *orden de evaluación* de los operadores. La *precedencia* establece la relación entre el operador y la variable sobre la que actúa. En el ejemplo del código 3.18, se ha visto en la LÍNEA 10, operaciones con la variable `ent` y sólo después de realizarlas, incrementaba la variable en una unidad. Por tanto, cuando nos encontremos un **`i++`** (respectivamente, **`i--`**) como parte de una expresión, se puede interpretar como: *utiliza* **`i`**; *a continuación increméntalo* (respectivamente, *decreméntalo*). Por el contrario, **`++i`** (respectivamente, **`--i`**) significaría: *incrementa* (respectivamente, *decrementa*) **`i`**; *a continuación, utilízalo*.

Una fuente de problemas en el empleo de este tipo de operadores son sentencias del tipo:

```
sal = entero/2 + 8*(7 + entero++);
```

Operador	Significado
<	Menor que
<=	Menor o igual que
==	Igual a
>=	Mayor o igual que
>	Mayor que
!=	Distinto de

Cuadro 3.6 – *Operadores de relación*

en este caso, no está claro que el compilador vaya ejecutarla en el orden en que pensamos: empieza por `entero/2` y sigue con la línea. Pues bien, puede ocurrir que calcule el último término antes, realice el incremento y use el nuevo valor de `entero` para evaluar `entero/2`. Para evitar este tipo de problemas se recomienda

1. No se debe utilizar operadores incremento o decremento en variables que se emplean más de una vez como argumento de una función:

```
        printf("%10d, %10d\n", entero, entero*entero++); /* EVITAR */
```

2. No se debe utilizar operadores incremento o decremento en variables que se empleen más de una vez en una misma expresión.

3.4.5. Operadores de relación

Los *operadores de relación* se emplean para hacer comparaciones. La lista completa de estos operadores puede verse en el cuadro 3.6. Cada expresión en **C** tiene un valor numérico y esto no es una excepción para sentencias que involucran operadores de relación. Una operación de relación tiene dos posible valores: *verdadero* o *falso* y estos resultados los cuantifica numéricamente el **lenguaje C**. Veamos un ejemplo

Código 3.19 – *Operador de relación*

```
#include <stdio.h>
/* Ejemplo de operadores de relación */
int main(void)
{
       int verdadero, falso;

       verdadero = 16 > 8;
       falso = 15 == 3;

       printf("verdadero = %d; falso = %d \n", verdadero, falso);

       return 0;
}
```

con el resultado en pantalla

```
verdadero = 1; falso = 0
```

En la LÍNEA 7 comprueba que es verdadero que $16 > 8$ y este resultado lo almacena en la variable: `verdadero`. En este caso el valor es 1. Sin embargo, en la LÍNEA 8 comprueba si son iguales 15 y 3; al ser distintos el resultado es *falso* y almacena en la variable `falso`, el valor 0. Por tanto, 0 es ***falso*** y 1 es ***verdadero***. Realmente para **C**, todo valor numérico (generalmente entero) que no tome el valor 0 es ***verdadero*** (con la sentencia **`if-else`**, ya vista en el código 2.2 de pág. 15, se puede probar fácilmente este hecho).

3.4.6. Operadores lógicos

Algunas veces es útil comparar dos o más expresiones de relación. Para ello se emplean los *operadores lógicos*. En **C** existen tres operadores lógicos, como muestra el cuadro 3.7.

Operador	Interpretación	Empleo	Explicación
`&&`	**y**	**`exp1 && exp2`**	únicamente verdadero, si **`exp1`** y **`exp2`** son ciertas
`||`	**o**	**`exp1 || exp2`**	únicamente falso, si **`exp1`** y **`exp2`** son falsas
!	**no**	**`!exp`**	falso si **`exp`** es verdadero y al revés

Cuadro 3.7 – *Operadores lógicos*

Veamos algunos ejemplos en el código 3.20

Código 3.20 – *Utilización de operadores lógicos*

```
1  #include <stdio.h>
2  /* Ejemplo de operadores lógicos */
3  int main(void)
4  {
5        int decide;
6
7        decide = 7 > 3 && 8 < 3;
8        printf("7 > 3 && 8 < 3 es -> %d\n", decide);
9
10       decide = 7 > 3 || 8 < 3;
11       printf("7 > 3 || 8 < 3 es -> %d\n", decide);
12
13       decide = !(3 > 8);
14       printf("!(3 > 8) es -> %d\n", decide);
15
16       return 0;
17 }
```

obteniéndose en pantalla

```
7 > 3 && 8 < 3 es -> 0
7 > 3 || 8 < 3 es -> 1
!(3 > 8) es -> 1
```

En el código 3.20, la LÍNEA 7 es *falsa* porque sólo una de las desigualdades es cierta. Sin embargo, la expresión de la LÍNEA 10 es *verdadera* porque sólo una de las desigualdades es cierta. La sentencia de la LÍNEA 13 es *verdadera* ya que 3 no es mayor que 8. Todos estos resultados se pueden observar en la salida ya que, no olvidemos que para **C**, es *falso* si es $= 0$, pero *verdadero* si es $\neq 0$.

Precedencia

El operador `!` tiene una prioridad muy alta, mayor que la multiplicación y división, igual a la de los operadores *incremento y decremento* e inmediatamente inferior a la de los paréntesis. El operador `&&` tiene mayor prioridad que `||`, estando ambos situados por debajo de los *operadores de relación* y por encima de la *asignación*. Por consiguiente la expresión

```
a >= b && b < c || b > d;
```

se interpreta como

```
( (a >= b) && (b < c) ) || (b > d);
```

Orden de evaluación

Cuando se estudiaron los operadores *incrementales* y *decrementales* (véase subsección 3.4.4 pág. 52), ya se comentó sobre el *orden de evaluación de los operadores* no tenía que coincidir con su *precedencia*. Ahora bien, es importante tener en cuenta que normalmente en **C**, no se garantiza qué parte de una expresión compleja se evalúa primero. Por ejemplo, en la sentencia

```
esfuerzo = (3 + 5) * (15 - 9);
```

la expresión $3 + 5$ puede evaluarse antes que $15 - 9$ o al revés, sin embargo la *precedencia* garantiza que ambas se realizarán antes que efectuar el producto. Esto es una ambigüedad que se dejó a propósito en el lenguaje, a fin de permitir a los diseñadores de los compiladores, pudiesen preparar versiones más eficientes. Sin embargo, existe una excepción en el tratamiento de los *operadores lógicos*. En **C** se garantiza que las ***expresión lógicas*** se evalúan de izquierda a derecha. También queda garantizado que tan pronto se encuentre un elemento que invalida la expresión completa, cesa la evaluación de la misma. Por ejemplo

```
numero != 0 && numero/4 == 5;
```

indica que si la variable `numero` tiene un valor igual a 0, entonces la expresión es *falsa* y el resto **NO** se evalúa. En la tabla 3.8 se muestra la *precedencia o prioridad* de los operadores, que no tiene que coincidir con el *orden de evaluación* en una expresión, como ya se ha comentado.

3.5. Preprocesador

Se trata de un componente característico del **lenguaje C**, que no es habitual en otros lenguajes de programación. El *preprocesador* actúa sobre el código antes de que empiece la compilación, propiamente dicha, para

Precedencia de los operadores	
`()`	paréntesis
`!` ; `++` ; `--`	**no** lógico y los incrementales
`*` ; `/` ; `%`	multiplicación y división
`+` ; `-`	suma y resta
`<` ; `<=` ; `>` ; `>=`	desigualdades
`==` ; `!=`	igual que; distinto que
`&&`	**y** lógico
`||`	**o** lógico
`=`	asignación

Cuadro 3.8 – *Precedencia de los operadores. De mayor (más arriba) a menor (abajo)*

realizar ciertos cometidos. Uno de estos cometidos consiste, por ejemplo, en incluir las funciones básicas de *entrada y salida* de datos mediante la *instrucción* `#include`, como se ha visto en los ejemplos de programas anteriores.

Existen diferentes tipos de comandos propios de *preprocesador*, también llamados ***directivas***. Todos ellos deben empezar por el símbolo `#`. En esta sección únicamente se tratarán las directivas: `#define` e `#include`.

3.5.1. `#define`

Ya se ha visto en la sección 3.2 (pág. 25) que una manera de definir constantes simbólicas es mediante el cualificador **`const`**. Esto se puede hacer también en **C** utilizando la directiva `#define` en el *preprocesador*. Incluso con esta directiva es posible definir funciones sencillas con argumentos, como muestra el siguiente ejemplo

Código 3.21 – *Ejemplo con la directiva `#define`*

```
#include <stdio.h>

/* Ejemplos sencillos de preprocesador */
#define PI 3.1415926536
#define PA(X) printf("El área de la tortilla es -> %.14f cm2\n", X)
#define PL(X) printf("La longitud de la tortilla es -> = %.14f cm\n", X)

int main(void)
{

      double area, longitud, radio;

      printf("Introduce el radio de la tortilla -> (en cm, claro)");
      scanf("%lf", &radio);
      area = PI * radio * radio;
      longitud = 2 * PI * radio;

```

```
18          PA(area);
19          PL(longitud);
20
21
22          return 0;
23      }
```

una vez introducido el valor de 12, la salida en la pantalla es

```
El área de la tortilla es -> 452.38934211840001 cm2
La longitud de la tortilla es -> = 75.39822368640000 cm
```

En la LÍNEA 4 se establece `PI` como una constante simbólica, que posteriormente se utiliza en las LÍNEAS 15-16. Cualquier sentencia dentro de la función **`main`** del tipo:

```
        PI = 45.7;
```

hubiese dado un error de compilación. Por tanto, este procedimiento es equivalente a utilizar el cualificador **`const`**[9]. Así mismo, en las LÍNEAS 5-6 se utiliza, de nuevo la directiva `#define`, para definir funciones con un argumento (pueden ser más, si es necesario) que después se emplean en las LÍNEAS 18-19.

Como se observa en el código 3.21, cada utilización de la directiva `#define` consta de tres partes. En primer lugar, la directiva `#define`. La segunda parte, consiste en establecer la palabra que se quiere definir, que suele denominarse '***macro***' dentro del mundillo de la programación. La *macro* debe ser una única palabra, cuyo nombre debe seguir las mismas reglas que se utilizan para denominar las variables en **C**. En la tercera parte se escriben una serie de caracteres que van a ser representados por la *macro*.

Generalmente las *macro* se utilizan para definir funciones sencillas o establecer constantes simbólicas, como se ha visto. En caso de necesitar definir funciones más complejas se emplean procedimientos que se estudiarán más adelante.

3.5.2. `#include`

Cuando el *preprocesador* encuentra un comando o directiva `#include` busca el fichero que atiende por el nombre está situado a continuación y lo incluye en el fichero actual. El nombre del fichero puede venir de dos formas:

```
#include <stdio.h>
#include "misfunciones.h"
```

En el primer caso, los paréntesis *angulares* le está diciendo que busque el fichero en un directorio estándar del sistema. En el segundo caso, las *comillas* le dicen que lo busque en el directorio actual (en donde tiene el fichero fuente) en primer lugar, y si no lo encuentra, que lo busque en el directorio estándar.

La cuestión de ¿por qué es necesario incluir ficheros? se contesta simplemente: porque tienen la información que se necesita. El fichero **`stdio.h`**, por ejemplo, contiene generalmente las definiciones de **`printf`**, **`scanf`**, **`EOF`** (*End Of File*) y otras más.

El sufijo **`.h`** se suele emplear para ***ficheros de encabezamiento*** (*header*), es decir, con información que debe ir al principio del programa. Los *ficheros cabecera* (como también se llaman) consisten, generalmente, en sentencias para el *preprocesador*. Algunos como **`stdio.h`** vienen con el sistema, pero el usuario puede crear los que quiera para dar su *toque personal* a los programas.

[9] No obstante, cada vez se utiliza menos este procedimiento y se tiende a emplear más el cualificador **`const`**, por la compatibilidad con **C++**

3.6. Ficheros

En ciertas ocasiones, debido a la cantidad de datos obtenidos en la salida de un programa, resulta imposible realizar un análisis sobre la pantalla. Para solventar esta situación, se dispone de los ***ficheros***, que permiten almacenar los datos para un estudio posterior. Así pues, una posible definición de *fichero* sería: una porción de memoria *permanente* que puede consultarse cuando el usuario lo requiera. Desde el punto de vista del **C**, el concepto de *fichero* es más complicado, pero esto algo que no se estudiará en este documento. Sin embargo, resulta importante tener claro el procedimiento que debe realizarse para escribir o leer datos de un fichero mediante el **lenguaje C**:

1. Primeramente hay que *abrir* el fichero. Para ello es necesario introducir el *nombre del fichero*. El sistema comprueba si el fichero con ese nombre existe. Si no, y según en qué casos, lo puede crear.

2. Una vez abierto, hay que especificar qué procedimiento de E/S (ENTRADA/SALIDA) se va a realizar con dicho fichero. Si se utilizará para *leer*, habrá que especificar que debe abrirse en *modo lectura* (*read mode*). Si se utilizará para escribir datos, entonces se abrirá en *modo escritura* (*write mode*) y por último si se abre para añadir datos se deberá abrir en *modo adición* (*append mode*). En los dos últimos casos: *modo escritura* y *modo adición*, en caso de no existir los ficheros, el **C** los crea. En el caso del *modo lectura*, si no existe el fichero, el sistema manda una señal de error.

3. Por último hay que informar al programa cómo puede identificar el fichero. Para ello, una vez *abierto* el fichero, **C** asocia a dicho fichero un número. Dicho número corresponde al valor de una variable del *tipo* ***puntero a fichero***[10], que hay que declarar antes de abrir el fichero. Mediante este número, el programa ya puede identificar al fichero y con las funciones adecuadas del **lenguaje C**, ya se pueden realizar las operaciones de E/S en este fichero. Es frecuente llamar a la variable *canal de interconexión* o simplemente *canal*.

La función **`fopen`** permite *abrir* un fichero, adjudicar un *modo* (*lectura, escritura o adición*) y devolver un valor para el *puntero a fichero* que identificará el programa con el fichero. La función tiene dos argumentos:

```
fi = fopen("cadena de caracteres", "modo")
```

El primer argumento: **`cadena de caracteres`**, hace referencia al *nombre del fichero* que se va a abrir. Si es una constante, esto es, un nombre, debe escribirse entre comillas: `"Datos.dat"`. El segundo: **`modo`**, describe el uso que se le va a dar al fichero, identificado por tres letras:

- `"r"`: un *fichero de lectura* (*read*). Únicamente se puede leer datos. Si el fichero no existe, devuelve un valor que se asocia al nombre de `NULL` que indica un fallo en la apertura. Un ejemplo de utilización del valor de *puntero a fichero* `NULL`, puede apreciarse en el siguiente trozo de código:

```
#include <stdio.h>

int main(void)
{

    FILE *fi;

    fi = fopen("Datos.dat", "r");

    if (fi == NULL){
```

[10] El concepto de *puntero* se tratará más adelante en el documento (véase capítulo 6, pág. 133) de manera más amplia.

```
                printf("¡¡¡¡ ERROR en la apertura del fichero: Datos.dat !!!!
                    ");
                return 5; //
            } else {
                // LECTURA DE DATOS DEL FICHERO
                .............................
                .............................
```

- `"w"`: un *fichero de escritura* (*write*). Se utiliza para escribir datos. Si no existe el fichero en el directorio, lo crea. Si existe, borra todo lo que hay en su interior e introduce los nuevos datos generados por el programa.
- `"a"`: un *añadido al fichero* (*append*). Si el fichero existe, entonces añade los nuevos datos generados, a partir de los ya existentes en el fichero.

Por último, si no ha habido ningún error en la apertura, la función **`fopen`** retorna un valor que se almacena en un *puntero a fichero* (*canal*), que será el valor que utiliza el programa para conectarse con el fichero. En el caso anterior, se almacena en la variable **`fi`**.

Un ejemplo de utilización de la función **`fopen`** puede verse en el código 3.22. En la LÍNEA 10, se *abre* un fichero llamado `Datos.dat` en *modo escritura* (`"w"`). Si no está en el escritorio, lo crea y si existiese, borra todo su contenido. Así mismo, la función **`fopen`** retorna un valor, que se almacena en la variable *puntero a fichero*: `fi`. Es importante destacar que esta variable ha sido declarada previamente, en la LÍNEA 5.

Así mismo, existe la función **`fclose`**, cuya misión es *cerrar* el fichero. Esto es, desconectar el fichero del programa. Habitualmente si no se pone explícitamente, el sistema provoca el *cierre* automáticamente, al terminar la ejecución. No obstante, es una buena costumbre de programación hacer esta *desconexión* explícitamente. La función **`fclose`** sólo tiene un argumento, que es la variable *puntero a fichero* obtenida con la función **`fopen`**. En la LÍNEA 18 del código 3.22, puede observarse un ejemplo de utilización de **`fclose`**.

Uno de los empleos más habituales de los ficheros es almacenar datos. Estos datos se pueden escribir en el fichero mediante el comando **`fprintf`** (véase el código 3.22). Es similar a la orden **`printf`**, ya estudiada. La única diferencia está en que **`fprintf`**, necesita como primer argumento el valor del *puntero a fichero*, obtenido mediante la función **`fopen`**.

También es posible leer datos con los comandos de lectura, por ejemplo, **`fscanf`**, que tiene las mismas particularidades que el comando **`fprintf`**. No obstante, en este documento nos centraremos, fundamentalmente en la escritura de datos en los ficheros. Para una información más extensa, puede consultarse el libro de Rodriguez–Losada *et al*[18].

Código 3.22 – *Ejemplo de operador incremental con ficheros*

```
1  #include <stdio.h>
2  /* Ejemplo de operaciones incrementales con fichero */
3  int main(void)
4  {
5        FILE *fi;
6
7        int incre1, incre2, ent = 4;
8
9
10       fi = fopen("Datos.dat", "w");
11
12       incre1 = 2 * ++ent;
13       ent = 4;
```

```
14          incre2 = 2 * ent++;
15          fprintf(fi, "incre1 = %5d, incre2 = %5d\n", incre1, incre2);
16          fprintf(fi, "ent -> %d\n", ent);
17
18          fclose(fi);
19
20          return 0;
21
22  }
```

La salida es idéntica a la obtenida mediante el código 3.18 (pág. 52), pero con la diferencia que con este programa, la salida se almacena en el fichero `Datos.dat` que se crea en el directorio en el que estemos.

El **C** proporciona una *señal* para indicar que se ha terminado de leer los datos de un fichero. Se encuentra definida en el fichero **`stdio.h`** y se representa por **`EOF`**.

3.7. Funciones matemáticas

A veces a la hora de realizar un programa es necesario utilizar alguna función matemática. El compilador de **C** tiene un fichero que contiene ciertas definiciones de estas funciones. Se trata del fichero `<math.h>` que se sitúa en el encabezamiento del fichero fuente y que es gestionado por el *preprocesador* mediante la directiva **#include**. Las funciones matemáticas más habituales se pueden observar en la tabla 3.9 (pág. 62). En esta tabla las variables **`x`** e **`y`** son de tipo *double* y todas funciones devuelven un *double*. Los ángulos de las funciones trigonométricas están representadas en radianes. Para trabajar con el *tipo* **`float`** o **`double`** se debe añadir el carácter **`f`** al nombre de la función. Si es de *tipo* **`complex`** (véase sección 5.6.1, pág. 121) se debe añadir el carácter **`c`** al nombre de la función.

Expresión	Función	Notas
`sin(x)`	$\mathrm{sen}(x)$	Función seno
`cos(x)`	$\cos(x)$	Función coseno
`tan(x)`	$\mathrm{tg}(x)$	Función tangente
`asin(x)`	$\mathrm{arc\,sen}(x)$	Función arco seno en $[-\pi/2, \pi/2]$ y $x \in [-1, 1]$
`acos(x)`	$\mathrm{arc\,cos}(x)$	Función arco coseno en $[0, \pi]$ y $x \in [-1, 1]$
`atan(x)`	$\mathrm{arc\,tg}(x)$	Función arco tangente en $[-\pi/2, \pi/2]$
`atan2(y,x)`	$\mathrm{arc\,tg}\,(y/x)$	Función arco tangente de (y/x) en $(-\pi, \pi]$
`sinh(x)`	$\mathrm{Sh}(x)$	Función seno hiperbólico
`cosh(x)`	$\mathrm{Ch}(x)$	Función coseno hiperbólico
`tanh(x)`	$\mathrm{Th}(x)$	Función tangente hiperbólica
`exp(x)`	e^x	Función exponencial
`log(x)`	$\ln(x)$	Función logaritmo natural $x > 0$
`log10(x)`	$\log_{10}(x)$	Función logaritmo en base 10 con $x > 0$
`pow(x,y)`	x^y	Función potencial
`sqrt(x)`	$\sqrt{x}$	Función raíz cuadrada positiva con $x \geqslant 0$
`ceil(x)`		Función que devuelve el menor entero no menor que x (*double*)
`floor(x)`	$[x]$	Función *parte entera*: devuelve el mayor entero no mayor que x (*double*)
`fabs(x)`	$\lvert x \rvert$	Función *valor absoluto*

Cuadro 3.9 – *Funciones matemáticas*

3.8. Problemas

1. Realiza un programa que muestre, para cada *tipo* definido en el capítulo, el número de *bytes* que tiene asignado, junto con el rango y la precisión. Por ejemplo, parte de la salida podría ser:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 12345
**************************
        ..........
        ...........
        ..........
El tipo float ocupa 4 bytes
El rango de sus valores es de 1.17549e-038 a 3.40282e+038
Su precisión es de 6 cifras significativas
    ....................
     ....................
```

2. Escribe un programa que dado un carácter, obtenga su correspondiente código ASCII en decimal, octal y hexadecimal. Un ejemplo de ejecución del código podría ser:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 12345
**************************

Introduce un carácter -> c
El carácter c tiene el código ASCII: 99 decimal, 143 octal, 63 hexadecimal
```

Seguidamente, haz otro programa que ejecute el proceso inverso esto es, que se pueda introducir un valor entre 0 y 255 y obtenga el correspondiente carácter. Analiza si existe alguna diferencia entre utilizar **`char`** y **`unsigned char`**.

3. Realiza un programa que determine que para **C**, cualquier valor distinto de 0 es *verdadero*, pero si es 0, entonces es *falso*[11].

4. La relación que transforma los *grados centígrados* a *grados Fahrenheit* es

$$F = \frac{9}{5}C + 32$$

Se pide que hagas un programa que pida *grados centígrados* y los transforme en *grados Fahrenheit*. Haz otro programa que realice el proceso inverso.

5. El *Índice de Masa Corporal (IMC) o Body Mass Index*:

$$\text{IMC} = \frac{\text{Peso en Kg.}}{\text{Altura en metros}^2}$$

da una estimación sobre el peso correcto según la altura. Se pide que realices un programa en que dados un peso y una altura, se calcule el IMC. Una posible ejecución podría ser:

[11] Una posible solución puede encontrarse en el código 4.1 (pág. 69). No obstante, el objetivo es que el alumno trabaje este ejercicio, antes de consultar la solución.

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 12345
**************************

Introduce tu peso en Kg. -> 70.2
Introduce tu altura en Mts. -> 1.73

Tu Índice de Masa Corporal es 23.46
```

6. Escribe un programa en el que se introduzcan las coordenadas cartesianas de un punto del plano y devuelva por pantalla su distancia al origen.
(**Nota:** se pueden utilizar las funciones matemáticas del cuadro 3.9, pág. 62).

7. Escribe un programa en el que se introduzcan las coordenadas de un punto y devuelva sus coordenadas polares.
(**Nota:** se pueden utilizar las funciones matemáticas del cuadro 3.9, pág. 62).

8. Dada la siguiente instrucción

```
while(( sal = scanf("%i %i", &ent1, &ent2 ) ) != 2){
        ..........
        BUCLE
        ..........
    }
```

a) Indica qué es lo que hace.

b) ¿Cuándo se ejecuta el bucle?.

9. Haz un programa que lea un entero *sin signo* y lo imprima, pero debe comprobar que la entrada es correcta. Esto es, si se introduce un carácter, ha de tenerse la posibilidad de repetir la entrada, sin detener la ejecución del programa. Un posible ejemplo sería:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 12345
**************************

Introduce un entero sin signo-> x

Disculpa, debes introducir un ENTERO SIN SIGNO

Introduce un entero sin signo-> 134

El entero sin signo introducido es 134
```

¿qué sucede si introduces un entero *con signo*? ¿y un valor con decimales?

10. Siguiendo con la idea del problema anterior, haz un programa que lea dos números decimales y los imprima (por supuesto, comprobando que la entrada es correcta). A continuación que el programa lea dos caracteres (con comprobación) y que los imprima. Por último, que lea un entero (también con comprobación) y que lo imprima.

11. Utilizando las funciones **`getchar(), putchar()`** y la instrucción **`while`** ¿serías capaz de hacer un programa que leyese un texto del teclado y lo imprimiese en la pantalla? ¿y además que contase el número de caracteres introducido[12]?.

12. Sabiendo el tamaño de bytes de memoria que reserva un tipo **`unsigned long long`**, haz un programa que realice una tabla con los factoriales del 1 al 25 y se almacene una una variable de este tipo. El resultado debe volcarse en un fichero llamado `Datos.dat`. ¿Qué observas del resultado?. Repite el mismo programa pero almacenando los factoriales en una variable decimal **`double`**. Compara los resultados, ¿puedes sacar alguna conclusión?

[12] De nuevo, una orientación sobre la solución puede encontrarse en el código 4.7 (pág. 75). No obstante, el objetivo es que el alumno trabaje este ejercicio, antes de consultar la solución.

CAPÍTULO 4

Control del flujo

4.1. Introducción

Lo que hace distinto un ordenador de una potente calculadora (no programable), es que el ordenador puede soportar programas que sean capaces de *tomar decisiones* o *repetir una secuencia de órdenes*. Esto se realiza en los lenguajes de programación mediante las ***sentencias de control***, que deben presentar tres características básicas:

- Ejecución de una serie de sentencias.
- Repetición de una secuencia de sentencias hasta que se cumpla una determinada condición.
- Empleo de un test para decidir entre distintas acciones alternativas.

En este capítulo, se van a tratar las *sentencias de control* más habituales en el **lenguaje C**. Se comenzará por aquellas sentencias de control de flujo que suelen ser más frecuentes, para continuar con aquellas que permiten elaborar programas más complejos.

4.2. Sentencias básicas

4.2.1. Sentencia `if`

Se trata de una sentencia de tipo *condicional*. Existen diferentes modelos, dependiendo de la complejidad de las opciones de la decisión, que se planteen en el programa.

`if` Es la opción más sencilla. Ya se introdujo en la sección 2.3, en el código 2.2 (pág. 15). Permite ejecutar una instrucción simple o un conjunto de ellas, según se cumpla o no una determinada condición. Esta sentencia tiene la siguiente expresión general

```
if(expresion) sentencia;
```

o bien

```
if(expresion) {
   sentencia;
   sentencia;
        .
        .                          /* <--- BLOQUE IF  */
        .
  sentencia;
}
```

Interpretación: se evalúa la **condición**. Si el resultado es *verdadero* ($\Longleftrightarrow \neq 0$), entonces se ejecuta la sentencia o sentencias que se encuentran entre las llaves, en el BLOQUE IF. Si es *falso* ($\Longleftrightarrow = 0$) entonces prosigue en la siguiente línea ejecutable después de la llave: }, del BLOQUE IF.

if...else Esta sentencia permite realizar una *bifurcación*, ejecutando una parte u otra según se cumpla o no cierta condición. Este tipo de condición ya se utilizó en el código 2.2 (pág. 15) y en el código 2.3 (pág. 19) La expresión toma la forma:

```
if (condición) {
     sentencia;
          .
          .             /*  <--- BLOQUE IF */
          .
     sentencia;
} else {
     sentencia;
          .
          .           /*  <--- BLOQUE ELSE  */
          .
     sentencia;
}
```

Interpretación: se evalúa la **condición**. Si el resultado es *verdadero* ($\Longleftrightarrow \neq 0$), entonces se ejecutan las sentencias que se encuentran entre llaves en el BLOQUE IF y una vez terminados, la siguiente sentencia ejecutable ***después*** del BLOQUE ELSE. Si el resultado es *falso* ($\Longleftrightarrow = 0$), entonces se ejecutan las sentencias del BLOQUE ELSE y una vez terminadas, sigue con las siguientes sentencias ejecutables.

if...else múltiple Permite seleccionar varias opciones. Su estructura es de la forma

```
if (condición_1){
     sentencia;
          .
          .             /*  <--- BLOQUE IF */
          .
     sentencia;
} else if (condición_2) {
     sentencia;
          .
          .           /*  <--- BLOQUE ELSE IF */
          .
     sentencia;
} else if (condición_3) {
```

```
            sentencia;
                .
                .              /*  <--- BLOQUE ELSE IF */
                .
            sentencia;
        } else {
            sentencia;
                .
                .            /*  <--- BLOQUE ELSE  */
                .
            sentencia;
        }
```

Interpretación: se evalúa la **condición_1**. Si el resultado es *verdadero* ($\iff \neq 0$), entonces se ejecutan las sentencias que se encuentran entre llaves en el BLOQUE IF y una vez terminados, la siguiente sentencia ejecutable ***después*** del BLOQUE ELSE. Si el resultado es *falso* ($\iff = 0$), entonces evalúa la **condición_2**. Si el resultado es *verdadero* ($\iff \neq 0$), entonces se ejecutan las sentencias que se encuentran entre llaves en el BLOQUE ELSE IF y una vez terminados, la siguiente sentencia ejecutable ***después*** del BLOQUE ELSE. Así, sucesivamente. Si todas las *condiciones* de los BLOQUES ELSE IF han sido *falsas*, entonces ejecuta las sentencias del BLOQUE ELSE, y una vez acabado, continua con la siguiente sentencia ejecutable.

Ejemplos

Cuando se estudiaron los *operadores de relación* (véase subsección 3.4.5, pág. 54) ya se comentó que para el **lenguaje C** todo lo que no sea cero es *verdad*. En efecto, esto se puede justificar a partir del siguiente código

Código 4.1 – *Función `if`*

```c
#include <stdio.h>
/* Ejemplo de sentencias de control */
int main(void)
{
      int positivo, negativo;

      positivo = 247;
      if (positivo){
          printf("El número %d indica VERDAD para C\n", positivo);
      } else {
          printf("El número %d indica FALSO para C\n", positivo);
      }

      negativo = -392;
      if (negativo){
          printf("El número %d indica VERDAD para C\n", negativo);
      } else {
          printf("El número %d indica FALSO para C\n", negativo);
      }

      if (0){
          printf("El número 0 indica VERDAD para C\n");
      } else {
          printf("El número 0 indica FALSO para C\n");
```

```
25          }
26
27          return 0;
28  }
```

con el resultado de

```
El número 247 indica VERDAD para C
El número -392 indica VERDAD para C
El número 0 indica FALSO para C
```

Este ejemplo se puede repetir para cualesquiera números distintos de cero, y el resultado será el mismo. Por tanto, cualquier valor distinto de cero es *verdad* para **C**. En el siguiente ejemplo se codifica en **C**, la función

$$f(x) = \begin{cases} -x & \text{si } x < 0 \\ x^2 & \text{si } 0 \leqslant x < 1 \\ \text{sen}(\pi \cdot x) & \text{si } 1 \leqslant x \end{cases}$$

El programa sería:

Código 4.2 – *Función `if` múltiple*

```
1   #include <stdio.h>
2   #include <math.h>
3   /* Ejemplo de sentencias de control */
4   #define PI 3.1415926536
5   int main(void)
6   {
7         double numero, resultado;
8
9         FILE *fi;
10
11        fi = fopen("Datos.dat", "w");
12
13        printf("Introduce un numero -> ");
14        scanf("%lf", &numero);
15
16        if (numero < 0){
17            resultado = - numero;  /* BLOQUE 1 */
18        } else if (numero < 1){
19             resultado = numero * numero;  /* BLOQUE 2 */
20        } else {
21             resultado  = sin(PI * numero);  /* BLOQUE 3 */
22        }
23        fprintf(fi, "El resultado para el valor %f es %f\n", numero, resultado);
24
25        fclose(fi);
26
27        return 0;
28  }
```

En este caso, el argumento de la función se introduce por pantalla y según su valor, se distribuye por los distintos bloques. Debido al funcionamiento del **`if` *múltiple***, no hace falta especificar la condición $0 \leqslant x < 1$, sino

simplemente $x < 1$, ya que si es menor que 0, ejecuta el BLOQUE 1. En el BLOQUE 3 se utiliza la función *seno* tal y como la identifica el **C**. Para obtener una descripción más completa de las funciones matemáticas que soporta el **lenguaje C**, se recomienda consultar el libro de Kernighan y Ritchie [9]. En la variable `resultado`, se almacena el valor de la función que se escribe en un fichero con el nombre `Datos.dat`, junto con el valor introducido.

4.2.2. Operador condicional

El *operador condicional* plantea una manera compacta de escribir:

```
if (condición)
   sentencia1;
else
   sentencia2;
```

La forma general de este operador es:

```
condición ? sentencia1 : sentencia2;
```

Interpretación: se evalúa la **condición**. Si el resultado es *verdadero* ($\Longleftrightarrow \neq 0$), entonces se ejecuta la **sentencia1** y una vez realizada, salta a la siguiente línea que tenga una sentencia ejecutable. Si el resultado es *falso* ($\Longleftrightarrow = 0$), entonces se ejecuta la **sentencia2** y una vez terminada, sigue con las siguientes líneas del programa. Veamos un ejemplo,

Código 4.3 – *Operador condicional*

```
#include <stdio.h>
/* Ejemplo de operador condicional */
#define ABS(X) ((X) < 0 ? -(X) : (X))
int main(void)
{

      double a, b, c, max;

      FILE *fi;

      fi = fopen("Datos.dat", "w");

      printf("Introduce un numero -> ");
      scanf("%lf", &a);

      printf("\n Introduce otro numero -> ");
      scanf("%lf", &b);

      max = (a > b) ? a : b;
      fprintf(fi, "El máximo de %f y %f es %f \n", a, b, max);

      printf("\n Introduce un numero mas -> ");
      scanf("%lf", &c);

      fprintf(fi, "El valor absoluto de %f es %f \n", c, ABS(c));
      fclose(fi);

```

```
28        return 0;
29  }
```

Para los valores introducidos por el teclado de: 3.467, -247.23 y -9 respectivamente, el resultado escrito en el fichero `Datos.dat` es

```
El máximo de 3.467000 y -247.230000 es 3.467000
El valor absoluto de -9.000000 es 9.000000
```

Este ejemplo tiene la particularidad de utilizar el *operador condicional* para definir dos funciones de maneras distintas. La primera, para la función ABS, se establece mediante una *macro* por medio de la *directiva* `#define`, para que sea manipulada directamente por el *preprocesador*. La segunda forma, se plantea en la LÍNEA 19 para definir la función *max*.

4.2.3. Bucle `while`

El bucle **`while`** permite ejecutar una sentencia o un conjunto de sentencias, *mientras se cumpla una determinada condición*, de manera repetitiva. La forma general es

```
    while (condición)
        sentencia;
```

en el caso de una única sentencia o entre llaves, en caso de un conjunto de sentencias

```
    while (condición){
         sentencia;
              .
              .                        /* <--- BLOQUE WHILE  */
              .
         sentencia;
    }
```

Interpretación: se evalúa la **`condición`**. Si es *falsa* ($\iff = 0$) entonces prosigue en la siguiente línea ejecutable después de la llave: }, del BLOQUE WHILE. Si el resultado es *verdadero* ($\iff \neq 0$), entonces se ejecuta la sentencia o sentencias que se encuentran entre llaves, en el BLOQUE WHILE. Una vez acabada con la última sentencia de este BLOQUE, se vuelve a comprobar la **`condición`**, si es *verdadero*, se vuelve a repetir la ejecución del *bloque* y una vez terminado se comprueba la **`condición`** nuevamente y así sucesivamente, hasta que la **`condición`** sea *falsa*, que entonces, el control del programa salta el BLOQUE WHILE y continúa con la siguiente sentencia ejecutable. Para que esta estructura tenga sentido, se suele poner algún criterio dentro del bucle, que actualice la **`condición`** en cada ejecución del bucle.

Ya se ha empleado la instrucción **`while`** en diversos ejemplos anteriores y se debe tener una idea clara de su funcionamiento. Sin embargo, resulta interesante su utilización con operadores *incrementales* (véase subsección 3.4.4, pág. 52) y con los operadores de *asignación compuestos* (véase subsección 3.4.3, pág. 52). Para ilustrar su utilización de una manera clara, se cambia parte del código 2.3 (pág. 19). En este caso, se trata de calcular la suma de los 100 primeros números naturales y su resultado se imprime en un fichero llamado `Datos.dat`. La modificación, realizada para utilizar los operadores de asignación compuestos, se observan en las LÍNEAS 17–20 del programa 4.4

Código 4.4 – *Bucle `while`*

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

```c
/* Ejemplo del bucle while */

#define TOPE 100

int main(void)
{
      int suma = 0, numero;
      int suma_Gauss = 0;

      FILE *fi;

      fi = fopen("Datos.dat", "w");

      numero = 1;

      while (numero <= TOPE){
            suma +=  numero;  /* suma = suma + numero */
            numero++;
      }

      fprintf(fi, "La suma de los %d  ", TOPE);
      fprintf(fi, " primeros números es %d\n\n", suma);
      fprintf(fi,"Veamos si me he equivocado: ");
      fprintf(fi,"voy a aplicar la regla de Gauss");

      suma_Gauss = TOPE * (TOPE + 1) / 2;
      fprintf(fi, "\n \n suma_Gauss es %d", suma_Gauss);
      if (suma == suma_Gauss)
               fprintf(fi, "\t Bien !!!! sé sumar");

      else
               fprintf(fi, "\t Vaya !!!! fallé, tengo que repasar");

      fclose(fi);

      return 0;
}
```

La salida en el fichero `Datos.dat`, sería:

```
La suma de los 100   primeros números es 5050

Veamos si me he equivocado: voy a aplicar la regla de Gauss

 suma_Gauss es 5050      Bien !!!! sé sumar
```

Es habitual que a los programadores de **C** les guste compactar el código (es lógico, pues es un lenguaje creado con esta intención) y por tanto, es frecuente reescribir el bucle de la siguiente manera:

```c
      while (numero++ <= TOPE)
            suma +=  numero;  /* suma = suma + numero */
```

es decir, añadir el *operador incremento* a la **`condición`**. Sin embargo, con este pequeño cambio el resultado sería el siguiente:

```
La suma de los 100   primeros números es 5150

Veamos si me he equivocado: voy a aplicar la regla de Gauss

 suma_Gauss es 5050      Vaya !!!! fallé, tengo que repasar
```

¿Qué es lo que falla? En la LÍNEA 15 del código 4.4 se realiza la *inicialización* de la **condición** del bucle `while`, con la asignación del valor 1 a la variable `numero`. La variable `numero` toma entonces el valor 1, pero con el *operador incremento* aumenta en una unidad, *después* de comparar con el valor de `TOPE`. Con lo cual, entra con el valor de 1, se compara con `TOPE`, una vez hecha esta comparación, se incrementa en una unidad, en este caso 2 y con este valor entra en el bucle. Así pues, se calcula la suma desde 2 al 101, obteniéndose 5150. Para tratar de remediar la situación, se sustituye la LÍNEA 15 por

```
      entero = 0;
```

Se ejecuta de nuevo el código 4.4 y la respuesta de nuevo es fallida:

```
La suma de los 100   primeros números es 5151

Veamos si me he equivocado: voy a aplicar la regla de Gauss

 suma_Gauss es 5050      Vaya !!!! fallé, tengo que repasar
```

Ahora la razón está en que el *operador incremental* tiene una expresión *sufija*. Esto quiere decir, como ya se ha comentado, que primero compara con el valor de `TOPE` y *después* realiza el incremento. Al comienzo, el proceso es correcto, pero cuando llega `numero` al valor `TOPE`, entonces comprueba que son iguales los valores y da paso a ejecutar el bucle. De manera inmediata aumenta en una unidad el valor `numero`, con lo cual entra en el bucle con el valor de `TOPE + 1`. Es decir, se está calculado, realmente la suma desde 1 hasta 101, esto es, 5151. El siguiente paso sería intercambiar la expresión *sufija* por la *prefija*, en el *operador incremento*, pero manteniendo el valor inicial de `entero` a cero:

```
      while (++numero <= TOPE)
            suma +=  numero;  /* suma = suma + numero */
```

con este procedimiento, se hace primero la comparación de la variable `numero` con el valor de `TOPE` y *después* se realiza el incremento de la variable `numero`. Así pues, la salida ya es correcta

```
La suma de los 100   primeros números es 5050

Veamos si me he equivocado: voy a aplicar la regla de Gauss

 suma_Gauss es 5050      Bien !!!! sé sumar
```

A la vista de estos ejemplos, hay que tener cierta precaución al utilizar los *operadores incrementales y/o decrementales* junto con el bucle `while`. Puesto que esta situación suele ser fuente de numerosos errores no detectados por el compilador, tal vez merece la pena, insistir un poco más en ello. Sean los ejemplos

Código 4.5 – *Operador incremental sobre bucles* `while`

```
1 #include <stdio.h>
2 /* Ejemplo de operaciones incrementales y bucle while  */
3 int main(void)
4 {
```

```
5         int entero = 0, sal;
6
7
8         while(++entero < 3){
9            sal = 3*entero + 2;
10           printf("entero -> %6d, sal -> %6d\n", entero, sal);
11        }
12
13        return 0;
14  }
```

cuya salida es

```
entero ->      1, sal ->      5
entero ->      2, sal ->      8
```

mientras que el código siguiente

Código 4.6 – *Operador incremental sobre bucles* `while`

```
1   #include <stdio.h>
2   /* Ejemplo de operaciones incrementales y bucle while */
3   int main(void)
4   {
5         int entero = 0, sal;
6
7
8         while(entero++ < 3){
9            sal = 3*entero + 2;
10           printf("entero -> %6d, sal -> %6d\n", entero, sal);
11        }
12
13        return 0;
14  }
```

genera una salida de la forma

```
entero ->      1, sal ->      5
entero ->      2, sal ->      8
entero ->      3, sal ->     11
```

Una vez más, se observa que el procedimiento *sufijo* del código 4.6 obliga a que se realice una vez más el *bucle*, que con el operador en forma *prefija* del código 4.5. En el programa 4.6, incrementa la variable `entero` **después** de comprobar si es menor que 3, mientras que el código 4.5, lo incrementa **antes** de comprobar si es menor que 3. Así pues, los *operadores incrementales y decrementales* resultan muy prácticos a la hora de programar, pero hay que utilizarlos con cierta cautela porque pueden producir serios *quebraderos de cabeza*.

Debido a la estructura del bucle **`while`** es posible operar fácilmente con caracteres utilizando las funciones **`getchar()`** y **`putchar()`** (véase subsección 3.3.3 en la pág. 47). En el siguiente ejemplo, el usuario escribe una frase y el programa reproduce la frase y cuenta el número de caracteres:

Código 4.7 – *Bucle* `while` *y funciones* `getchar()` *y* `putchar()`

```
1   #include <stdio.h>
2   #define PARA '\n'
```

```
3  int main(void)
4  {
5        int contador = 0;
6        char car;
7
8        printf("Teclea una frase \n\n");
9        while ((car = getchar()) != PARA){
10             putchar(car);
11             contador++;
12       }
13       printf("\n He leído un total de %d caracteres. \n", contador);
14
15       return 0;
16 }
```

Una vez que el usuario escribe una frase en la pantalla y teclea ENTER, los caracteres almacenados en el *buffer* de entrada se procesan y se obtiene una copia de la frase, en la pantalla. La 'tecla ENTER' está almacenada en la constante `PARA`, definida mediante la *directiva* `#define`. Para la frase

```
Me gusta aprender C
```

en resultado en pantalla sería

```
Teclea una frase

Me gusta aprender C
Me gusta aprender C

 He leído un total de 19 caracteres
```

El código 4.7 puede *compactarse* (como tanto les gusta a los programadores de **C**) sustituyendo las LÍNEAS 9–12 por una expresión más *profesional*

```
       while (putchar(getchar()) != PARA)
             contador++;
```

Para evitar que la frase aparezca dos veces en la pantalla, se puede redireccionar la segunda frase a un fichero. Esto se consigue con la función **`putc()`**. Es similar a la función `putchar()` pero ahora hay que añadir, como segundo argumento, el *puntero* al fichero. El programa anterior, pero con la salida redirigida al fichero `Datos.dat`, sería:

Código 4.8 – *Bucle `while` y funciones `getchar()` y `putchar()` con salida redirigida a un fichero*

```
1  #include <stdio.h>
2  /* Ejemplo para el bucle while y salida a fichero */
3  #define PARA '\n'
4  int main(void)
5  {
6        int contador = 0;
7        FILE *fi;
8
9        fi = fopen("Datos.dat", "w");
10
11       printf("Teclea una frase \n\n");
```

```
12        while (putc(getchar(), fi) != PARA)
13              contador++;
14        fprintf(fi,"\n He leído un total de %d caracteres. \n", contador);
15
16
17        fclose(fi);
18        return 0;
19  }
```

En la LÍNEA 12 se observa la utilización de la función **putc()**. La salida obtenida en el fichero `Datos.dat`, una vez escrita en pantalla la frase:

```
Quiero seguir aprendiendo C
```

sería:

```
Quiero seguir aprendiendo C

 He leído un total de 27 caracteres.
```

De la misma manera que existe una función **putc()**, existe una función **getc()** que permite leer un carácter de un fichero. La ventaja de la utilización de esta función para leer caracteres de un fichero es que no se detiene cuando encuentra un *salto de línea* (‘\n’), frente a la función **getchar()**, que sí lo hacía. La función se detiene cuando encuentra el indicativo *final de fichero* o **EOF** (véase sección 3.6, pág. 59). Como sucedía con **putc**, en la función **getc** hay que introducir el *puntero* a fichero, como argumento.

Si se quiere realizar un programa que lea un texto de un fichero (con *saltos de línea*, incluidos) y lo imprima en otro fichero distinto, un posible fragmento utilizando las funciones **putc** y **getc**, sería

```
1  while((c = getc(fent)) != EOF){
2       putc(c, stdout);  //Salida en pantalla
3       putc(c, fsal);
4       cont++;
5  }
```

En la LÍNEA 1, la función **getc** lee caracteres del fichero asociado al *puntero*: `fent`, hasta que encuentre el *final de fichero*, representado con **EOF**. Esta construcción permite que se sigan leyendo caracteres, incluso si hay *saltos de línea* en el fichero de entrada. En la LÍNEA 2, la función **putc** imprime el carácter en la salida establecida *por defecto*. Esta salida, el **C** lo identifica a través del nombre **stdout**. Habitualmente es la pantalla. Hay también una entrada *por defecto*, que suele utilizar la función **getc**. Se identifica por **stdin** y por lo general, es el teclado. En la LÍNEA 3, se imprime el carácter en el fichero de salida, que está identificado con el *puntero*: `fsal`. Por último, la variable **cont** es el contador de caracteres, que se va incrementando.

4.2.4. Bucle `for`

Este tipo de bucle tiene una estructura lógica similar al del bucle **while**, pero con la particularidad de agrupar en un sólo lugar cuatro acciones:

1. Valor con el que comienza el bucle (*inicialización*)
2. Condición bajo la cual el bucle continúa. Generalmente consiste en la evaluación de unos operadores de relación (*evaluación*)
3. Cambios de la variable de control o contador (*actualización*)

4. Instrucciones del bucle (*ejecución*).

por tanto, su presentación resulta más compacta. Su forma general es:

```
for (inicialización; condición; actualización)
    sentencia;
```

en el caso de una única sentencia, o bien entre llaves si deben ejecutarse un grupo de sentencias:

```
for (inicialización; condición; actualización){
     sentencia;
          .
          .                    /* <--- BLOQUE FOR  */
          .
     sentencia;
}
```

Interpretación: antes de iniciarse el bucle se ejecuta la **inicialización**, que consiste en una o más sentencias que asignan valores iniciales a ciertas variables o contadores. A continuación se evalúa la **condición**. Si el resultado es *falso* ($\Longleftrightarrow = 0$) entonces prosigue en la siguiente línea ejecutable después del comando `for` o bien, de la llave: } en el caso de tener un BLOQUE FOR. Si el resultado es *verdadero* ($\Longleftrightarrow \neq 0$), entonces se ejecuta la sentencia o sentencias que se encuentran entre llaves, en el BLOQUE FOR. Una vez acabada con la última sentencia de este BLOQUE, se efectúa la **actualización** de la variable de control y se vuelve a comprobar la **condición**. Si es *verdadera*, se repite la ejecución del *bloque* y se vuelven a realizar los procesos de *actualización* y *evaluación de la condición*, así sucesivamente. En el momento que la **condición** sea *falsa*, entonces el control del programa salta el BLOQUE FOR y continúa con la siguiente sentencia ejecutable. Aunque en la **inicialización** se pueden asignar valores a distintas variables, en la **actualización** sólo se puede modificar una. Ésta es la llamada *variable de control* del bucle.

Un ejemplo sencillo para ilustrar su funcionamiento se presenta en el código 4.9, que únicamente imprime las cifras de 3 al 9 mediante una variable *incremento*.

Código 4.9 – *Primer ejemplo con `for`*

```
1  #include <stdio.h>
2  /* Sentencia for */
3  int main(void)
4  {
5        int incremento, comienzo;
6        int valor;
7
8        for(comienzo = 3, incremento = 0; incremento <= 6; incremento++){
9               valor  = comienzo + incremento;
10              printf("Resultado de valor: %d \n", valor);
11       }
12       return 0;
13 }
```

y el resultado que se obtiene sería

```
Resultado de valor: 3
Resultado de valor: 4
Resultado de valor: 5
Resultado de valor: 6
```

```
Resultado de valor: 7
Resultado de valor: 8
Resultado de valor: 9
```

Se observa en la LÍNEA 8, que este comando permite inicializar *a la vez* varias variables, que formarán parte de la ejecución del bucle[1].

Un ejemplo más completo, consiste en sustituir el bucle **`while`** por el **`for`**, en el programa 4.4 (pág. 72), como muestra el código 4.10

Código 4.10 – *Bucle `for`*

```
1  #include <stdio.h>
2  /* Ejemplo del bucle for */
3
4  #define TOPE 100
5  int main(void)
6  {
7
8
9        int suma = 0, numero;
10       int suma_Gauss = 0;
11
12       FILE *fi;
13
14       fi = fopen("Datos.dat", "w");
15
16       for(numero = 1; numero <= TOPE; numero++)
17             suma +=  numero;  /* suma = suma + numero */
18
19       fprintf(fi, "La suma de los %d  ", TOPE);
20       fprintf(fi, " primeros números es %d\n\n", suma);
21       fprintf(fi,"Veamos si me he equivocado: ");
22       fprintf(fi,"voy a aplicar la regla de Gauss");
23
24       suma_Gauss = TOPE * (TOPE + 1) / 2;
25       fprintf(fi, "\n \n suma_Gauss es %d ", suma_Gauss);
26       if (suma == suma_Gauss)
27                fprintf(fi, "\t Bien !!!! Sé sumar");
28
29       else
30                fprintf(fi, "\t Vaya !!!! Fallé, tengo que repasar");
31
32       fclose(fi);
33       return 0;
34 }
```

La salida en el fichero `Datos.dat`, sería:

```
La suma de los 100   primeros números es 5050

Veamos si me he equivocado: voy a aplicar la regla de Gauss
```

[1] A partir de la versión **C99**, en esta parte del bucle **`for`** es posible incluso, *declarar* las variables. Este procedimiento no se seguirá en este curso, pues resulta más claro una *declaración* de **todas** las variables, al principio del programa.

```
suma_Gauss es 5050      Bien !!!! Sé sumar
```

El resultado **NO** varía si en la parte de **actualización** del comando **for**, se sustituye la expresión *sufija* por la *prefija* del *operador incremental*. Esto es, se cambian las LÍNEAS 16-17 por

```
    for(numero = 1; numero <= TOPE; ++numero)
          suma +=  numero;  /* suma = suma + numero */
```

En este caso, la variable `numero` es la *variable de control*.

4.3. Otras sentencias

4.3.1. Sentencia do-while

Tanto el bucle **for** como **while** son bucles con *condición de entrada*: la condición del test se comprueba **antes** de cada iteración del bucle. En **C** existe la posibilidad de un bucle con la *condición de salida*, en el cual se comprueba la condición al final de cada iteración. Se trata del bucle **do-while** que tiene el siguiente formato general

```
    do{
        sentencia;
            .
            .              /*  <--- BLOQUE DO--WHILE */
            .
        sentencia;
    } while (condición);
```

Interpretación: el bucle se ejecuta una vez, como mínimo, ya que el test se realiza tras la ejecución de la iteración. El proceso es similar al bucle **while** anterior. Una vez que se llega a la **condición**, ésta se evalúa. Si el resultado es *falso* ($\iff = 0$) entonces prosigue en la siguiente línea ejecutable. Si el resultado es *verdadero* ($\iff \neq 0$), entonces se vuelve a ejecutar la sentencia o sentencias que se encuentran en el BLOQUE DO–WHILE. Una vez acabada con la última sentencia de este BLOQUE, se vuelve a comprobar la **condición**. Si es *verdadera*, se vuelve a repetir la ejecución del *bloque* y una vez terminado se comprueba la **condición** nuevamente y así sucesivamente, hasta que la **condición** sea *falsa*, que entonces, el control del programa continúa con la siguiente sentencia ejecutable. Para que esta estructura tenga sentido, al igual que sucedía con el bucle **while**, se suele poner algún criterio dentro del BUCLE, que compruebe la **condición**, en cada ejecución del bucle.

A la vista de las diferentes opciones que ofrece el **C** para establecer un bucle, la pregunta clave es ¿qué bucle elegir? En primer lugar hay que de decidir si se necesita un bucle con *condición de entrada o salida*. Lo habitual es que el bucle necesite una *condición de entrada*. Kernighan y Ritchie [9, pág. 70] estiman que la *condición de salida* es mucho menos utilizada. La opinión de este escaso uso es que suele ser mejor, *mirar antes de saltar*. Otra razón de peso es que un programa es más legible, si la condición de test se encuentra al comienzo del bucle. No hay que olvidar que existen aplicaciones en las que es importante que el bucle se evite si el test no se cumple en un primer momento.

En cuanto a la utilización de los bucles **for** y **while**, la cosa no está tan clara. Generalmente, lo que puede hacer uno de ellos, lo puede hacer *con más o menos dificultad*, el otro. Por lo que concierne al estilo, parece más apropiado usar un bucle **for** cuando el bucle implique inicializar y actualizar una variable, tal y como se ha planteado en el programa 4.10 (pág. 79). Sin embargo, para condiciones que no implican casos de incrementos, tal y como sucede con el manejo de caracteres, tal vez sea mejor el bucle **while**, como muestra el código 4.8 (pág. 76)

4.3.2. Sentencias `switch` y `break`

Se trata de una sentencia que permite una selección entre varias posibilidades. La idea es similar al **`if-else` múltiple** anteriormente visto, aunque presenta importantes diferencias. Su aspecto general es el siguiente:

```
switch (expresión){
     case cte_1 :
        sentencia;
           .
           .            /*  <--- BLOQUE  SWITCH 1*/
           .
        sentencia;
     case cte_2 :
        sentencia;
          .
          .           /*  <--- BLOQUE SWITCH 2*/
          .
        sentencia;
     ..................
     ..................
     case cte_n :
        sentencia;
          .
          .            /*  <--- BLOQUE SWITCH n */
          .
        sentencia;
   default :
        sentencia;
          .
          .           /*  <--- BLOQUE DEFAULT  */
          .
        sentencia;
}
```

Interpretación: se evalúa la **`expresión`** y se considera su resultado. Si el resultado coincide con el valor de **`cte_1`** entonces ejecuta la sentencia o sentencias del BLOQUE SWITCH 1 y ¡¡atención!! ...las sentencias siguientes correspondientes a los BLOQUES SWITCH 2,..., n, incluso ejecuta las sentencias del BLOQUE DEFAULT y el control del programa continúa con la siguiente sentencia ejecutable después de la llave } del **`switch`**. En general, se ejecutan las sentencias de todos los BLOQUES SWITCH que están a continuación de la expresión **`cte`** cuyo valor coincide con el obtenido al calcular las operaciones en **`expresión`**. Si este resultado no coincide con el valor de ninguna **`cte`**, entonces se ejecuta el BLOQUE DEFAULT y cuando termina, sigue con la siguiente sentencia ejecutable después de la llave } del **`switch`**.

Veamos el ejemplo de un programa, en el que se introduce una vocal y te contesta cuál es. Si no es vocal, te avisa. Los resultados se guardan en un fichero llamado `Datos.dat`.

Código 4.11 – *Sentencia `Switch`*

```
1 #include <stdio.h>
2 /* Ejemplo de sentencia switch */
3 int main(void)
4 {
5       char ch;
6
```

```
7        FILE *fi;
8
9        fi = fopen("Datos.dat", "w");
10
11       printf("Introduce una vocal (minúscula) -> ");
12
13       while((ch = getchar()) != '@'){
14            if (ch != '\n'){
15               if (ch >= 'a' && ch <= 'z')
16                  switch (ch) {
17                     case 'a' :
18                        fprintf(fi, "Es la vocal: a\n");
19                     case 'e' :
20                        fprintf(fi, "Es la vocal: e\n");
21                     case 'i' :
22                        fprintf(fi, "Es la vocal: i\n");
23                     case 'o' :
24                        fprintf(fi, "Es la vocal: o\n");
25                     case 'u' :
26                        fprintf(fi, "Es la vocal: u\n");
27                     default :
28                        fprintf(fi, "Es una consonante. ");
29                        fprintf(fi, "No me mientas\n \n");
30                  }  /* fin switch */
31                else
32                     fprintf(fi, "Sólo letras minúsculas");
33                printf("Introduce otra vocal o @ para terminar -> ");
34            } /* end if nueva línea*/
35       }  /* end while */
36       fclose(fi);
37
38       return 0;
39   }
```

Si se teclea la letra `a` el resultado sería, tal y como está definido **`switch`**,

```
Es la vocal: a
Es la vocal: e
Es la vocal: i
Es la vocal: o
Es la vocal: u
Es una consonante. No me mientas
```

que no es lo que uno esperaría. Hagamos otra prueba: se teclea la letra `i` y se obtiene

```
Es la vocal: i
Es la vocal: o
Es la vocal: u
Es una consonante. No me mientas
```

y si se teclea `f` el resultado es

```
Es una consonante. No me mientas
```

Todas estas salidas coinciden con la definición dada para la sentencia **`switch`**, pero es claro que **no** se ajusta a lo que desearía de un programa que te devuelve la vocal introducida. Para evitar esta situación se introduce la

sentencia **break**. Esta proposición, ***fuerza la salida*** de la sentencia **switch** y el programa pasa a ejecutar la sentencia a continuación de la llave } del **switch**. Con la introducción de **break** en cada BLOQUE SWITCH, ya el programa se comporta como era de esperar. Esto es, el programa quedaría de la forma:

Código 4.12 – *Sentencia* `Switch`

```
#include <stdio.h>
/* Ejemplo de sentencia switch */
int main(void)
{
      char ch;

      FILE *fi;

      fi = fopen("Datos.dat", "w");

      printf("Introduce una vocal (minúscula) -> ");

      while((ch = getchar()) != '@'){
           if (ch != '\n'){
              if (ch >= 'a' && ch <= 'z')
                 switch (ch) {
                    case 'a' :
                       fprintf(fi, "Es la vocal: a\n");
                       break;
                    case 'e' :
                       fprintf(fi, "Es la vocal: e\n");
                       break;
                    case 'i' :
                       fprintf(fi, "Es la vocal: i\n");
                       break;
                    case 'o' :
                       fprintf(fi, "Es la vocal: o\n");
                       break;
                    case 'u' :
                       fprintf(fi, "Es la vocal: u\n");
                       break;
                    default :
                       fprintf(fi, "Es una consonante. ");
                       fprintf(fi, "No me mientas\n \n");
                 }  /* fin switch */
               else
                    fprintf(fi, "Sólo letras minúsculas");
               printf("Introduce otra vocal o @ para terminar -> ");
           } /* end if nueva línea*/
      }  /* end while */
      fclose(fi);

      return 0;
}
```

de esta manera, si se teclea la letra a, ya se obtiene lo *esperado*.

```
Es la vocal: a
```

Se teclea la `i` y se obtiene

```
Es la vocal: i
```

y por último si se teclea `f` aparece

```
Es una consonante. No me mientas
```

que es lo lógico.

La sentencia **`break`** también proporciona una salida anticipada de un **`for`**, **`while`** y **`do-while`**, tal y como lo hace en el **`switch`**.

4.3.3. Sentencias `continue y goto`

La sentencia **`continue`** es utilizable en los tres tipos de bucles, pero no en el **`switch`**. Al igual que el **`break`** interrumpe el flujo del programa, el comando **`continue`** también interrumpe el bucle, pero la diferencia está en que evita salir completamente del bucle y fuerza, el comienzo de una nueva iteración.

En principio no es necesario la utilización de la sentencia **`goto`** en **C**, y por tanto no se empleará en este documento. Su presencia se debe a la compatibilidad con otros lenguajes de programación como el FORTRAN O BASIC. Para más información véanse los libros de Rodríguez–Losada [18] o Kernighan y Ritchie [9].

4.4. Números aleatorios

4.4.1. Introducción

Cuando se habla de números aleatorios es habitual entender que se hace referencia a una ***secuencia de números sin orden aparente***. Su utilización es frecuente en los procesos de simulación. Por tanto, resulta muy importante diseñar algoritmos computacionales que permitan generar este tipo de números. No obstante, es necesario destacar que los procesos computacionales son deterministas, lo que implica que no generan números aleatorios propiamente dichos, y por esta razón se les suele llamar números *pseudo-aleatorios*. Es decir, que a partir de un cierto valor fijo inicial, siempre generan la misma secuencia de números.

Un procedimiento para generar números *pseudo-aleatorios* es el ***método congruencial lineal***. Consiste en generar una sucesión de la forma

$$n_{i+1} = \Big[(\, a \cdot n_i + b \,) \mod(m) \Big] \qquad \text{con } i = 0, 1, 2, \ldots \tag{4.1}$$

Los números a, b y m deben ser números enteros positivos, que se denominan respectivamente *multiplicador, incremento* y *módulo*. El valor inicial n_0 es la ***semilla*** y debe ser un dato inicial conocido.

Una objeción clara a este método y en general a todos los procesos de generación de números *pseudo-aleatorios* es que no genera números aleatorios en absoluto. La idea en este caso, es tomar unos valores adecuados para a, b y m de manera que obtención de los números sea lo más *aleatoria* posible. Una condición habitual es tomar m suficientemente grande con $m > a, b, n_0$.

4.4.2. Números aleatorios en C

En **C**, la función `rand()` genera números aleatorios enteros, en el intervalo $[0, \texttt{RAND\_MAX}]$. La constante `RAND_MAX`, representa la cota superior de números *pseudo-aleatorios* que puede generar el compilador. Tanto

esta función como la constante, se encuentran definidas en el fichero `stdlib.h`, que hay que poner en el *encabezamiento* del programa. En el **código** 4.13, se muestra un ejemplo de generación de números *pseudo–aleatorios*, con la función `rand()`.

Código 4.13 – *Ejemplo de generación de números aleatorios*

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

int main (void){


      int numero, NumMax, cont;

      FILE *fi;

      fi = fopen("Salida.dat", "w");

      printf("Introduce la cantidad de números aleatorios -> ");
      scanf("%i", &NumMax);

      fprintf(fi, "Los %i numeros aleatorios son: \n\n", NumMax);
      for (cont = 1; cont <= NumMax; cont++){
          numero = rand();
          fprintf(fi, "%i ", numero);
      }
      fprintf(fi, "\n\n");
      fprintf(fi, "******************\n\n");

      fprintf(fi, "El rango del numero aleatorio es %lli", RAND_MAX);

      fclose(fi);

      return 0;
}
```

y cuya salida en el fichero **`salida.dat`** después de haber introducido en `NumMax` el valor de 10, sería la siguiente

```
Los 10 numeros aleatorios son:

41 18467 6334 26500 19169 15724 11478 29358 26962 24464

******************

El rango del numero aleatorio es 8519976755933511679
```

Los números obtenidos a través de la función `rand()` siguen una distribución *uniforme* ($X \rightsquigarrow \mathcal{U}[0, \texttt{RAND\_MAX}]$), con la idea que todos los números enteros que se generen en el intervalo $[0, \texttt{RAND\_MAX}]$ tengan la misma probabilidad.

Es importante fijarse que en el programa anterior no ha sido necesario introducir ningún valor para la *semilla*. El compilador de **C** ya proporciona una predefinida. Esto implica que si no se cambia, se generará siempre, la ***misma secuencia*** de números *pseudo–aleatorios*. Sin embargo, el **C** permite modificar el valor de la semilla a través de la función `srand`, que como argumento entra un valor entero, que es la *semilla* para

proceso de generación. En el **código** 4.14, se muestra un ejemplo de utilización de esta función, junto con la generadora `rand()`. La salida se escribe en el fichero **`salida.dat`**.

Código 4.14 – *Ejemplo de generación de números aleatorios*

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

int main (void){


        int numero, NumMax, cont;
        int semilla;

        FILE *fi;

        fi = fopen("Salida.dat", "a");

        printf("Introduce la cantidad de números aleatorios -> ");
        scanf("%i", &NumMax);

        printf("Introduce el valor para la semilla -> ");
        scanf("%i", &semilla);

        srand ( semilla );

        fprintf(fi, "Los %i numeros aleatorios para la semilla %i son: \n\n",
           NumMax, semilla);
        for (cont = 1; cont <= NumMax; cont++){
            numero = rand();
            fprintf(fi, "%i ", numero);
        }
        fprintf(fi, "\n\n");
        fprintf(fi, "******************\n\n");

        fclose(fi);

        return 0;
}
```

y cuya salida en el fichero **`salida.dat`** después de haber ejecutado varias veces el programa e introduciendo en `NumMax` el valor de 10, sería la siguiente

```
Los 10 numeros aleatorios para la semilla 2 son:

45 29216 24198 17795 29484 19650 14590 26431 10705 18316

******************

Los 10 numeros aleatorios para la semilla 2 son:

45 29216 24198 17795 29484 19650 14590 26431 10705 18316

******************

Los 10 numeros aleatorios para la semilla 5 son:
```

```
54 28693 12255 24449 27660 31430 23927 17649 27472 32640

******************
```

Como se observa en la salida, si el valor de la semilla no cambia (`semilla = 2`), la sucesión de números *pseudo–aleatorios* coincide.

Si se quiere introducir un valor de la semilla, que no dependa del usuario, una posibilidad es que obtenga este valor del reloj del sistema, a través de la función `time`. La definición de esta función está contenida en el fichero `time.h` y por tanto hay que especificarla en el encabezamiento del programa. La utilización de este tipo de semilla se hace mediante la instrucción

```
srand( time(NULL) );
```

Números enteros

Generalmente se necesita una secuencia de números enteros *pseudo–aleatorios* dentro de un intervalo dado. Para ello es emplea el *resto de la división entera*. Por ejemplo, si se desea una sucesión de números *pseudo–aleatorios* en el intervalo $[1, 10]$ se podría operar de la forma:

```
numero = rand() % 10 + 1;
```

En general si se desea una sucesión de números *pseudo–aleatorios* en el intervalo $[M, N]$, se plantearía la operación

```
numero = rand() % (N - M + 1) + M;
```

Numeros reales

En ciertas ocasiones también es necesario generar una sucesión de números *pseudo-aleatorios*, pero reales. Una manera sencilla de obtener números *pseudo–aleatorios* en el intervalo $[0, 1]$ es dividir el resultado por la constante `RAND_MAX`. Esto es, una operación del tipo

```
numero =  (double) rand() / RAND_MAX;
```

Si además se quisiera obtener los números *pseudo–aleatorios* reales en el intervalo $[a, b]$, bastaría con las operaciones

```
numero =  (double) rand() / RAND_MAX;
numero = numero * (b-a) + a;
```

4.4.3. Números aleatorios con distribución Normal en C

Los números aleatorios generados hasta ahora siguen una distribución *Uniforme* ($X \rightsquigarrow \mathcal{U}[0, \texttt{RAND\_MAX}]$), como ya se ha comentado. En ciertas ocasiones, resulta de interés generar números aleatorios con una distribución *Normal* ($X \rightsquigarrow \mathcal{N}(0, 1)$). La obtención de números con esta distribución no es directa en **C**. Es necesario realizar algún tipo de transformación a partir de números números obtenidos, previamente con una distribución *Uniforme*.

Una de estas trasformaciones es la llamada Box-Muller (véase Press *et al* [16, pág. 287]) o método Polar que consiste en que a partir de dos v.a. uniformes $U_1 \rightsquigarrow \mathcal{U}(0,1)$ y $U_2 \rightsquigarrow \mathcal{U}(0,1)$ se generan dos v.a. Normales $X \rightsquigarrow \mathcal{N}(0,1)$ e $Y \rightsquigarrow \mathcal{N}(0,1)$ independientes, mediante las operaciones

$$X = \sqrt{-2\log U_1}\,\cos(2\pi U_2) \tag{4.2a}$$

$$Y = \sqrt{-2\log U_1}\,\operatorname{sen}(2\pi U_2) \tag{4.2b}$$

La justificación de este procedimiento se basa en considerar dos v.a.'s que sigan una distribución Normal estándar y que sean independientes. En este caso la función de distribución conjunta de ambas variables vendrá dada por:

$$f(x,y) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}x^2}\,\frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}y^2}$$

pasando a coordenadas polares y teniendo en cuenta el valor del *jacobiando de la tranformación* resulta la función de densidad

$$g(r,\theta) = r f(x,y) = \frac{1}{2\pi} r e^{-\frac{1}{2}r^2}$$

Esta función se descompone en producto de otras dos funciones

$$g(r,\theta) = g_\theta(\theta) g_r(r)$$

donde

$$g_\theta(\theta) = \frac{1}{2\pi} \qquad \text{y} \qquad g_r(r) = r e^{-\frac{1}{2}r^2}$$

El siguiente paso es conseguir unos valores aleatorios para cada una de las funciones. En el caso de g_θ resulta inmediato, pues se genera una distribución *Uniforme* en el intervalo $[0, 2\pi]$, con valor $\frac{1}{2\pi}$. El segundo caso no es tan inmediato, pero es posible definir la función

$$F(r) = \int_0^r t^{-\frac{1}{2}t^2}\,dt = 1 - e^{-\frac{1}{2}r^2} = 1 - u = v$$

con lo cual

$$r = F^{-1}(v) = \sqrt{-2\log(1-v)}$$

donde u es el valor de una v.a. $U \rightsquigarrow \mathcal{U}(0,1)$. De esta manera se obtienen valores aleatorios para r y θ que generan dos v.a. Normales independientes a partir de la ecuación (4.2). Así pues, el algoritmo Polar o de Box-Muller estaría descrito de la forma

1. Genera un valor con una distribución *Uniforme* ($U_1 \rightsquigarrow \mathcal{U}(0,1)$).
2. Calcula $\theta = 2\pi u_1$
3. Genera un valor con una distribución *Uniforme* ($U_2 \rightsquigarrow \mathcal{U}(0,1)$).
4. Calcula $r = \sqrt{-2\log(u_2)}$.
5. Aplica las ecuaciones (4.2)

Es destacable que este proceso genera dos números en cada iteración. Sin embargo, tiene el inconveniente de utilizar las funciones $\sqrt{\cdot}, \log, \operatorname{sen}$ y $\cos$. Esto resulta poco eficiente desde un punto de vista computacional. Una alternativa es reducir las llamadas a estas funciones, evitando las trigonométricas, mediante un resultado básico de trigonometría. En efecto si

$$(v_1, v_2) \in (-1,1)\times(-1,1) \quad \text{con } 0 < v_1^2 + v_2^2 \leqslant 1 \Longrightarrow \begin{cases} \cos\theta = \dfrac{v_1}{\sqrt{v_1^2+v_2^2}} \\ \operatorname{sen}\theta = \dfrac{v_2}{\sqrt{v_1^2+v_2^2}} \end{cases}$$

Entonces el algoritmo de Box-Muller o Polar quedaría de la forma:

1. Genera dos valores v_1 y v_2 tales que $V_1 \rightsquigarrow \mathcal{U}(-1,1)$ y $V_2 \rightsquigarrow \mathcal{U}(-1,1)$, respectivamente.
2. Si $s = v_1^2 + v_2^2 \geqslant 1$, volver al paso anterior.
3. Como $S = V_1^2 + V_2^2 \rightsquigarrow \mathcal{U}(0,1)$, calcula $r = \sqrt{-2\log(s)}$.
4. Calcula

$$x = \sqrt{\frac{-2\log(s)}{s}}\, v_1$$

$$y = \sqrt{\frac{-2\log(s)}{s}}\, v_2$$

4.5. Tiempo de ejecución

Un procedimiento básico para contrastar la eficacia de un programa, consiste en medir su tiempo de ejecución. A menor tiempo, menor coste computacional lo cual mejora la operatividad del código.

Existen diferentes maneras de medir el tiempo de ejecución en un programa en **C**. Una muy simple es calcular la diferencia entre el instante anterior a la ejecución del programa y el instante posterior. Las variables que van almacenar los tiempos necesitan un tipo de declaración especial: **`clock_t`**, cuya definición está en la biblioteca **`time.h`**. Realmente es otra forma de identificar un tipo **`long integer`**. En estas variables se almacenan los enteros que devuelve la función **`clock()`**. Este valor entero, recoge el número de *ticks de reloj (clock tics)*. Generalmente los *ticks* suelen ser una medida arbitraria que puede depender del sistema operativo y del propio microprocesador[2]. En cualquier caso, es un valor que asocia **C** con un instante y es lo que se utiliza para medir el tiempo.

La forma de medir el tiempo de ejecución de un conjunto de sentencias de un programa, sería

Ejemplo de media de tiempo de ejecución

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

int main(void)
{
    ............
    time_t t_ini, t_final;
    double Tiempo_total;

    ......................

    t_ini = clock(); //Comienza a contar el reloj
           ..
           ..
           ..
       SENTENCIAS PARA  MEDIR EL TIEMPO DE EJECUCIÓN
           ..
           ..
```

[2]Realmente se trata de una medida poco normalizada. Mientras que ciertos autores la relacionan con los ciclos de reloj de un procesador, otros autores no, y la identifican con las órdenes del sistema operatico con el microprocesador.

```
    t_final = clock(); //Finaliza de contar el reloj

    Tiempo_total = (double)(t_final - t_ini) / CLOCKS_PER_SEC;

    ......................
    ......................

    return 0;
}
```

La variable `CLOCKS_PER_SEC` tiene almacenado el número de *ticks* que se realizan en un segundo. Por tanto, `Tiempo_total` es una variable real que almacena el número de segundos que se ha tardado en realizar la ejecución.

4.6. Problemas

1. En la sección 3.8 de Problemas (pág. 63) se ha definido el *Índice de Masa Corporal*. Se pide que hagas un programa que una vez calculado este coeficiente, tenga una salida que se ajuste al cuadro 4.1

Resultados del Índice de Masa Corporal	
Peso bajo	IMC $<$ 18.5
Peso normal	18.5 $\leqslant$ IMC $<$ 25
Sobrepeso	25 $\leqslant$ IMC $<$ 30
Obesidad	30 $\leqslant$ IMC

Cuadro 4.1 – *Interpretación del Índice de Masa Corporal*

utilizando **`if...else`** múltiple. Una posible ejecución podría ser

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 12345
**************************

Introduce tu peso en Kg. -> 70.2
Introduce tu altura en Mts. -> 1.73

Tu Índice de Masa Corporal es 23.46 y tu peso es ADECUADO.
```

2. Realiza un programa de forma que introducido un año, determine si es bisiesto o no. Para resolver este ejercicio se puede consultar la página *Leap Year* de Wikipedia en inglés (https://en.wikipedia.org/wiki/Leap_year) o bien en español, *año bisiesto* también de Wikipedia.

3. Realiza un programa que escriba en un fichero, una tabla con las potencias cuadradas y cúbicas de los n primeros números naturales, donde n debe ser seleccionado desde el teclado (ha de cumplirse que

$n \leqslant 100$). Este resultado debe escribirse en un fichero llamado `Datos.dat`. Una posible ejecución del programa podría ser

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 12345
*************************

Introduce el tamaño de la tabla -> -15
Has de introducir un número positivo menor o igual a 100
Introduce de nuevo, el tamaño de la tabla -> 5
```

y una posible solución sería

```
      N^1         N^2         N^3
       1           1           1
       2           4           8
       3           9          27
       4          16          64
       5          25         125
```

4. **Razón áurea.** Calcula el valor de la *fracción continua*

$$1+\cfrac{1}{1+\cfrac{1}{\ddots\cfrac{}{\ddots\cfrac{}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+1}}}}}}}}$$

La entrada del programa debe ser un valor entero que determine la cantidad de términos que deben calcularse.

Se trata de una sucesión que se aproxima al ***número áureo*** o ***razón áurea***: $(1+\sqrt{5})/2$. A continuación se pide que hagas un programa que dado un valor (por ejemplo: 10^{-6}), se determine el número de términos de la sucesión que se necesitan, para obtener una aproximación de la *razón áurea*, menor que el valor introducido. En la salida también debe darse esta aproximación.

5. Escribe un programa que determina todos los divisores de un numero natural y el resultado lo escriba en un fichero llamado `Datos.dat`. Una posible ejecución podría ser:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 12345
```

```
**************************

Introduce el número natural -> -72
El número debe ser positivo
Introduce de nuevo, el número natural -> 64
```

y la salida en el fichero `Datos.dat` debe ser

```
 Los divisores de 64 son:  1 2 4 8 16 32 64
```

6. Utilizando lo comentado en la sección 4.2.3 (pág. 72) con las funciones **`getc`** y **`putc`** escribe un programa, que lea un texto de un fichero de entrada (por ejemplo `Entrada.dat`) y escriba dicho texto, en un fichero de salida (por ejemplo `Salida.dat`). Se debe poder contar el número de caracteres del texto (incluidos los *blancos*). Por ejemplo, el fichero `Entrada.dat` podría contener el texto:

```
Esto es un texto para probar
que el programa funciona
```

7. Utilizando un bucle **`for`**, escribe un programa que calcule el factorial de un número natural n, introducido por el teclado. Ten en cuenta, que como consecuencia del ejercicio planteado en la sección 3.8 de Problemas, el valor del factorial para números mayores de 21 no es muy precisa[3], debido a un posible desbordamiento (*overflow*). Una posible ejecución del programa podría ser:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 12345
**************************

Introduce el número natural -> 59
Introduce un Número Natural menor o igual que 21!!!
Introduce de nuevo, el número natural -> 15
```

y el resultado

```
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 = 15 ! = 1307674368000
```

8. **Variaciones sin repetición**. Se llama *variaciones de m elementos tomados de n en n* ($0 \leqslant n \leqslant m$) al número

$$V_{m,n} = m \cdot (m-1) \cdots (m-n+1) = \frac{m!}{(m-n)!}$$

Haz un programa consistente en el cálculo de las variaciones, en el que los valores m y n, deben ser introducidos por teclado y pueden ser introducidos en cualquier orden.

9. **Variaciones con repetición**. Se llaman *variaciones con repetición de m elementos tomados de n en n* al número

$$VR_{m,n} = m^n$$

Haz un programa, empleando la instrucción **`for`**, pero sin utilizar la función **`pow`** (de la biblioteca de funciones matemáticas), que calcule las variaciones con repetición de m elementos tomados de n en n. Los números m y n serán introducidos por teclado y en cualquier orden.

[3] Condicionado por las especificaciones propias del compilador DEV-C++ utilizado.

10. **Combinaciones**. Se llaman *combinaciones de m elementos tomados de n en n* $(0 \leqslant n \leqslant m)$ al número

$$\binom{m}{n} = \frac{V_{m,n}}{n!} = \frac{m!}{(m-n)!\,n!}$$

Haz un programa que calcule el número combinatorio correctamente, dados dos números naturales m y n que entren por teclado y en cualquier orden[4].

11. Mediante la instrucción **`do-while`**, haz un programa que compruebe si un número introducido por teclado es un entero sin signo. Si lo es, que lo imprima. En caso contrario que vuelva a pedirlo por teclado.

12. En la sección 2.5 de Problemas (pág. 20) se planteó el *Algoritmo de Euclides*. Ahora se pide que programes este algoritmo con la instrucción **`do-while`**.

13. Escribe un programa que determine si un número entero positivo n es primo o no. Por lo general, basta con comprobar si todos los números menores o iguales que $\sqrt{n}$, dividen a n.

14. Utilizando una estructura **`switch`** escriba un programa que dados dos números enteros positivos, permita elegir al usuario, si obtener $V_{m,n}$, $VR_{m,n}$ o $\binom{m}{n}$. El programa terminará cuando se introduzca el carácter `@`.

15. Realiza un programa que imprima un cuadrado de asteriscos en un fichero llamado `Datos.dat`. El dato de entrada n, será el tamaño del lado. Por tanto el cuadrado contendrá $n \times n$ asteriscos. Un ejemplo de ejecución sería:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 12345
**************************

Introduce el tamaño del lado del cuadrado -> -25
Debe ser un entero positivo
Introduce, de nuevo, el tamaño del lado (entero positivo) -> 5
```

con una salida de la forma

```
*****
*****
*****
*****
*****
```

a) Con la misma idea que en el caso anterior, ahora que imprima, para un tamaño n, un triángulo de forma:

```
&
&&
&&&
&&&&
&&&&&
&&&&&&
```

[4] Este programa se puede simplificar definiendo el concepto de función que se verá más adelante

(en este caso $n = 6$)

b) Ahora un triángulo de la forma

```
$$$$$$
$$$$$
$$$$
$$$
$$
$
```

($n = 6$)

c) El siguiente con la forma

```
######
 #####
  ####
   ###
    ##
     #
```

($n = 6$)

d) Por último, una pirámide de la forma:

```
     @
    @@@
   @@@@@
  @@@@@@@
 @@@@@@@@@
@@@@@@@@@@@
```

($n = 6$)

16. Se llama *triángulo de Floyd* al triángulo rectángulo formado con números naturales. Para crear un *triángulo de Floyd*, se comienza con un 1 en la esquina superior izquierda, y se continúa escribiendo la secuencia de los números naturales de manera que cada línea contenga un número más que la anterior. Por ejemplo, el *triángulo de Floyd* para $n = 4$ sería:

1

2 3

4 5 6

7 8 9 10

Se pide que hagas un programa que dado un número n (tamaño del cateto), se imprima el *triángulo de Floyd*, en un fichero.

17. Realiza un programa que implemente el algoritmo planteado en (4.1) y obtenga 20 números aleatorios. La salida debe realizarse a un fichero llamado **`Salida.dat`** y se debe poder guardar los resultados obtenidos con ejecuciones anteriores. Prueba con los valores $a = 5, b = 3, m = 16$ y $n_0 = 7$. Repite la ejecución cambiando los valores de la semilla.

18. Realiza un programa que determine los n números enteros *pseudo-aleatorios* en el intervalo $[M, N]$ cuyos extremos deben introducirse por teclado y la salida debe escribirse en un fichero. Se debe introducir también el valor de la semilla y observar en el fichero las diferentes sucesiones obtenidas para diferentes semillas. Implementar la obtención de la semilla mediante la lectura del reloj del sistema.

19. Realiza un programa que determine los n números reales *pseudo-aleatorios* en el intervalo $[a, b]$ cuyos extremos deben introducirse por teclado y la salida debe escribirse en un fichero. Se debe introducir también el valor de la semilla y observar en el fichero las diferentes sucesiones obtenidas para diferentes semillas. Implementar la obtención de la semilla mediante la lectura del reloj del sistema.

20. Realiza un programa que simule el lanzamiento de una moneda $1.000.000$ de veces y obtener la frecuencia absoluta para cada una de las distintas caras.

21. Realiza un programa que simule el lanzamiento de un dado $6.000.000$ de veces y obtener la frecuencia absoluta para cada una de las distintas caras. ¿Cuál sería la probabilidad de cada una de las caras?

22. Utiliza uno de los dos algoritmos de Box-Muller planteados en teoría, para simular un conjunto de números con distribución Normal $N(\mu, \sigma)$ cuyos valores μ y σ deben introducirse, así como el número de datos que se deben generar.

23. Comprueba, tal y como dice en la teoría, que el segundo algoritmo de Box-Muller es más eficiente que el primero.

24. Anteriormente has realizado un programa que calculaba la tabla de los cuadrados y cubos de los n primeros números naturales. Amplía este programa para que también te pida la potencia. Por ejemplo, una posible ejecución podría ser:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 12345
**************************

Introduce el número de potencia máxima -> 5

Introduce la longitud de la tabla -> 10
```

y la salida

```
      N^1       N^2       N^3       N^4       N^5
        1         1         1         1         1
        2         4         8        16        32
        3         9        27        81       243
        4        16        64       256      1024
        5        25       125       625      3125
        6        36       216      1296      7776
        7        49       343      2401     16807
        8        64       512      4096     32768
        9        81       729      6561     59049
       10       100      1000     10000    100000
```

(Propuesto en examen: julio – 2015)

25. Se pide que a partir de un intervalo dado se generen una cierta cantidad de números aleatorios ***enteros***: $x_1, \ldots, x_n$ y con ellos calcules la *media* y la *cuasi-desviación típica muestral*, esto es:

$$\overline{x} = \frac{1}{n}\sum_{i=1}^{n} x_i \qquad \text{y} \qquad s = \sqrt{s^2}$$

donde

$$s^2 = \frac{1}{n-1}\left(\sum_{i=1}^{n} x_i^2 - n\overline{x}^2\right)$$

El programa debe permitir al usuario seleccionar:

a) la cantidad de números *pseudo-aleatorios* que deben generarse

b) los extremos del intervalo (que deben ser valores ***enteros***), en cualquier orden.

c) semilla (un valor entero), para generar diversas muestras de números *pseudo–aleatorios*.

Una ejecución típica del programa con salida en pantalla sería:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 1111111
**************************

Introduce la cantidad de números a generar ( > 0) -> -7
Introduce la cantidad de números a generar ( > 0) -> 100

Introduce los extremos del intervalo (enteros), en cualquier orden -> 10 1


Introduce la semilla -> -3

******** Salida de resultados ********

El intervalo es [1, 10]
La semilla es -3
El número de datos es 100:

La media es 5.690000000
La CuasiDesviación Típica muestral es 2.751289872
```

(Propuesto en examen: curso 15/16)

26. Dado un número ***real*** x definido de la forma

$$x = 1 + \left(\frac{1}{2}\right)^n \qquad \text{con} \quad n \in \mathbb{N}$$

se pide que realices un programa que determine el mayor número natural n_0 tal que se cumpla:

$$x = 1 + \left(\frac{1}{2}\right)^n > 1$$

El programa debe permitir al usuario seleccionar si el cálculo se realiza en *simple precisión*, tecleando la letra **S** o en *doble precisión*, tecleando la letra **D**.

Una ejecución típica del programa con salida en pantalla sería:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 1111111
**************************

Introduce S (simple precisión) o D (doble precisión) -> P
Introduce S (simple precisión) o D (doble precisión) -> S

El valor de n0 es: 95
El valor de 1/2 elevado a n0 es: 0.0023430127
```

(Propuesto en examen: curso 15/16)

CAPÍTULO 5

Datos estructurados

5.1. Introducción

Dentro del conjunto de de los ***datos estructurados*** que se pueden definir en **C**, los más sencillos son los *vectores y matrices*. Desde el punto de vista informático, un ***vector*** es un conjunto de datos del mismo *tipo* y bajo un mismo nombre, cuyos elementos pueden localizarse independientemente por medio de **un** *índice*. El concepto de ***matriz*** es análogo al de *vector*, con la única diferencia que los datos se localizan mediante **dos** *índices*.

En general, en informática, para referirse a este tipo de estructuras de datos, e incluso a aquellas para los que es necesario la utilización de más de dos *índices*, se utiliza la palabra *array*. Así, los *arrays de un índice* corresponde a la idea de *vectores*, mientras que los *arrays de dos índices* correspondería a las matrices. Los *arrays de más de dos índices*, tienen una difícil traducción en castellano[1]. En este documento se utilizará únicamente la palabra *array*, cuando sea necesario la utilización de más de dos índices. Los *vectores, matrices y arrays*, constituyen unos *datos estructurados* llamados ***homogéneos***, puesto que todos sus datos son del mismo *tipo*.

En **C**, se pueden definir unos datos estructurados más complejos en los que es posible mezclar diferentes *tipos de datos*. Se conocen como *estructuras* y por tanto, son unos *datos estructurados* ***heterogéneos***. Con esta modalidad, el programador puede definir su propio *tipo* de variable, proporcionando una gran flexibilidad.

En este capítulo, se estudiará de qué manera el **lenguaje C** representa y opera con estas estructuras de datos.

5.2. Vectores

En matemáticas, los vectores en espacios de dimensión finita se pueden asociar a *n-uplas* de números reales. Esto es,

$$\boldsymbol{v} = [a_1, a_2, \ldots, a_n] \qquad \text{con} \quad a_i \in \mathbb{R}$$

Como ya se ha dicho, en informática y por tanto, en **C** la idea de vector es similar, pero algo más general; sus elementos, por ejemplo, no tienen por qué ser números, pueden ser caracteres, y dar lugar a una *cadena de caracteres*.

[1] Desde un punto de vista matemático, si los datos son números reales, esta disposición de datos se pueden asociar al concepto de *tensor*, expresado en una base de dimensión finita.

Los *vectores* en **C** tienen un tratamiento similar a las variables: se declaran, se inicializan, se modifican, se operan, *etc*. La manera de declarar un vector es

```
tipo nombre[número_de_elementos];
```

Este tipo de declaración tiene varias consecuencias:

1. El **`número_de_elementos`** (o *dimensión*) puede establecerse mediante una expresión constante o variable entera[2].
2. Cada componente del vector es a su vez una variable del *mismo tipo* que el utilizado en la declaración.
3. El tamaño de memoria que se reserva para las variables vectoriales, se calcula como producto del tamaño del *tipo* (**`int`** (4 *bytes*), **`float`** (4 *bytes*), **`double`** (8 *bytes*), **`char`** (1 *byte*), ...) por la dimensión (número de elementos).
4. En **C** los elementos se numeran desde el 0 hasta **`número_de_elementos`**-1.

Por ejemplo:

```
1  const int Dim = 30;
2
3  int vec_ent[Dim];
4  double vec_real[15];
5  char frase[25];
```

En la LÍNEA 3, se reserva un espacio de 30 variables de *tipo* **`int`**, asociadas al mismo nombre de `vec_ent`. Esto quiere decir que en memoria se reservan $30 \times 4 = 120$ *bytes* consecutivos, para estas 30 variables. La dimensión de esta variable vectorial, se establece a partir de la *constante* `Dim`. La numeración de las variables empieza en el 0 y termina en el 29. Cada variable se representa por el nombre y entre corchetes, el valor del índice. Esto es, se tendría las siguientes variables enteras

```
vec_ent[0], vec_ent[1], ..., vec_ent[29]
```

De manera análoga, en la LÍNEA 4 se reserva un espacio para 15 variables reales **`double`**, con el nombre de `vec_real` y en la última línea se define un vector de nombre `frase` y de tamaño 25 caracteres (o *bytes*: cada carácter se almacena en un *byte*)).

Para acceder a un elemento del vector, basta con incluir en una expresión su nombre seguido del *índice* entre corchetes. En **C**, no se puede operar con todo un vector o matriz como única entidad (otros lenguajes como MATLAB, sí lo permiten), se debe tratar cada una de sus componentes de manera independiente. Esto implica la utilización de bucles, generalmente **`for`** o **`while`**, para que todos las componentes se vean afectadas de la misma forma.

En **C** las componentes de los vectores, se utilizan como cualquier otra variable (de hecho, son variables). Son perfectamente factibles las siguientes expresiones:

```
vec_real[7] = 12.56;
vec_real[0] = vec_real[26] * vec_real[18];
vec_real[24] = vec_real[13]/vec_real[7] + 23.8 * vec_real[21];
vec_real[20] = vec_real[20] + 1;
```

[2]Los compiladores más actuales de **C** ofrecen la posibilidad de dimensionar un vector mediante una variable (llamado *dimensionamiento dinámico*, debido a que el dimensionamiento se realiza durante la ejecución del programa). El compilador DEV-C++, que se utiliza en este curso, sí permite esta posibilidad (véase sección 5.7, pág. 125).

La *inicialización* de los vectores puede hacerse de diferentes maneras

```
int enteros[7] = {21, 34, -23, 10, 0, -7, 40};
double reales[] = {1.4, -6.0945, 123.56};
float reales_cortos[10] = {1.23, 34.87, -23.1};
int Vtr[10] = {[3] = 1, [1] = -1, [7] = 52};
```

En la primera línea se inicializa cada elemento del vector `enteros` por los valores considerados, de esta forma se tiene:

`enteros[0] = 21; enteros[1] = 34; ··· ; enteros[6] = 40`

En la siguiente, mediante esta inicialización la dimensión de `reales[3]`, queda implícita. Con la inicialización que viene a continuación, **C** considera que el resto de los componentes del vector `reales_cortos`, que no se inicializan explícitamente, son 0. Así pues, para hacer 0 todas las componentes del vector `reales_cortos`, hubiese bastado con la declaración:

```
float reales_cortos[10] = {0.};
```

En la última línea, se muestra cómo se pueden inicializar sólo algunas de las componentes del vector.

Un programa básico sobre la inicialización y operaciones con un vector puede verse en el código 5.1

Código 5.1 – *Ejemplo de inicialización y operación*

```
#include <stdio.h>
/* Ejemplo de programa con vectores */
int main(void)
{
    int Vtr[6] = {-5, -4,-2, [5]=2};
    unsigned int cont;


    FILE *fi;

    fi = fopen("Datos.dat", "w");

    fprintf(fi, "El vector antes de operar:\n");
    for(cont = 0; cont < 6; cont++){
        fprintf(fi, "Vtr[%u] = %i; ", cont, Vtr[cont]);
    }
    fprintf(fi, "\n\n");
    fprintf(fi, "El vector después de operar:\n");
    for(cont = 0; cont < 6; cont++){
        Vtr[cont] = 2 * cont;
        fprintf(fi, "Vtr[%u] = %i; ", cont, Vtr[cont]);
    }
    fclose(fi);

    return 0;
}
```

la salida en el fichero `Datos.dat` es

```
El vector antes de operar:
```

```
Vtr[0] = -5; Vtr[1] = -4; Vtr[2] = -2; Vtr[3] = 0; Vtr[4] = 0; Vtr[5] = 2;

El vector después de operar:
Vtr[0] = 0; Vtr[1] = 2; Vtr[2] = 4; Vtr[3] = 6; Vtr[4] = 8; Vtr[5] = 10;
```

Hay que insistir que las componentes de un vector están constituidas por variables de un *mismo tipo*. Por tanto, todo lo visto hasta ahora, referente a variables es de aplicación a cada una de las componentes de un vector. De esta forma, también es posible introducir los valores de las componentes de un vector a través del teclado, mediante la función **scanf**, tal y como se ilustra en el código 5.2. En este caso se introducen los valores de las componentes y el resultado se imprime en el ficheros `Datos.dat`.

Código 5.2 – *Ejemplo de introducción de datos*

```c
#include <stdio.h>

int main(void)
{
    const int LIM = 5;
    int Vec[LIM];
    unsigned int i;

    FILE *fi;

    fi = fopen("Datos.dat", "w");

    // ******* Lectura de datos del vector  *******
    printf("\n\n Introduce las componentes del vector: \n\n");
    for(i = 0; i < LIM; i++){
        printf("Vec[%u] = ", i+1);
        scanf("%i", &Vec[i]);
        while(getchar() != '\n');
        printf("\n");
    }
    fprintf(fi, "El vector introducido es: \n\n");
    for(i=0; i < LIM; i++){
        fprintf(fi, " Vect[%u] = %i; ", i+1, Vec[i]);
    }
    fclose(fi);

    return 0;
}
```

La ejecución del programa sería (se introducen por el teclado los números: $23, -4, 0, -123, 42$)

```
  Introduce las componentes del vector:

Vect[1] = 23
Vect[2] = -4
Vect[3] = 0
Vect[4] = -123
Vect[5] = 42
```

y la salida en el fichero `Datos.dat` es

```
El vector introducido es:
```

```
Vect[1] = 23;  Vect[2] = -4;  Vect[3] = 0;  Vect[4] = -123;  Vect[5] = 42;
```

No es frecuente introducir las componentes de un vector a través del teclado, sobre todo si el número de componentes es elevado. Lo más habitual es leer los valores de un fichero, en el que previamente se han introducido por la acción de cierto proceso anterior. Mediante la función **fscanf**, se pueden leer valores de variables en ficheros. Su estructura y funcionamiento es similar a la función **fprintf**, que ya ha sido utilizada en programas anteriores. La diferencia fundamental con la función **scanf** es que hay que añadir una nueva componente a su argumento, que corresponderá al identificador del fichero, como ocurre con **fprintf**.

En el código 5.3, se plantea un ejemplo sencillo en el que se leen los valores de las componentes de un vector. Posteriormente se almacenan en un fichero llamado `Fichero.dat`. Por último, se leen de este fichero los datos introducidos y se muestran en la pantalla.

Código 5.3 – *Ejemplo de lectura de datos desde un fichero*

```
#include <stdio.h>

int main(void)
{
    const int LIM = 5;
    float Vec[LIM], Vec2[LIM];
    unsigned int i;

    FILE *fi;

    fi = fopen("Fichero.dat", "w");


    // ******* Lectura de datos del vector  *******
    printf("\n\n Introduce las componentes del vector: \n\n");
    for(i = 0; i < LIM; i++){
        printf("Vec[%u] = ", i+1);
        scanf("%f", &Vec[i]);
        while(getchar() != '\n');
        printf("\n");
    }
    for(i=0; i < LIM; i++){
        fprintf(fi, "%8.4f", Vec[i]);
        fprintf(fi, "\n");
    }
    fclose(fi);
    fi = fopen("Fichero.dat", "r");
    for(i=0; i < LIM; i++){
        fscanf(fi, "%f", &Vec2[i]);
    }
    printf("El vector introducido es:\n");
    for(i=0; i < LIM; i++){
        printf(" Vect2[%u] = %.4f; \n", i+1, Vec2[i]);
    }
    fclose(fi);

    return 0;
}
```

La lectura de los datos desde el fichero `Fichero.dat` se realiza, mediante un bucle **`for`** en la LÍNEAS 28-30. Es importante señalar que primeramente se ha *abierto* (o *creado*) el fichero en modo *escritura*, mediante el *modo* `w`, como se observa en la LÍNEA 11 (véase sección 3.6, Ficheros pág. 59). Pero para leer los datos almacenados, se ha debido *cerrar* (LÍNEA 26) y volver *abrir* previamente, en el *modo* `r` (LÍNEA 27). Esto se hace, para que el *cabezal* del fichero se vuelva a posicionar al principio de fichero y pueda leer los datos desde el comienzo del fichero. Además se debe *abrir* con la opción `r`, porque con `w` hubiese borrado el contenido, previamente almacenado.

5.2.1. Sucesión de Fibonacci

De la misma manera que pueden realizarse operaciones con las componentes de un vector, también puede operarse con los índices de las componentes. En este sentido, un ejemplo sencillo e interesante consiste en escribir una tabla con los elementos de la *sucesión de Fibonacci*. Cada término de la *sucesión de Fibonacci* se forma a partir de la suma de los dos anteriores. Los dos primeros términos son $0, 1$, el tercero 1, el cuarto 2, el quinto 3, el sexto 5 y así sucesivamente:

$$0, \quad 1, \quad 1, \quad 2, \quad 3, \quad 5, \quad 8, \quad 13, \quad 21, \quad 34, \quad \ldots$$

Un ejemplo de código que genera los términos de la *sucesión de Fibonacci* se presenta en el programa 5.4. En este código se pide un número N mayor que 2 pero menor que 30 y genera los N primeros números de la sucesión.

Código 5.4 – *Sucesión de Fibonacci*

```
1  #include <stdio.h>
2
3  /* Construye una tabla con los números de Fibonacci */
4
5  int main(void)
6  {
7
8
9      int N;
10     unsigned long long TablaFibo[30] = {0ULL ,1ULL};
11     unsigned int ind;
12
13     FILE *fi;
14
15     fi = fopen("Datos.dat", "w");
16
17     printf("\n\n **** Este programa determina la tabla de los número de
           Fibonacci ****\n\n");
18     printf("Introduce un entero 2 < N < 30 -> ");
19     while(scanf("%i", &N) != 1 || N < 3 || N > 29){
20         while(getchar() != '\n');  /* limpia el flujo de entrada de datos*/
21             /*Se introduce de nuevo el numero */
22         printf("\n\nHas de introducir entero positivo 2 < n < 30 -> ");
23     }
24     while(getchar() != '\n');
25
26    // Construye los términos de la sucesión
27     for(ind = 2; ind < N; ind++){
28         TablaFibo[ind] = TablaFibo[ind-2] + TablaFibo[ind-1];
```

```
29      }
30
31      /* Imprime la tabla  */
32      fprintf(fi, "Los %i primeros números de Fibonacci son: \n\n", N);
33      for (ind = 0; ind < N; ind++){
34          fprintf(fi, "%llu ; ", TablaFibo[ind]);
35      }
36
37      fclose(fi);
38
39      return 0;
40  }
```

En la LÍNEA 28 se aprecia cómo para obtener el siguiente término de la sucesión a partir de los dos anteriores, únicamente hay que operar con los índices de las componentes del vector `TablaFibo`.

5.2.2. Mínimo de las componentes de un vector

A veces, resulta necesario calcular el valor máximo y mínimo de las componentes de un vector, así como su posición. En el ejemplo 5.5, se ha realizado un programa que determina el valor mínimo de un vector, junto con la posición que ocupa

Código 5.5 – *Valor mínimo de un vector*

```
1   #include <stdio.h>
2   /* Algoritmo para localizar la componente de valor mínimo */
3   #define DIM 5
4
5   int main(void)
6   {
7       int i;
8       double Vec[DIM] = {2.4, 4.5, -23.5, 2.56, 7.95};
9       double Min;
10      unsigned int Pos;
11
12      FILE *fi;
13      fi = fopen("Datos.dat", "w");
14
15      Min = Vec[0];
16      Pos = 0;
17      for (i = 1; i < DIM; i++){
18          if (Vec[i] < Min){
19              Min = Vec[i];
20              Pos = i;
21          }
22      }
23      fprintf(fi, "El valor mínimo del vector: \n");
24      for(i = 0; i < DIM; i++){
25          fprintf(fi, "%.4lf ", Vec[i]);
26      }
27      fprintf(fi,"\n");
28      fprintf(fi,"es %.4lf y ocupa la posición %u \n", Min, Pos+1);
29
30      fclose(fi);
```

```
31
32      return 0;
33  }
```

La salida en el fichero `Datos.dat` es

```
El valor mínimo del vector:
2.4000 4.5000 -23.5000 2.5600 7.9500
es -23.5000 y ocupa la posición 3
```

5.2.3. Algoritmo de Horner

Dada una función polinómica:

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 = \sum_{i=0}^{n} a_i x^i$$

se puede operar computacionalmente con ella, guardando sus coeficientes a_i en las componentes de un vector de tamaño $n+1$. Una operación bastante habitual es evaluar dicha función polinómica en un punto dado x_0, esto es

$$p(x_0) = a_n x_0^n + a_{n-1}x_0^{n-1} + \cdots + a_1 x_0 + a_0 \tag{5.1}$$

Su expresión en **C** sería:

```
1       punto = x0;
2       for (valor = a[0], i = 1; i <= n; i++){
3           valor += a[i]*pow(punto, i);
4       }
```

donde **pow** es la función *potencia* de la *biblioteca* **math.h**. Sin embargo, resulta más eficiente desde el punto de vista del número de operaciones, proceder de la siguiente manera:

$$p(x_0) = a_0 + x_0(\, a_1 + x_0(\, a_2 + \cdots + x_0(a_{n-1} + a_n x_0)\cdots)),$$

Se trata del *Algoritmo de Horner* y cuya representación en **C** sería:

```
1       punto = x0;
2       for (valor = a[n], i = n-1; 0 <= i; i--){
3           valor *=  punto;
4           valor +=  a[i];
5       }
```

En este caso se realizan n adiciones y n productos, mientras que en el procedimiento clásico (5.1) se realizan n sumas y $2n-1$ productos, es decir $n-1$ multiplicaciones de más. Se trata de un procedimiento similar a la *Regla de Ruffini* que es utilizada en cursos básicos, para evaluar un polinomio en un punto dado.

En el código 5.6 se muestra un ejemplo en el que se utiliza el *algoritmo de Horner*, para evaluar el polinomio

$$p(x) = 2.4 + 4.5x - 23.5x^2 + 2.56x^3 + 7.95x^4$$

en el punto $x_0 = -1.6$.

Código 5.6 – *Algoritmo de Horner*

```
1  #include <stdio.h>
2  #include <math.h>
3  /* Algoritmo de Horner */
4  #define DIM 5
5
6
7  int main(void)
8  {
9      int i;
10     double pol[DIM] = {2.4, 4.5, -23.5, 2.56, 7.95};
11     double x0 = -1.6;
12     double valor, valor2;
13
14     FILE *fi;
15     fi = fopen("Datos.dat", "w");
16
17     valor = pol[DIM-1]; //pol[DIM-1] = a_{n}
18     for (i = DIM - 2; 0 <= i; i--){
19         valor *=  x0;
20         valor += pol[i];
21     }
22
23     fprintf(fi, " Valor con Horner: %.9f\n", valor);
24
25     for (valor2 = pol[0], i = 1; i <= DIM-1; i++){
26         valor2 +=  pol[i]*pow(x0, i);
27     }
28
29     fprintf(fi, " Valor procedimiento habitual es: %.9f\n", valor2);
30
31     fclose(fi);
32
33     return 0;
34 }
```

En este programa los coeficientes del polinomio se introducen a través de un vector de *tipo* `double`, llamado **`pol`** (LÍNEA 10) y que se inicializa con los valores deseados. El *algoritmo de Horner* codifica en las LÍNEAS 17-21. El método tradicional de evaluación, se presenta en las LÍNEAS 25-27. El resultado de la ejecución del programa 5.6 se almacena el fichero `Datos.dat` y contiene, para los datos de entrada (LÍNEAS 10-11):

```
Valor con Horner: -23.344640000
Valor procedimiento habitual es: -23.344640000
```

5.3. Cadenas de caracteres

Una ***cadena de caracteres*** es un *vector* de tipo **`char`**. Esto es, en cada posición del vector almacena un carácter. Sin embargo, tiene algunas particularidades frente a los vectores numéricos, que merecen la pena destacar. Una de ellas, tal vez la más importante, es que automáticamente añade al final del texto, el carácter '`\0`', para separar la parte del vector que contiene texto, de la que está vacía. Así pues, si se declara

```
char Nombre[20] = "Francisco José";
char OtroNombre[] = "María José";
```

la cadena `Nombre` contendría

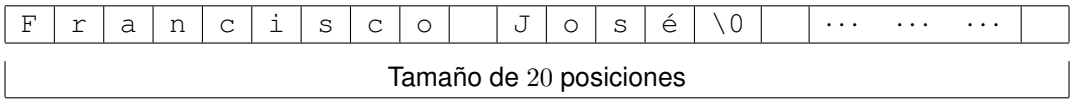


Figura 5.1 – *Esquema de almacenamiento de la cadena* `Nombre`

mientras que la cadena `OtroNombre` tendría almacenado:

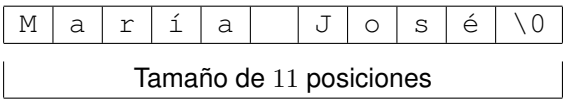


Figura 5.2 – *Esquema de almacenamiento de la cadena* `OtroNombre`

Las *cadenas de caracteres* tienen su propio especificador de formato, que es `s`. Con lo cual si se quiere escribir una *cadena*, se pondría:

```
printf("El nombre elegido es -> %s\n", Nombre);
```

Si se quiere introducir una *cadena de caracteres* se emplearía la función **`scanf`**, como muestra el ejemplo

```
printf("Introduce el nombre elegido -> ");
scanf("%s", Nombre);
```

aunque existe una pequeña diferencia con los casos ya vistos, en capítulos anteriores. Ahora ya no es necesario la utilización del *operador dirección*: `&`, porque el nombre de la *cadena*, constituye un *puntero*[3]. No obstante, a la hora de introducir una *cadena de caracteres* por medio del teclado, es más adecuado utilizar la función **`gets`**. La diferencia es que **`scanf`** se para en el momento de localizar un blanco y **`gets`** se detiene cuando encuentra un *retorno de carro*. Su utilización es muy simple, como se observa en el siguiente ejemplo:

```
printf("Introduce el nombre elegido -> ");
gets(OtroNombre);
printf("El nombre introducido es  %s\n", OtroNombre);
```

Para una descripción detallada, puede verse en el libro de Rodríguez–Losada *et al* [18].

5.3.1. Funciones específicas

El **lenguaje C** ofrece una serie de funciones que facilitan el trabajo con las *cadenas de caracteres*. Estas funciones se localizan en la *biblioteca* **`<string.h>`**. En esta sección, únicamente se tratarán aquellas más habituales (para un estudio más detallado puede consultarse el libro de Rodríguez–Losada *et al* [18]).

Función `strlen`

Es una función que devuelve la longitud de una cadena de caracteres. Su sintaxis es **`strlen(cadena)`**.

[3] Realmente un *puntero a constante*, que es el nombre de la *cadena*. El concepto de *puntero* se estudiará con detalle más adelante.

Código 5.7 – ***strlen()***

```
#include <stdio.h>
#include <string.h>
/* Ejemplo de programa con cadenas de caracteres */

int main(void)
{
      char fraseguay[] = "Me encantan las clases de C";

      FILE *fi;
      fi = fopen("Datos.dat", "w");

      fprintf(fi, "La frase es: / %s /\n", fraseguay);
      fprintf(fi, " y está formada por %d caracteres", strlen(fraseguay));


      fclose(fi);

      return 0;
}
```

La salida resultante sería:

```
La frase es: / Me encantan las clases de C /
 y está formada por 27 caracteres
```

Función `strcat`

Su expresión es **`strcat(cadena1, cadena2)`** y permite concatenar dos cadenas de caracteres, de la siguiente forma: toma dos *cadenas de caracteres* como argumentos y ***añade*** una copia de la segunda al ***final*** de la primera y hace que esta versión combinada sea la nueva *cadena de caracteres*. Por tanto, es importante asegurarse que en la primera *cadena*, hay suficiente espacio para almacenar, posteriormente la segunda.

Código 5.8 – ***strcat()***

```
#include <stdio.h>
#include <string.h>
/* Ejemplo de programa con cadenas de caracteres */

int main(void)
{
      char frase_a[50] = "Me gusta mucho el C. ";
      char frase_b[] = "Me puede poner más ejercicios?";

      FILE *fi;
      fi = fopen("Datos.dat", "w");

      fprintf(fi, "La frase es: ' %s' \n", strcat(frase_a, frase_b));

      fclose(fi);

      return 0;
}
```

La salida que se obtendría sería:

```
La frase es: 'Me gusta mucho el C. Me puede poner más ejercicios?'
```

Función `strcmp`

Compara alfabéticamente dos *cadenas de caracteres*: **`strcmp(cadena1, cadena2)`**. Únicamente devuelve tres valores: $-1, 0, 1$.

- **`0`**: Ambas *cadenas* son iguales.
- **`1`**: De izda. a decha. hay algún carácter en la **`cadena1`** que es ***posterior*** a la **`cadena2`**. Por tanto, **no** están ordenadas alfabéticamente.
- **`-1`**: De izda. a decha. hay algún carácter en la **`cadena1`** que es ***anterior*** a la **`cadena2`**. Por tanto, están ordenadas alfabéticamente.

Veamos el siguiente ejemplo:

Código 5.9 – ***`strcmp()`***

```c
#include <stdio.h>
#include <string.h>
/* Ejemplo de programa con cadenas de caracteres */

int main(void)
{
      FILE *fi;
      fi = fopen("Datos.dat", "w");


      fprintf(fi, "A-A: %d\n", strcmp("A", "A"));
      fprintf(fi, "A-B: %d\n", strcmp("A","B"));
      fprintf(fi, "B-A: %d\n", strcmp("B","A"));
      fprintf(fi, "C-A: %d\n", strcmp("C","A"));
      fprintf(fi, "Sánchez-Sanchez: %d\n", strcmp("Sánchez", "Sanchez"));
      fprintf(fi, "Spabcy-Spabzl: %d\n", strcmp("Spabcy", "Spabzl"));

      fclose(fi);

      return 0;
}
```

La salida sería:

```
A-A: 0
A-B: -1
B-A: 1
C-A: 1
Sánchez-Sanchez: 1
Spabcy-Spabzl: -1
```

Realmente calcula la diferencia de los códigos ascii de cada uno de los caracteres que constituyen las respectivas *cadenas*. Si el resultado es positivo la salida es 1 y las *cadenas* **no** están ordenadas alfabéticamente, si es -1 las cadenas están ordenadas alfabéticamente y si es 0 quiere decir que todos los códigos ASCII son iguales y por tanto, las *cadenas* son idénticas.

Función `strcpy`

Su expresión es **`strcpy(cadena1, cadena2)`**: copia en la primera *cadena de caracteres*, el contenido de la segunda. De manera análoga a lo que ocurría con la función **`strcat`**, el tamaño de la *cadena* a la que apunta: **`cadena1`**, deberá tener un tamaño suficiente para guardar la **`cadena2`**. Veamos un ejemplo de esta función, utilizándola para ordenar una serie de *cadenas de caracteres*.

Código 5.10 – ***strcpy()***

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

/* Ejemplo de programa con cadenas de caracteres */

#define FILA 5
#define COLUM 10
int main(void)
{
      int i,j;

      char apellidos[FILA][COLUM] ={"Sánchez", "López", "Abad", "Nadal", "
         Rodrigo"};
      char aux[COLUM];

      FILE *fi;
      fi = fopen("Datos.dat", "w");

      fprintf(fi, "La lista original es:\n");
      for (i = 0; i < FILA; i++)
         fprintf(fi, "%s\n", apellidos[i]);

      /* Se ordena alfabéticamente */

      for(i = 0; i < FILA-1; i++)
          for(j = i + 1; j < FILA; j++) {
                if ( strcmp(apellidos[i], apellidos[j]) > 0 ){
                     strcpy(aux, apellidos[i]);
                     strcpy(apellidos[i], apellidos[j]);
                     strcpy(apellidos[j], aux);
                     }
          }
      fprintf(fi, "\n\n");
      fprintf(fi, "La lista ordenada es:\n");
      for (i = 0; i< FILA; i++)
         fprintf(fi, "%s\n", apellidos[i]);

      fclose(fi);

```

```
39        return 0;
40  }
```

El resultado de la ejecución sería:

```
La lista original es:
Sánchez
López
Abad
Nadal
Rodrigo


La lista ordenada es:
Abad
López
Nadal
Rodrigo
Sánchez
```

5.3.2. Introducción del nombre de un fichero

Hay veces que resulta útil introducir el nombre del fichero o ficheros desde el teclado, para realizar las operaciones de E/S (ENTRADA/SALIDA). A partir del concepto de *cadenas de caracteres* que se acaba de introducir, resulta sencillo. La idea es que el usuario introduzca desde el teclado el nombre del fichero que desea *abrir*, ya sea para escribir datos o leerlos. El código 5.11 presenta un programa que *abre* un fichero en *modo lectura* (*read mode*) (véase sección 3.6, pág. 59). Si no existe un fichero con el nombre tecleado, el programa repite la ejecución, hasta que se introduce correctamente.

Código 5.11 – *Introducción de un nombre de fichero*

```
1  #include <stdio.h>
2  //Introduce el nombre de un fichero
3
4  int main(void)
5  {
6      const int LIM = 100;
7      char nom_fic[LIM];
8
9      FILE *fi;
10
11     do{
12         printf("\n\n Introduce el nombre del fichero -> ");
13         gets(nom_fic);
14         printf("\n\n El nombre introducido es: %s\n\n", nom_fic);
15         if ( (fi = fopen(nom_fic, "r")) == NULL ){
16             printf("*** ¡¡¡ OJO ERROR EN LA APERTURA DEL FICHERO !!!! ***\n\n");
17         }
18     } while (fi ==  NULL);
19     fclose(fi);
20
21     return 0;
22 }
```

5.4. Matrices

Las *matrices* tienen unas características análogas a las ya vistas para los *vectores*. Sin embargo, ahora hay que tener en cuenta que son estructuras con dos *índices*. El aspecto general de la declaración de una matriz, toma la forma:

```
tipo nombre[número de filas][número de columnas];
```

Como ocurre con los vectores, tanto las *filas* como las *columnas*, se numeran también a partir de 0. La forma de acceder a los elementos de la matriz es utilizando su nombre, seguido de las expresiones enteras correspondientes a los *índices*, entre corchetes. Esto es, `a[i][j]` indica el elemento situado en la *fila* i y *columna* j.

Es importante destacar que en **C**, las matrices se almacenan por *filas*. Esto quiere decir, que los elementos de una fila ocupan posiciones de memoria contiguas[4]. Por tanto, en **C**, suele ser mucho más eficiente variar primeramente, el índice de la columna y a continuación el de la fila, si quiere desplazarse por las componentes de una matriz. Realmente **C**, considera las matrices, como *vectores* cuyas componentes son a su vez *vectores*.

Las componentes de las matrices, al igual que sucedía con los vectores, pueden ser números o caracteres. Un ejemplo de este último caso, puede apreciarse en el código 5.10 (pág. 111) en la variable `apellidos`.

La inicialización de matrices sería análoga a la de los *vectores*. Se pueden inicializar todas sus componentes o alguna de ellas, poniendo a 0, el resto. Un ejemplo muy simple se muestra en el código 5.12, en el que se inicializa algunas de las componentes de una matriz y después se imprime en un fichero.

Código 5.12 – *Inicialización de matrices*

```
#include <stdio.h>

/* Inicialización de matrices */

int main(void)
{
    int Mat1[3][3] = {[0][0] = -5, [1][2] = -6};
    int Mat2[2][3] = {{1, 4}, {-7, 2}};
    unsigned int f, c;

    FILE *fi;
    fi = fopen("Datos.dat", "w");

    fprintf(fi, "Coeficientes de Mat1 \n\n");
    for (f = 0; f <  3; f++){
        for (c = 0; c < 3; c++){
            fprintf(fi, "%4i", Mat1[f][c]);
        }
        fprintf(fi, "\n");
    }
    fprintf(fi,"\n");
    fprintf(fi, "Coeficientes de Mat2\n\n");
    for (f = 0; f <  2; f++){
        for (c = 0; c < 3; c++){
            fprintf(fi, "%4i", Mat2[f][c]);
        }
```

[4] No es siempre así en todos los lenguajes. Por ejemplo en FORTRAN y MATLAB las matrices se almacenan por *columnas*.

```
27            fprintf(fi,"\n");
28        }
29        fclose(fi);
30
31        return 0;
32    }
```

La salida que se obtiene en fichero `Datos.dat` es

```
Coeficientes de Mat1

  -5   0   0
   0   0  -6
   0   0   0

Coeficientes de Mat2

   1   4   0
  -7   2   0
```

5.4.1. Suma de matrices

Como ya se comentó, el **lenguaje C** no es capaz de operar con los vectores o matrices como una *unidad*, hay que hacerlo compomente a componente. Uno de los casos más sencillos es la suma de matrices. Sean dos matrices $A \in \mathcal{M}_{m\times n}$ y $B \in \mathcal{M}_{m\times n}$. Esto es

- $A = [a_{i,j}]$ con $i = 1, \ldots, m$ y $j = 1, \ldots, n$
- $B = [b_{i,j}]$ con $i = 1, \ldots, m$ y $j = 1, \ldots, n$

La suma de estas dos matrices es una nueva matriz $C \in \mathcal{M}_{m\times n}$ definida de la forma

$$c_{i,j} = a_{i,j} + b_{i,j} \qquad \text{con } \; i = 1, \ldots, m; \; j = 1, \ldots, n \tag{5.2}$$

en **C** su codificación sería directa:

```
      for(i = 0; i < Num_Filas; i++){
          for( j = 0; j < Num_Columnas; j++){
             C[i][j] = A[i][j] + B[i][j];
          }
      }
```

Como se observa, se hace variar para cada fila, todas sus columnas.

5.4.2. Producto de matrices

Otro ejemplo interesante es el producto de matrices. No es tan directo como el caso de la *suma de matrices*, pero también sencillo. Sean dos matrices $A \in \mathcal{M}_{n\times p}$ y $B \in \mathcal{M}_{p\times m}$. Esto es

- $A = [a_{i,k}]$ con $i = 1, \ldots, n$ y $k = 1, \ldots, p$
- $B = [b_{k,j}]$ con $k = 1, \ldots, p$ y $j = 1, \ldots, m$

El producto de estas dos matrices es una nueva matriz $C \in \mathcal{M}_{n\times m}$ definida de la forma

$$c_{i,j} = \sum_{k=1}^{p} a_{i,k} \cdot b_{k,j} \qquad \text{con } i = 1,\ldots,n;\ j = 1,\ldots,m \tag{5.3}$$

Un ejemplo de código que permite el producto de matrices, tal y como se ha establecido el algoritmo (5.3), puede verse en el programa 5.13. Primeramente se inicializan las matrices y se escriben en un fichero, para comprobar la entrada de datos. Seguidamente se realiza el producto de matrices (LÍNEAS 44-50), para terminar con la impresión de la matriz producto.

Código 5.13 – *Producto de matrices*

```
1  #include <stdio.h>
2
3  #define M 3
4  #define P 2
5  #define N 3
6
7  int main(void)
8  {
9      int i, j, k;
10
11     int MatA[N][P] = {{1, 2}, {3,  4}, {5, 6}};
12     int MatB[P][M] = {{-4, 3, -7}, {-1, 11, 2}};
13     int MatC[N][M] = {0};
14
15     FILE *fi;
16
17     fi = fopen("Datos.dat", "w");
18
19     fprintf(fi,"Matriz A: \n");
20     for (i = 0; i < M; i++){
21         for(k = 0; k < P; k++){
22             fprintf(fi, "%4i", MatA[i][k]);
23         }
24         fprintf(fi, "\n");
25     }
26
27     fprintf(fi, "Matriz B: \n");
28     for (k = 0; k < P; k++){
29         for (j = 0; j < M; j++){
30             fprintf(fi, "%4i", MatB[k][j]);
31         }
32         fprintf(fi, "\n");
33     }
34
35     fprintf(fi, "Matriz producto C, inicial: \n");
36     for (i = 0; i < N; i++){
37         for(j = 0; j < M; j++){
38             fprintf(fi, "%4i", MatC[i][j]);
39         }
40         fprintf(fi, "\n");
41     }
42
```

```
43      //Producto de matrices:
44      for (i = 0; i < N; i++){
45          for(j = 0; j < M; j++){
46              for(MatC[i][j] = 0, k = 0; k < P; k++){
47                  MatC[i][j] += MatA[i][k] * MatB[k][j];
48              }
49          }
50      }
51
52      //Salida de resultados
53      fprintf(fi,"\n ********* RESULTADO ******* \n\n");
54      fprintf(fi, "Matriz Producto C:\n");
55      for(i = 0; i < N; i++){
56          for(j = 0; j < M; j++){
57              fprintf(fi, "%4i", MatC[i][j]);
58          }
59          fprintf(fi, "\n");
60      }
61
62      fclose(fi);
63
64      return 0;
65  }
```

y la salida se almacena en el fichero `Datos.dat`, siendo:

```
Matriz A:
   1   2
   3   4
   5   6
Matriz B:
  -4   3  -7
  -1  11   2
Matriz producto C, inicial:
   0   0   0
   0   0   0
   0   0   0

 ********* RESULTADO *******

Matriz Producto C:
  -6  25  -3
 -16  53 -13
 -26  81 -23
```

5.5. Estructuras

Hasta ahora las estructuras de datos utilizadas tenían la particularidad de ser *homogéneas* (todos las variables eran del mismo *tipo*). Con el nombre de *estructura* se hace referencia a una colección de variables, *no necesariamente todas* del mismo *tipo*, agrupadas bajo un mismo *identificador o nombre*. Se trata entonces, de una estructura de datos *heterogénea*.

En el **lenguaje C** las variables *tipo* ***estructura*** se declaran mediante la palabra **`struct`**. El modelo o patrón para definir estas *estructuras* tiene el siguiente aspecto:

```
      struct nombre_estructura {
```

```
        declaración var_1;
        declaración var_2;
               ...
               ...
        declaración var_n;
    };
```

Supongamos que se quiere hacer una lista de los alumnos de la asignatura de Informática. Se define una estructura llamada `alumno`. En ella se declaran las variables: `Apellido`, `Nombre`, `N_mat`, `Notas` y `Nota_f`. Debido a que son variables con diferente propósito, serán variables de diferente *tipo*. Veamos el ejemplo:

```
    struct alumno {
        int N_mat;
        char Apellido[35];
        char Nombre[15];
        float Notas[4];
        float Nota_f;
    };
```

Cada una de estas variables recibe el nombre de ***miembro*** de la *estructura*. Este código crea o define el *tipo* de dato **`alumno`**. Para poder declarar las variables de *tipo* **`alumno`**, en **C** debe utilizarse el identificador o nombre de las variables, junto con las palabras **`struct`** y **`alumno`**. Ejemplo:

```
    struct alumno estudiante1, estudiante2;
```

Estas dos variables **`estudiante1`** y **`estudiante2`**, son cada una de ellas una *estructura* que agrupan una variable entera: `N_mat`; dos cadenas de caracteres: `Apellido` de hasta 35 caracteres y `Nombre` de hasta 15 caracteres, un vector real de 4 componentes `Notas[4]` y un número real `Nota_f`, respectivamente. También se podían haberse declarado ambas variables, al mismo tiempo que la definición de la estructura. Por ejemplo, en el caso:

```
    struct alumno {
        int N_mat;
        char Apellidos[35];
        char Nombre[15];
    } estudiante1, estudiante2;
```

Para acceder a los miembros de una *estructura* se utiliza el ***operador punto:*** $\boxed{\cdot}$, precedido por el *identificador de la estructura* y seguido del nombre del *miembro*. Por ejemplo, para asignar el número de matrícula 12.234 al primer estudiante, se procedería de la forma:

```
  estudiante1.N_mat = 12234;
```

y para asignar los apellidos:

```
    estudiante1.Apellidos = "Alonso Castro";
```

y análogamente, el nombre:

```
 estudiante1.Nombre = "Rosendo";
```

La lectura se hace de forma análoga a la realizada para las variables. Por ejemplo:

```
scanf("%d", &estudiante1.N_mat);
```

o bien

```
gets(estudiante1.Apellidos);
```

Un simple ejemplo de manipulación de estructuras se observa en el programa 5.14

Código 5.14 – *Estructuras*

```
#include <stdio.h>
/* Ejemplo de programa con estructuras */
int main(void)
{
      struct alumno{
             int N_mat;
             char Apellidos[35];
             char Nombre[15];
             };
      struct alumno estudiante1;
      struct alumno estudiante2 ={1321, "Santiago Barroso", "Ana Luisa"};

      FILE *fi;
      fi = fopen("Datos.dat", "w");

      printf("Introduce el Número de matrícula -> ");
      scanf("%d", &estudiante1.N_mat);
      while (getchar() != '\n'); /* Para vaciar el buffer */
      printf("\n\n");
      printf("Introduce los Apellidos -> ");
      gets(estudiante1.Apellidos);
      printf("\n\n");
      printf("Introduce el Nombre -> ");
      gets(estudiante1.Nombre);

      fprintf(fi, " Los datos introducidos son: \n\n");
      fprintf(fi, "* Número de matrícula: %d\n", estudiante1.N_mat);
      fprintf(fi, "* Apellidos: %s \n", estudiante1.Apellidos);
      fprintf(fi, "* Nombre: %s \n", estudiante1.Nombre);
      fprintf(fi, "\n");
      fprintf(fi, " Los datos introducidos son: \n\n");
      fprintf(fi, "* Número de matrícula: %d\n", estudiante2.N_mat);
      fprintf(fi, "* Apellidos: %s \n", estudiante2.Apellidos);
      fprintf(fi, "* Nombre: %s \n", estudiante2.Nombre);

      fclose(fi);

      return 0;
}
```

y la salida que se obtiene es:

```
 Los datos introducidos son:

* Número de matrícula: 21345
* Apellidos: Alonso Castro
* Nombre: Rosendo

 Los datos introducidos son:

* Número de matrícula: 1321
* Apellidos: Santiago Barroso
* Nombre: Ana Luisa
```

En la LÍNEA 11 se ha procedido a inicializar la variable **estudiante2** del *tipo* **alumno**.

Los *miembros* de las estructuras pueden variables de cualquier *tipo*, además de estar formados por otros datos estructurados como *vectores* o *matrices* e incluso otras *estructuras* previamente definidas. Por ejemplo:

```
      struct alumno otro_alumno, Grupo_Tarde[90];
```

En este caso, **otro_alumno** es una *estructura de tipo alumno* y **Grupo_Tarde[90]** es un *vector de estructuras* en la que cada componente es una *estructura de tipo alumno*. El *miembro* `Apellidos` del alumno 32 vendría representado de la forma:

```
Grupo_Tarde[32].Apellidos;
```

Las *estructuras* permiten ciertas operaciones globales que por su propia naturaleza *no se pueden realizar* con los *vectores y matrices*, en general con los *arrays*. Por ejemplo:

```
Grupo_Tarde[28] = otro_alumno;
```

hace que se copien todos los *miembros* de la estructura **otro_alumno** en los *miembros* correspondientes de la estructura **Grupo_Tarde[28]**. Esta asignación no es posible con los datos estructurados *arrays*.

Es posible además, que los *miembros* de una estructura sea a su vez *estructuras*, previamente definidas. Por ejemplo, se considera el programa 5.15, que define una *estructura anidada*, que a su vez se asocia a un vector. Se trata por tanto, de una muestra de manejo de una *estructura anidada* junto con un *vector de estructuras*.

Código 5.15 – *Estructuras*

```
1  #include <stdio.h>
2  /* Ejemplo de programa con estructuras */
3  int main(void)
4  {
5        int i;
6        struct alumno{
7               int N_mat;
8               char Apellidos[35];
9               char Nombre[15];
10              };
11
12       struct informatica{
13             struct alumno Iden;
14             float Notas[2];
15             float N_f;
16             };
```

```
17        struct informatica Grupo_M[3];
18
19
20        FILE *fi;
21        fi = fopen("Datos.dat", "w");
22
23        for(i = 0; i<=2; i++){
24              printf("Introduce el Número de matrícula -> ");
25              scanf("%d", &Grupo_M[i].Iden.N_mat);
26              while (getchar() != '\n'); /* Para vaciar el buffer */
27              printf("\n\n");
28              printf("Introduce los Apellidos -> ");
29              gets(Grupo_M[i].Iden.Apellidos);
30              printf("\n\n");
31              printf("Introduce el Nombre -> ");
32              gets(Grupo_M[i].Iden.Nombre);
33              printf("Introduce las notas -> ");
34              scanf("%f %f", &Grupo_M[i].Notas[0], &Grupo_M[i].Notas[1]);
35              while (getchar() != '\n'); /* Para vaciar el buffer */
36              };
37        for(i = 0; i<= 2; i++)
38              Grupo_M[i].N_f = 0.5 * (Grupo_M[i].Notas[0] + Grupo_M[i].Notas[1]);
39        fprintf(fi, " La salida es: \n\n");
40        for (i = 0; i<=2; i++){
41            fprintf(fi, "* Mat: %d\n", Grupo_M[i].Iden.N_mat);
42            fprintf(fi, "* Apellidos: %s \n", Grupo_M[i].Iden.Apellidos);
43            fprintf(fi, "* Nombre: %s \n", Grupo_M[i].Iden.Nombre);
44            fprintf(fi,"* Nota final: %f\n", Grupo_M[i].N_f);
45            fprintf(fi, "\n\n");
46            }
47        fclose(fi);
48        return 0;
49  }
```

Para los datos introducidos se obtiene el siguiente resultado

```
La salida es:

* Mat: 21345
* Apellidos: Mestre Amor
* Nombre: Roberto
* Nota final: 6.800000


* Mat: 43215
* Apellidos: Beter Patro
* Nombre: Mar
* Nota final: 4.200000


* Mat: 32569
* Apellidos: Santos Puertas
* Nombre: Luis
* Nota final: 7.950000
```

5.6. typedef

La palabra **typedef** permite renombrar los *tipos* con un nombre arbitrario, escogido por el usuario. Supongamos que se desea que **real** sustituya a **float**. En este caso se haría:

```
typedef float real;
```

A partir de este momento se puede utilizar **real** para declarar variables:

```
real x, vec[30], *Ptr;
```

Para el caso de las *estructuras* resulta muy útil:

```
typedef struct alumno E_alumno
```

se podría entonces declarar:

```
E_alumno estudiante1, estudiante2, Grupo[100];
```

Todas estas variables serían ahora del *tipo estructura* **alumno**, definida anteriormente.

5.6.1. Números complejos

Introducción

Sea el conjunto de pares ordenados $(a, b) \in \mathbb{R} \times \mathbb{R}$ para los que se establecen las siguientes *operaciones de composición interna*:

- ***Suma:*** $(a, b) + (c, d) = (a + c, b + d)$
- ***Producto:*** $(a, b) \cdot (c, d) = (ac - bd, ad + bc)$

Este conjunto de pares ordenados con estas operaciones tienen unas propiedades, que se resumen diciendo que tiene estructura algebraica de **cuerpo** y recibe el nombre de ***cuerpo de los números complejos*** $\mathbb{C}$. Así pues, cualquier elemento de este conjunto se representa como $z = (a, b) \in \mathbb{C}$.

A partir de esta definición se realizan las identificaciones: $(x, 0) \equiv x \in \mathbb{R}$ y $(0, 1) \equiv i$. Con lo cual, aplicando estas identificaciones, junto con la definición de producto se tiene:

$$(y, 0) \cdot (0, 1) = (0, y) \equiv yi$$

Esto permite expresar un complejo de la forma:

$$z = (x, y) = (x, 0) + (0, y) = (x, 0) + (y, 0) \cdot (0, 1) \equiv x + yi$$

que es la llamada ***forma binómica*** de un complejo z. Además por definición, si $z = x + iy$ se utiliza la siguiente nomenclatura

$$\mathrm{Re}\,(z) = x \in \mathbb{R} \text{ Parte real} \qquad \mathrm{Im}\,(z) = y \in \mathbb{R} \text{ Parte imaginaria}$$

Operaciones básicas con números complejos son:

1. **Suma.** Sean $z_1 = x_1 + y_i i$ y $z_2 = x_2 + y_2 i$, entonces

$$z_1 + z_2 = (x_1 + x_2) + (y_1 + y_2)i$$

2. **Producto.** Sean $z_1 = x_1 + y_i i$ y $z_2 = x_2 + y_2 i$, entonces

$$z_1 \cdot z_2 = (x_1 x_2 - y_1 y_2) + (x_1 y_2 + x_2 y_1) i$$

3. **Conjugación.** Sea $z = x + yi$, entonces

$$\overline{z} = x - yi$$

4. **Producto por su conjugado.** Sea $z = x + yi$, entonces

$$z \cdot \overline{z} = (x + yi) \cdot (x - yi) = x^2 + y^2 \equiv |z|^2$$

5. **Cociente.** Sean $z_1 = x_1 + y_1 i$ y $z_2 = x_2 + y_2 i$, entonces

$$\frac{z_1}{z_2} = \frac{\overline{z_2} \cdot z_1}{\overline{z_2} \cdot z_2} = \frac{\overline{z_2} \cdot z_1}{|z_2|^2}$$

Estructura `complejo`

El concepto de número complejo, se puede abordar en **C** a partir del concepto de *estructura*. Un número complejo puede definirse a través de una *estructura* con dos miembros: uno correspondiente a la *parte real* y otro a la *parte imaginaria*. Si además se utiliza la instrucción **`typedef`** una definición del *tipo* **complejo** podría establecerse de la forma:

```
struct estruct_complejo {
    double real;
    double imag;
};
typedef struct estruct_complejo  complejo;

complejo z, z1 = {0.5, 3.};
```

o bien de la manera:

```
typedef struct {
    double real;
    double imag;
} complejo;

complejo z, z1 = {0.5, 3.};
```

Un ejemplo de utilización de este tipo de *estructura* puede verse en el código 5.16, en el que declaran dos variables complejas: z y z_1, una de las cuales está inicializada y los valores de la otra se introduce por teclado. A continuación, calcula la distancia entre ambos y la salida la escribe en el fichero `Datos.dat`.

Código 5.16 – *Ejemplo de estructura complejo*

```
1 #include <stdio.h>
2 #include <math.h>
3 /* Ejemplo de estructura complejos */
4
5 int main(void)
6 {
```

```
7       typedef struct {
8           double real;
9           double imag;
10      } complejo;
11
12      complejo z, z1 = {0.5, 3.};
13      double dist;
14
15      FILE *fi;
16
17      fi = fopen("Datos.dat", "w");
18
19      // ******* Lectura de datos de la matriz *******
20      printf("\n\n Introduce las componentes del número complejo: \n");
21      printf("\n Introduce la componente real -> ");
22      scanf("%lf", &z.real);
23      while(getchar() != '\n');
24
25      printf("\n Introduce la componente imaginaria -> ");
26      scanf("%lf", &z.imag);
27      while(getchar() != '\n');
28
29      fprintf(fi, "\n El numero complejo introducido es %.4f + %.4f i\n",
30                                                      z.real, z.imag);
31      fprintf(fi, "\n El numero complejo z es %.4f + %.4f i\n",
32                                          z1.real, z1.imag);
33        /**********************/
34
35      dist = sqrt((z.real - z1.real) * (z.real - z1.real) +
36                  (z.imag - z1.imag) * (z.imag - z1.imag));
37
38      fprintf(fi, "La distancia  entre ellos  es %.5f\n", dist);
39
40      fclose(fi);
41
42      return 0;
43  }
```

Estructura `complex`

En las últimas actualizaciones realizadas en el **lenguaje C**, ya está disponible un *tipo* de variable *compleja*, que se indentifica con el nombre de **`complex`**. Es la misma idea de estructura definida en el código 5.16, sin embargo esta modalidad de *tipo* que ofrece **C**, tiene más posibilidades y es más completa[5]. Es posible operar con este tipo de variables directamente, sin necesidad de hacer referencia a los miembros de la estructura, como se hizo en el caso anterior. Para poder utilizar este *tipo*, es necesario añadir en el *preámbulo* del programa la directiva **`#include <complex.h>`**. En el código 5.17, se muestra un ejemplo de utilización de este *tipo* para realizar una serie de operaciones elementales con números complejos.

Código 5.17 – *Operaciones con complejos y la estructura* `complex`

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

[5] En el entorno DEV-C++ que se utiliza en este curso, el compilador ya ofrece este *tipo* `complex`.

```c
#include <complex.h>
#include <math.h>

//Operaciones con números complejos

int main(void)
{
    double complex z1 = 1. - 5.*I;
    double complex z2 = -4. + .3*I;

    double complex suma, prod, cociente, conjugado;
    double complex potencia;

    FILE *fi;

    fi = fopen("Datos.dat", "w");

    fprintf(fi, "Números complejos introducidos:\n");
    fprintf(fi, "z1 = %.2f %+.2fi\n", creal(z1), cimag(z1));
    fprintf(fi, "z2 = %.2f %+.2fi\n", creal(z2), cimag(z2));

    suma = z1 + z2;
    fprintf(fi, "La suma z1 + z2 = %.2f %+ .2fi\n",
                            creal(suma), cimag(suma));

    prod = z1 * z2;
    fprintf(fi, "El producto  z1 * z2 = %.2f %+.2fi\n",
                             creal(prod), cimag(prod));

    cociente = z1 / z2;
    fprintf(fi, "El cociente z1/z2 = %.2f %+.2fi \n",
                            creal(cociente), cimag(cociente));

    conjugado = conj(z1);
    fprintf(fi, "El conjugado de z1 =  %.2f %+.2fi \n",
                             creal(conjugado) ,cimag(conjugado));

    potencia = cpow(z1,2 - 0.5*I);
    fprintf(fi, "z1 elevado a 2-0.5i = %.4f %+.4fi \n",
                            creal(potencia), cimag(potencia));

    fclose(fi);

    return 0;
}
```

y la salida en el fichero `Datos.dat` queda de la forma:

```
Números complejos introducidos:
z1 = 1.00 -5.00i
z2 = -4.00 +0.30i
La suma z1 + z2 = -3.00 -4.70i
El producto  z1 * z2 = -2.50 +20.30i
El cociente z1/z2 = -0.34 +1.22i
El conjugado de z1 =  1.00 +5.00i
```

```
z1 elevado a 2-0.5i = -11.9484 +5.3320i
```

En las LÍNEAS 9-10 se aprecia cómo debe introducirse una constante compleja. Es importante, que la i de la parte imaginaria de la expresión binómica, se representa en **C** con **`I`**. Esto es, el número complejo: $2 - \frac{1}{2}i$ se expresa como **`2 - 0.5*I`**. Por último resaltar, que al aplicar la función **`pow`** de la biblioteca, al ser utilizada con el *tipo complejo* debe ponerse como **`cpow`** (véase LÍNEA 39).

5.7. Dimensionamiento dinámico (Variable Length Arrays – VLA)

Hasta ahora, para utilizar un vector o una matriz en un programa se utilizaba el ***dimensionamiento estático*** es decir, era necesario imponer su dimensión al comienzo de programa mediante expresiones constantes. Esto planteaba el inconveniente de que si sólo se deseaba utilizar un vector de 15 componentes, por ejemplo, ¿para qué reservar un espacio para 500? (con la correspondiente *pérdida de espacio en memoria*), o al revés, si se dimensionaba un vector con 15 componentes se obligaba en el programa a no pasarse de este tamaño, con la correspondiente restricción en la ejecución.

En las versiones más recientes de **C** se ha cambiado esta situación, haciendo los programas más flexibles. Ahora es posible dimensionar un vector o una matriz, a partir de variables cuyo valor es introducido por el teclado. Por esto, esta característica recibe el nombre de ***dimensionamiento dinámico*** de vectores y matrices o utilización de *vectores y matrices de tamaño variable* (*Variable–Length Arrays*)[6].

Este nombre puede ser un tanto confuso, porque puede creerse que los *arrays* declarados de esta forma pueden cambiar de tamaño durante la ejecución del programa. Esto **no** es así. Una vez dimensionados, eso sí, durante la ejecución del programa, su tamaño ya permanece fijo hasta el final.

Para ilustrar el procedimiento de *dimensionamiento dinámico* se modifica el código 5.4 (pág. 104), para poder seleccionar, desde el teclado, el tamaño de la tabla de los *números de Fibonacci*.

Código 5.18 – *Sucesión de Fibonacci, direccionamiento dinámico*

```
1  #include <stdio.h>
2
3  /* Construye una tabla con los números de Fibonacci */
4
5  int main(void)
6  {
7
8
9      int N;
10     unsigned int ind;
11
12     FILE *fi;
13
14     fi = fopen("Datos.dat", "w");
15
16     printf("\n\n **** Este programa determina la tabla de los número de
           Fibonacci ****\n\n");
17     printf("Introduce un entero 2 < N -> ");
18     while(scanf("%i", &N) != 1 || N < 3 ){
19         while(getchar() != '\n');  /* limpia el flujo de entrada*/
20             /*Se repite la entrada */
21         printf("\n\nHas de introducir entero positivo 2 < N -> ");
```

[6] El compilador que trae el entorno DEV–C++, que se utiliza en este curso, permite esta posibilidad de *dimensionamiento dinámico*.

```
22      }
23      while(getchar() != '\n');
24
25
26      // Dimensionamiento dinámico
27      unsigned long long TablaFibo[N];
28
29      //Inicialización
30      TablaFibo[0] = 0ULL;
31      TablaFibo[1] = 1ULL;
32
33      // Construye los términos de la sucesión
34      for(ind = 2; ind < N; ind++){
35          TablaFibo[ind] = TablaFibo[ind-2] + TablaFibo[ind-1];
36      }
37
38      /* Imprime la tabla  */
39      fprintf(fi, "Los %i primeros números de Fibonacci son: \n\n", N);
40      for (ind = 0; ind < N; ind++){
41          fprintf(fi, "%llu ; ", TablaFibo[ind]);
42      }
43
44      fclose(fi);
45
46      return 0;
47
48  }
```

En la LÍNEA 27 se genera el *dimensionamiento dinámico*. El tamaño del vector `TablaFibo` se adapta a la cantidad de números de la *sucesión de Fibonacci* que se desean. Así pues, como ya no existe la limitación inicial de 30 componentes, como consecuencia del dimensionamiento *fijo* del código 5.4, se ha eliminado esta restricción de la entrada de datos. Por otra parte, es importante señalar, cómo se realiza la inicialización. Con el *dimensionamiento dinámico* ya **no** se permiten inicializaciones del tipo

```
        //   Dimensionamiento dinámico
    unsigned long long TablaFibo[N] = {0ULL, 1ULL};
```

similar a la que se hacía en la LÍNEA 10 del código 5.4 (pág. 104). Ahora la inicialización se debe hacer según se muestra en las LÍNEAS 30-31 del código 5.18.

Hay que tener en cuenta, que una vez seleccionada *dinámicamente* el tamaño del vector o matriz, ya no es posible volver a cambiar esta dimensión, durante la ejecución del programa.

5.8. Problemas

1. Dado un vector real de doble precisión:

$$\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$$

se pide, que realices un programa que permita:

a) Calcular el valor de las normas:

$$\|\boldsymbol{x}\|_2 = \sqrt{\boldsymbol{x}\cdot\boldsymbol{x}}; \qquad \|\boldsymbol{x}\|_1 = \sum_{i=1}^{n} |x_i|; \qquad \|\boldsymbol{x}\|_\infty = \max_{1\leqslant i\leqslant n} |x_i|$$

b) Invertir las componentes del vector inicial.

c) Ordenar de menor a mayor, las componentes del vector inicial.

Una ejecución típica del programa debe ser:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 1111111
**************************

Introduce la dimensión del vector -> -1
Por favor introduce un número positivo !!!! -> 5
Introduce V[1] = 0
Introduce V[2] = -1
Introduce V[3] = 1
Introduce V[4] = 3
Introduce V[5] = -6
```

La salida típica en un fichero de datos debe ser:

```
El vector introducido  es:
V[1] = 0.00; V[2] = -1.00; V[3] = 1.00; V[4] = 3.00; V[5] = -6.00;
La norma 2 es: 6.85565

La norma 1 es: 11.00000

La norma infinito es: 6.00000

El vector invertido es:
V[1] = -6.00; V[2] = 3.00; V[3] = 1.00; V[4] = -1.00; V[5] = 0.00;

El vector ordenado es:
V[1] = -6.00; V[2] = -1.00; V[3] = 0.00; V[4] = 1.00; V[5] = 3.00;
```

2. Escribe un programa que abra un fichero con un nombre especificado por el usuario y añada lo que el usuario teclea.

3. Dada una cadena de caracteres escribe un programa que cuente el número de caracteres e invierta la frase (debe aparecer escrita al revés). Una ejecución típica del programa debe ser:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 1111111
**************************

Introduce el nombre del fichero de salida -> Datos.dat

El nombre del fichero introducido es: Datos.dat

Introduce la frase -> El valle estaba bonito y florido
```

La salida típica en un fichero de datos debe ser:

```
La frase introducida es ' El valle estaba bonito y florido '

La frase contiene 32 caracteres.

La frase invertida es ' odirolf y otinob abatse ellav lE '.
```

(Propuesto en examen: curso 2014/15).

4. ***Producto de dos polinomios.*** Sean los polinomios

$$p(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$$
$$q(x) = b_m x^m + b_{m-1}x^{m-1} + \cdots + b_1 x + b_0$$

El polinomio producto

$$p(x) \cdot q(x) = r_{n+m}x^{n+m} + r_{n+m-1}x^{n+m-1} + \cdots + r_1 x + r_0$$

tiene por coeficientes:

$$r_k = \sum_{i+j=k} a_i b_j \qquad \text{con} \quad k = 0, 1, \ldots, n+m$$

Se pide que hagas un programa que calcule el producto de dos polinomios cuyos coeficientes se introducirán por el teclado y el polinomio producto resultante, se imprimirá en un fichero de datos, cuyo nombre se deberá introducir por teclado, previamente.

5. Realiza una tabla de números primos. Se debe introducir un número natural n y escribir en un fichero, la tabla de todos los primos menores o iguales que n, así como la cantidad de números primos que hay. Un ejemplo de ejecución sería:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 12345
**************************

Introduce un número entero positivo mayor que 6 -> 25
```

y el resultado

```
Hay 10 números primos menores o iguales que 25 y son:

1 ; 2 ; 3 ; 5 ; 7 ; 11 ; 13 ; 17 ; 19 ; 23
```

6. Dada una matriz $\mathbf{A} \in \mathcal{M}_{m\times n}$ haz un programa que imprima su matriz transpuesta en un fichero de datos. A continuación, si $\mathbf{A}$ es una matriz cuadrada de dimensión n, haz un programa que determine su transpuesta y la almacene en las mismas posiciones de memoria que la matriz de entrada (obviamente los datos de la matriz de entrada cambiarán por los coeficientes de la matriz transpuesta). Si quieres puedes utilizar el *dimensionamiento dinámico*.

7. ***Producto diádico o tensorial de dos vectores.***
A partir de dos vectores $\boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^n$ se puede construir una matriz de la forma

$$[\boldsymbol{a} \otimes \boldsymbol{b}] = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \begin{bmatrix} b_1 & b_2 & \cdots & b_n \end{bmatrix} = \begin{bmatrix} a_1 b_1 & a_1 b_2 & \cdots & a_1 b_n \\ a_2 b_1 & a_2 b_2 & \cdots & a_2 b_n \\ \vdots & \vdots & \ddots & \vdots \\ a_n b_1 & a_n b_2 & \cdots & a_n b_n \end{bmatrix} \tag{5.4}$$

En un espacio vectorial euclídeo (como $(\mathbb{R}^n, \cdot)$), esta operación se llama ***producto diádico o tensorial*** de $\boldsymbol{a}$ y $\boldsymbol{b}$ y da lugar a una *forma bilineal* que recibe el nombre de *tensor* y opera de la forma

$$(\boldsymbol{a} \otimes \boldsymbol{b}) \cdot \boldsymbol{v} = (\boldsymbol{b} \cdot \boldsymbol{v})\, \boldsymbol{a} \qquad \forall\, \boldsymbol{v} \in \mathbb{R}^n$$

Aplicando esta definición, se obtienen las componentes de este *producto diádico*

$$[\boldsymbol{a} \otimes \boldsymbol{b}]_{ij} = \boldsymbol{e}_i \cdot ((\boldsymbol{a} \otimes \boldsymbol{b}) \cdot \boldsymbol{e}_j) = \boldsymbol{e}_i \cdot ((\boldsymbol{b} \cdot \boldsymbol{e}_j)\, \boldsymbol{a}) = (\boldsymbol{b} \cdot \boldsymbol{e}_j)\, \boldsymbol{a} \cdot \boldsymbol{e}_i = a_i\, b_j$$

y que matricialmente coincide con la expresión (5.4).

Se pide que realices un programa, tal que dados dos vectores, calcule su *producto diádico* que se expresará matricialmente en un fichero de datos, cuyo nombre debe ser introducido desde el teclado.

8. Escribe un programa que calcule $\|\mathbf{A}\|_\infty$ y $\|\mathbf{A}\|_1$ con $\mathbf{A} \in \mathcal{M}_{n\times m}$.

$$\|\mathbf{A}\|_\infty = \max_{1 \leqslant i \leqslant n} \sum_{j=1}^{m} |a_{i,j}| \qquad \|\mathbf{A}\|_1 = \max_{1 \leqslant j \leqslant m} \sum_{i=1}^{n} |a_{i,j}|$$

9. Escribe un programa que determine si una matriz cuadrada es *diagonal dominante*. Se dice que una matriz cuadrada es *diagonal dominante* si

$$|a_{i,i}| \geqslant \sum_{\substack{j=1 \\ i \neq j}}^{n} |a_{i,j}|$$

10. Se pide que realices un programa que permita descomponer una matriz cuadrada real de tamaño n, $\mathbf{A}$ en suma de una matriz simétrica $\mathbf{S}$ y una matriz antisimétrica $\mathbf{T}$, utilizando la siguiente expresión:

$$\mathbf{A} = \underbrace{\frac{1}{2}\left(\mathbf{A} + \mathbf{A}^t\right)}_{\mathbf{S}} + \underbrace{\frac{1}{2}\left(\mathbf{A} - \mathbf{A}^t\right)}_{\mathbf{T}}$$

en donde $\mathbf{A}^t$ es la matriz traspuesta de $\mathbf{A}$. Una ejecución típica del programa debe ser:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 1111111
**************************

Introduce el tamaño de la matriz -> 3

Introduce A[1][1] = 1
Introduce A[1][2] = 2
```

```
Introduce A[1][3] = 3

Introduce A[2][1] = 4
Introduce A[2][2] = 5
Introduce A[2][3] = 6

Introduce A[3][1] = 7
Introduce A[3][2] = 8
Introduce A[3][3] = 9
```

La salida típica en un fichero de datos debe ser:

```
La matriz introducida es:
 A[1][1] = 1.00;  A[1][2] = 2.00;  A[1][3] = 3.00;
 A[2][1] = 4.00;  A[2][2] = 5.00;  A[2][3] = 6.00;
 A[3][1] = 7.00;  A[3][2] = 8.00;  A[3][3] = 9.00;

La matriz simétrica es:
 S[1][1] = 1.00;  S[1][2] = 3.00;  S[1][3] = 5.00;
 S[2][1] = 3.00;  S[2][2] = 5.00;  S[2][3] = 7.00;
 S[3][1] = 5.00;  S[3][2] = 7.00;  S[3][3] = 9.00;


La matriz anti-simétrica es:
 T[1][1] = 0.00;  T[1][2] = -1.00;  T[1][3] = -2.00;
 T[2][1] = 1.00;  T[2][2] = 0.00;  T[2][3] = -1.00;
 T[3][1] = 2.00;  T[3][2] = 1.00;  T[3][3] = 0.00;


La matriz suma debe coincidir con la inicial:
 S[1][1] + T[1][1] = 1.00;  S[1][2] + T[1][2] = 2.00;  S[1][3] + T[1][3] = 3.00;
 S[2][1] + T[2][1] = 4.00;  S[2][2] + T[2][2] = 5.00;  S[2][3] + T[2][3] = 6.00;
 S[3][1] + T[3][1] = 7.00;  S[3][2] + T[3][2] = 8.00;  S[3][3] + T[3][3] = 9.00;
```

(Propuesto en examen: curso 2014/15).

11. Resuelve la ecuación de segundo grado utilizando el *tipo* **`complex`**, para la solución.

12. Haz un programa que pida un número complejo por teclado y compruebe la igualdad:

$$e^{iz} = \cos z + i \operatorname{sen} z$$

13. ***Perímetro de un polígono.***

Un polígono $[p_1, \ldots, p_n]$ en el plano, con n lados y n vértices $p_1 = (x_1, y_1), \ldots, p_n = (x_n, y_n)$ es una *curva cerrada simple* que consiste en los segmentos que unen vértices consecutivos. Se ha de entender que p_1 es el vértice siguiente a p_n. Si se define

$$d(p_i, p_{i+1}) = \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}$$

entonces el perímetro del polígono es

$$\text{Perímetro}([p_1, \ldots, p_n]) = \sum_{i=1}^{n} d(p_i, p_{i+1})$$

entendiendo que el siguiente al índice n se toma como 1, en lugar de $n + 1$, como ya se ha comentado.

Escribe un programa que pida al usuario los vértices de un polígono plano y que calcule el perímetro del polígono introducido. Para hacer el ejercicio, se debe definir un vector de *tipo* **`punto`**, en la que en cada componente es una *estructura* que contiene las respectivas coordenadas de cada vértice del polígono. Un ejemplo de ejecución sería:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 1111111
**************************

Introduce el número de vértices ( > 1) -> 5

Introduce los vértices del polígono:
Introduce X[1] = 0
Introduce Y[1] = 0

Introduce X[2] = 2
Introduce Y[2] = 1

Introduce X[3] = 3
Introduce Y[3] = 4

Introduce X[4] = 0
Introduce Y[4] = 3

Introduce X[5] = 1
Introduce Y[5] = 2
```

y la salida, en un fichero, debe ser

```
Se ha introducido el polígono con vértices:
X[1] = 0.00  Y[1] = 0.00
X[2] = 2.00  Y[2] = 1.00
X[3] = 3.00  Y[3] = 4.00
X[4] = 0.00  Y[4] = 3.00
X[5] = 1.00  Y[5] = 2.00


El perímetro es 12.21090
```

CAPÍTULO 6

Punteros

6.1. Introducción

Los ***punteros*** son variables que almacenan direcciones de memoria. Esto permite desarrollar programas a *bajo nivel* que pueden resultar más flexibles y eficientes. Sin embargo, tiene el inconveniente de hacer también a los programas muy susceptibles, de generar errores de ejecución, si estas variables no se operan correctamente.

En este capítulo se tratará principalmente con *punteros a enteros* como ejemplo, pero todos los resultados y planteamientos pueden generalizarse de manera inmediata a *puntero* de cualquier *tipo*, con sólo intercambiar el nombre.

6.2. Variables y punteros

Ya se ha comentado (véase sección 1.3.3 en la pág. 7) que la memoria de un ordenador está *agrupada* en *bytes* y la posición que ocupa cada uno de estos *bytes* se identifican mediante un número llamado *dirección* (de memoria). Estos grupos pueden estar formados por un sólo *byte* (en el caso de almacenar un carácter), o por más *bytes*, dependiendo del *tipo* de variable que se declare (**`short`**, **`int`**, **`long`**, **`long long`**, **`float`**, **`double`**, **`...`**) tal y como se ha visto en la sección 3.2 en la pág. 25. Pues bien, los ***punteros*** (del inglés *pointers*) son variables que permiten operar con estas direcciones de memoria. La manera de *declarar* este tipo de variables, se muestra a continuación:

```
int *PtrEnt;            //*** Puntero a una variable entera
double *DirecDoble;     //*** Puntero a una variable real double
long long *PosLargo;    //*** Puntero a una variable entera de 8 bytes
float *PunteroReal;     //*** Puntero a una variable real de 4 bytes
```

Para trabajar con las direcciones de memoria, hay dos operadores básicos

Operador dirección (`&`). Dada una variable es posible conocer su *dirección* mediante el operador **`&`**. Si `entero` es una variable entera con cierto valor, entonces mediante la expresión

```
&entero;
```

estaría haciendo referencia a la dirección que ocupa en memoria la variable `entero`.

Operador indirección (`*`). Si `ptr` es una variable declarada como *puntero a entero* entonces esta variable sólo almacena una dirección. Si se ha depositado un valor en esta dirección, es posible saber cuál es éste, mediante la expresión

```
*ptr;
```

En este caso, el asterisco `*` tiene una interpretación distinta al utilizado en la *declaración* de la variable.

En el programa 6.1 se muestra un primer ejemplo del comportamiento de un *puntero*:

Código 6.1 – *Primer ejemplo con Punteros*

```
1  #include <stdio.h>
2  /* Ejemplo con punteros */
3  int main(void)
4  {
5        int prueba = 35;
6        int *ptr;    // **** Declaración de puntero a entero
7
8        ptr = &prueba;
9        printf("*ptr = %d, ptr = %u, &ptr = %u\n", *ptr, ptr, &ptr);
10       printf("prueba = %d, &prueba = %u \n", prueba, &prueba);
11
12       return 0;
13 }
```

En este caso, se declara un *puntero a entero* (LÍNEA 6)[1] y almacena la dirección de la variable **`prueba`** (LÍNEA 8). Así pues, **`*ptr`**: almacena el valor de **`prueba`**; **`ptr`**: almacena la dirección de **`prueba`** y **`&ptr`**: muestra la dirección de la variable **`ptr`**, que al fin y al cabo, es otra variable y por tanto, **C** le asigna una dirección. La salida en pantalla se muestra a continuación:

```
*ptr = 35, ptr = 2293324, &ptr = 2293312
prueba = 35, &prueba = 2293324
```

Una representación gráfica de la salida del programa 6.1 puede verse en la figura 6.1. Una variable puntero,

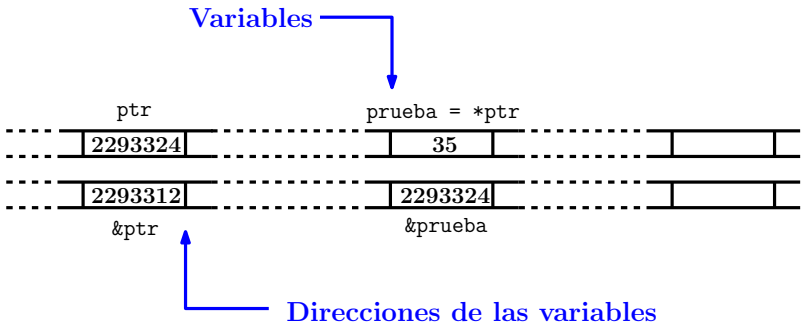


Figura 6.1 – *Representación de la salida del programa 6.1*

(cualquiera que sea el *tipo* a la que esté asociada) se suele identificar con el *formato* **`%p`**. Sin embargo, esta opción hace que la dirección de memoria se exprese en *hexadecimal*. En este curso, para que resulte más clara

[1] De manera análoga se podría haber declarado otro tipo de *puntero*, bien *float, char, long long, double*, ...

la dirección de memoria, se utilizará el formato **%u** (asociado a un *tipo unsigned*), que permite su expresión en forma *decimal*.

Para declarar una variable como *puntero* se debe conocer el tipo de variable a la que está apuntando dicho *puntero*. La razón es que las variables de distintos *tipos* ocupan diferentes cantidades de memoria y existen operaciones con punteros que requieren conocer el tamaño de almacenamiento. A continuación se muestra un ejemplo con diferentes *tipos* de *punteros*

Código 6.2 – *Segundo ejemplo con Punteros*

```
#include <stdio.h>
/* Ejemplo2 con punteros */
int main(void)
{

    char *ptrchar; //*** Declaración puntero

    int entero, cambio = 9;
    int *ptr, *otroptr; //*** Declaración puntero

    float real;
    float *ptrfloat; //*** Declaración puntero

    double realargo;
    double *ptrdouble; //*** Declaración puntero


    entero = 7;
    ptr = &entero;
    printf("entero -> %i, direccion -> %u <-> %p (hex.)\n", entero, ptr, ptr);

    *ptr = 14;
    cambio = *ptr;
    printf("entero -> %i; direccion -> %u; ", entero, ptr);
    printf("cambio -> %i \n", cambio);

    otroptr = ptr;
    *otroptr = 106; /*cambia el valor de la v. entero*/
    printf("otroptr -> %u; ptr -> %u; \n", otroptr, ptr);
    printf("entero -> %i; \n\n", entero);

    real = 2.595;
    realargo = 246.8932;

    ptrfloat = &real;
    ptrdouble = &realargo;
    printf("real -> %f; ptrfloat -> %u; \n", real, ptrfloat);
    printf("realargo -> %f; ptrdouble -> %u; ", realargo, ptrdouble);

    printf("Tamaño de ptrchar -> %u; bytes\n", sizeof(char *));
    printf("Tamaño de ptr -> %u bytes\n", sizeof(int *));
    printf("Tamaño de ptrfloat -> %u bytes\n", sizeof(float *));
    printf("Tamaño de ptrdouble -> %u bytes\n", sizeof(double *));


}
```

```
46      return 0;
47  }
```

La salida en pantalla toma la forma

```
entero -> 7, direccion -> 2293276 <-> 000000000022FE1C (hex.)
entero -> 14; direccion -> 2293276; cambio -> 14
otroptr -> 2293276; ptr -> 2293276;
entero -> 106;

real -> 2.595000; ptrfloat -> 2293272;
realargo -> 246.893200; ptrdouble -> 2293264; Tamaño de ptrchar -> 8; bytes
Tamaño de ptr -> 8 bytes
Tamaño de ptrfloat -> 8 bytes
Tamaño de ptrdouble -> 8 bytes
```

Es interesante destacar cómo en las LÍNEAS 6, 9, 12 y 15 aparecen las declaraciones de variables *puntero* a **`char, int, float`** y **`double`** respectivamente. Además, en la LÍNEA 19 mediante el operador **`&`** se introduce la dirección de la variable `entero`, en el puntero `ptr`. A continuación, en la LÍNEA 22 se utiliza el *operador indirección* para cambiar el valor de la variable `entero` al valor 14 y que posteriormente, se introduce en la variable `cambio`, también mediante el operador **`*`**. Esta última sentencia es equivalente a:

```
        cambio = entero;
```

En las LÍNEAS 27-28 se utiliza un nuevo puntero, para cambiar el valor de la variable `entero` a 106. En la última parte del programa 6.2 se ha obtenido el tamaño de las variables *puntero*. Todas ellas, ocupan 8 *bytes*[2].

6.2.1. Aritmética básica de punteros

Como ya se ha visto anteriormente, los *punteros* son variables y en principio, se pueden realizar operaciones aritméticas con ellos. Sin embargo, por la propia naturaleza del concepto de *puntero* (*únicamente contienen direcciones de memoria*) , carecen de sentido las operaciones de producto o división, ni tienen sentido resultados negativos o decimales. Tampoco resulta coherente, realizar operaciones con punteros asociados a *tipos* diferentes. Las operaciones aritméticas básicas, permitidas con punteros asociados a los mismos *tipos* son la ***suma*** y la ***resta***. Si **`ptr`** es una variable puntero, Las siguientes expresiones son correctas en **C**[3]

```
        ptr = ptr + 1;
        ptr = ptr - 2;
        ptr = ptr + contador;
        ptr++;
        ptr--;
```

Se considera el siguiente ejemplo:

Código 6.3 – *Ejemplo de aritmética de punteros*

```
1  #include <stdio.h>
2  /* Ejemplo de aritmética de punteros */
3  int main(void)
```

[2]Este resultado depende del compilador y del sistema operativo. En este caso, se utiliza el entorno DEV–C++ 5.11, cuyo compilador está configurado para trabajar a 64 *bits*. Por esta razón aparece un tamaño de 8 *bytes* en la salida.

[3]Hay que tener en cuenta, que el compilador no controla que con la ejecución de estas operaciones se pueda exceder de la cantidad de memoria direccionable o que la variable tome valores negativos, lo cual lleva irremediablemente a un error de ejecución.

```
4  {
5
6        char c, *ptr_c = &c;
7        int i, *ptr_i = &i;
8        float real, *ptr_f = &real;
9        double realmax, *ptr_d = &realmax;
10
11
12       printf("ptr_c -> %u;  prt_i -> %u;   ", ptr_c, ptr_i);
13       printf("ptr_f -> %u;  prt_double -> %u \n\n", ptr_f, ptr_d);
14
15       ptr_c--, ptr_i++, ptr_f++, ptr_d++;
16       printf("ptr_c-- -> %u;  prt_i++ -> %u;   ", ptr_c, ptr_i);
17       printf("ptr_f++ -> %u;  prt_double++ -> %u \n\n", ptr_f, ptr_d);
18
19       ptr_c += 3, ptr_i -=  5, ptr_f += 2, ptr_d -= 2;
20       printf("ptr_c + 3 -> %u;  prt_i - 5-> %u;   ", ptr_c, ptr_i);
21       printf("ptr_f + 2 -> %u;  prt_double - 2 -> %u \n", ptr_f, ptr_d);
22
23       return 0;
24 }
```

La salida en pantalla es

```
ptr_c -> 2293287;  prt_i -> 2293280;   ptr_f -> 2293276;  prt_double -> 2293264

ptr_c-- -> 2293286;  prt_i++ -> 2293284;   ptr_f++ -> 2293280;  prt_double++ -> 2293272

ptr_c+3 -> 2293289;  prt_i-5 -> 2293264;   ptr_f+2 -> 2293288;  prt_double-2 -> 2293256
```

Merece la pena analizar el resultado obtenido. En la primera fila de la salida, cada puntero tiene almacenada la dirección de sus respectivas variables declaradas, por medio de las sentencias de las LÍNEAS 6-9 del código 6.3. Como ya se ha comentado (véase subsección 1.3.3 en la pág. 7), la memoria del ordenador se agrupa en *bytes*. Los tamaños de estos grupos de *bytes*, depende del *tipo* de variable declarada. Por tanto, los números que se muestran en la primera línea de la salida, corresponden a la dirección del primer *byte*, del grupo de *bytes* que tiene asignado cada *tipo* de variable. De esta forma, la variable **c**, como es del tipo *carácter* y únicamente ocupa un *byte*, el número 2293287 es la dirección de ese *byte*. Sin embargo, en el caso de *puntero a entero*: **ptr_i**, el número que muestra en la salida: 2293280 corresponde al primer *byte* de los cuatro que tiene asignados por ser una dirección a una variable entera de 4 *bytes*. Algo análogo sucede con el puntero **ptr_f**, que almacena la dirección del primer *byte* de los cuatro asignados. En **ptr_double** se indica la dirección del primer *byte* de los ocho que tiene establecidos por ser un *puntero a double*.

Esta disposición, afecta a las operaciones aritméticas: cuando en la LÍNEA 15 se decrementa e incrementa estas variables, resulta que **ptr_c** apunta al *byte* anterior (por ocupar la variable un sólo *byte*, pero al incrementar **ptr_i** en una unidad, su valor aumenta en 4 *bytes*; y correspondería a la posición asignada al siguiente *entero*. Lo análogo sucede con la variable **prt_f**. En el caso de **ptr_double**, al ser un *puntero a double* el incremento en una unidad en la variable, supone un aumento de ocho *bytes* para posicionarlo en la dirección de la siguiente variable *double*. Con las sentencias de la LÍNEA 19, **ptr_c** se incrementa en 3 unidades, con lo que pasa a señalar la dirección del *byte* 2293573; **ptr_i** se decrementa en 5 unidades, con lo cual su dirección pasa a señalar 20 *bytes* anteriores ($20 = 4\,bytes \times 5$); algo similar sucede con **ptr_f**. Por último **ptr_double** se decrementa en 2 unidades, lo que implica que su dirección disminuye en 16 *bytes* ($16 = 8\,bytes \times 2$).

6.3. Vectores y punteros

Existe una estrecha relación entre los *vectores* y *punteros*. De hecho, a un *puntero*, se le puede asociar la dirección de la primera componente de un *vector*. Esto permite manipular las componentes del vector a través de operaciones con el puntero asignado. Es obligado que el puntero y el vector sean del mismo *tipo* de variable. Por ejemplo:

```
int Vtr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
int *ptr;
.....
.....
ptr = &Vtr[0];
.....
.....
```

Mediante la asignación

ptr = &Vtr[0]

no sólo el puntero **ptr** almacena la dirección de la variable **Vtr[0]**, sino que automáticamente, **ptr+1** es la dirección de la variable `Vtr[1]` (es decir que **ptr+1** coincide con **&Vtr[1]**); **ptr+2** almacena la dirección de **Vtr[2]** y así sucesivamente hasta **ptr+9**, que almacena la dirección de **Vtr[9]**. Además, por la propia definición de puntero, se cumplen las igualdades (aritméticas)

***(ptr + i)** $=$ **Vtr[i]**

con $i = 0, \ldots, 9$

A continuación se muestra un sencillo programa en donde se aprecia esta equivalencia.

Código 6.4 – *Programa de vector y puntero*

```
#include <stdio.h>
/* Ejemplo de programa de vectores y punteros */
#define DIM 4
int main(void)
{
    int i;
    int Vtr[DIM] = {28, 29, 30, 31};
    int *ptr;

    FILE *fi;
    fi = fopen("Datos.dat", "w");

    for (i = 0; i < DIM; i++)
        fprintf(fi, "&Vtr[%i] = %u <-> Vtr[%i] = %2d;\n",
                                i, &Vtr[i], i, Vtr[i]);

    fprintf(fi, "\n");

    ptr = &Vtr[0];
    fprintf(fi, "Asignación ptr = &Vtr[0]\n");
        fprintf(fi, "ptr = %u; &ptr = %u\n\n", ptr, &ptr);

    for (i = 0; i < DIM; i++)
```

```
24          fprintf(fi, "ptr + %i = %u <-> *(ptr + %i) = %2d;\n",
25                                  i, ptr+i, i, *(ptr+i));
26      fclose(fi);
27
28      return 0;
29  }
```

En este programa se declara e inicializa un vector de *tipo* **int** (LÍNEA 7) y se escribe su contenido de dos maneras: una es utilizando la estructura habitual de *vector* (LÍNEAS 13-15) y la otra mediante un puntero (LÍNEAS 23-25). En la LÍNEA 8 se declara la variable *puntero a entero*: **ptr** y en la LÍNEA 19 se introduce la *dirección* de **Vtr**, en esta variable. El resultado es el siguiente:

```
&Vtr[0] = 2293296 <-> Vtr[0] =  28;
&Vtr[1] = 2293300 <-> Vtr[1] =  29;
&Vtr[2] = 2293304 <-> Vtr[2] =  30;
&Vtr[3] = 2293308 <-> Vtr[3] =  31;

Asignación ptr = &Vtr[0]
ptr = 2293296; &ptr = 2293288

ptr + 0 = 2293296 <-> *(ptr + 0) =  28;
ptr + 1 = 2293300 <-> *(ptr + 1) =  29;
ptr + 2 = 2293304 <-> *(ptr + 2) =  30;
ptr + 3 = 2293308 <-> *(ptr + 3) =  31;
```

Existe todavía una relación más profunda entre los vectores y los punteros: el *nombre o identificador* de un *vector* es un **puntero** a la dirección de memoria que contiene el primer elemento del vector (realmente es un *puntero a constante* y siendo esta constante el nombre o identificador del vector). Siguiendo con el ejemplo anterior, resulta que **Vtr** contiene un valor que coincide con **&Vtr[0]**. Esto es, **Vtr** almacena la dirección de la primera componente del vector. Esta característica puede extenderse:

- Puesto que el nombre de un *vector* es un *puntero a constante*, cumplirá las normas de la *aritmética de punteros* (véase subsección 6.2.1 en la pág. 136). Por tanto, si **Vtr** contiene la dirección de **Vtr[0]**, entonces **(Vtr+1)** contendrá la dirección de **Vtr[1]** y así sucesivamente hasta **(Vtr+n)**, que contendrá la de **Vtr[n]**.
- Por otra parte, a los *punteros* se les puede asignar *índices*, al igual que a los *vectores*. De esta manera, si **ptr** a *apunta* a **Vtr[0]**, se admite escribir **ptr[i]**, siendo equivalente a **Vtr[i]**. Esto es, se tienen las igualdades (en sentido aritmético)

  (6.1) **ptr[i] = Vtr[i] = *(ptr + i) = *(Vtr + i)**

  con $i = 0, \ldots, 9$.

Así pues, no sólo tiene sentido la asignación:

ptr = &Vtr[0]

establecida en el código 6.4 anterior, sino también la asignación

(6.2) **ptr = Vtr**

LLegado este punto, es importante hacer la siguiente observación: al ser el *identificador* **Vtr** una *constante*, se cumple la igualdad (en sentido puramente *aritmético*):

&Vtr = Vtr = &Vtr[0]

Se tratan de ***constantes*** que almacenan la misma dirección de memoria, pero aunque **C** les asocia el mismo valor, **`&Vtr`** es *conceptualmente distinto* a **`Vtr`** y **`&Vtr[0]`**. Una representación gráfica de estas relaciones entre puntero y vector puede verse en la figura 6.2.

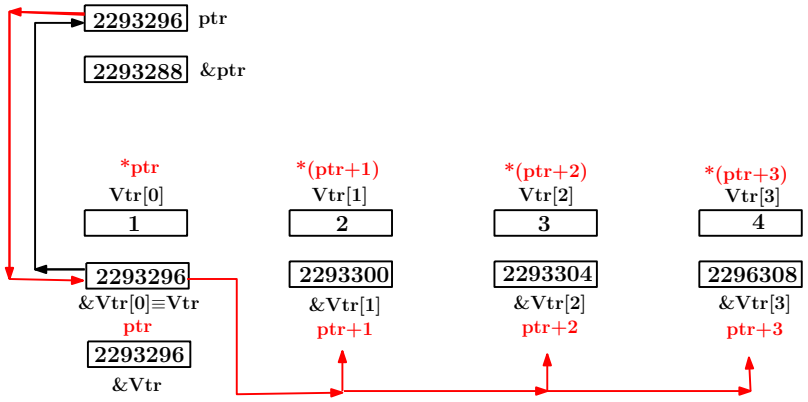


Figura 6.2 – *Representación gráfica de la asociación de un vector con un puntero.*

En este gráfico se muestra que **`&Vtr`** no se puede asignar al puntero **`ptr`** (no sale ninguna *flecha* de la casilla **`&Vtr`** con dirección a la casilla de **`ptr`**), aunque almacena la misma dirección de memoria que **`Vtr`** y **`&Vtr[0]`**. Las direcciones de memoria se ajustan a las obtenidas como resultado del código 6.4 anterior.

En en el siguiente programa se muestran las diferentes equivalencias a la hora de tratar a un *vector* y un *puntero*, obteniéndose el mismo resultado, ya sea expresado a través de un vector o mediante el puntero asociado,

Código 6.5 – *Equivalencias entre vector y puntero*

```
#include <stdio.h>
/* Ejemplo de programa de vectores y punteros */
#define DIM 4
int main(void)
{
    int i;
    int Vtr[DIM] = {28, 29, 30, 31};
    int *ptr;

    FILE *fi;
    fi = fopen("Datos.dat", "w");

    for (i = 0; i < DIM; i++)
        fprintf(fi, "Vtr[%i] = %2i\n", i, Vtr[i] );
    fprintf(fi, "\n");

    for (i = 0; i < DIM; i++)
        fprintf(fi, "*(Vtr + %i) = %2i\n", i, *(Vtr + i));
    fprintf(fi, "\n");

    for (ptr = &Vtr[0], i = 0; i < DIM; i++)
        fprintf(fi, "*(ptr + %i) = %2i \n", i, *(ptr + i) );
    fprintf(fi, "\n");

    for (ptr = &Vtr[0], i = 0; i < DIM; i++)
```

```
26          fprintf(fi, "ptr[%i] = %2i\n", i, ptr[i]);
27
28      fclose(fi);
29
30      return 0;
31  }
```

La salida se almacena en el fichero `Datos.dat` y para todos los procedimientos el resultado es el mismo:

```
Vtr[0] =  28
Vtr[1] =  29
Vtr[2] =  30
Vtr[3] =  31

*(Vtr + 0) =  28
*(Vtr + 1) =  29
*(Vtr + 2) =  30
*(Vtr + 3) =  31

*(ptr + 0) =  28
*(ptr + 1) =  29
*(ptr + 2) =  30
*(ptr + 3) =  31

ptr[0] =  28
ptr[1] =  29
ptr[2] =  30
ptr[3] =  31
```

Es importante destacar que en el **for** de la LÍNEA 25, se ha tenido que inicializar de nuevo la variable **ptr**, ya que su valor fue alterado durante la sentencia **for** de la LÍNEA 21.

6.3.1. Operaciones con punteros

Ya se ha planteado ciertos aspectos de la aritmética de punteros (véase subsección 6.2.1, pág. 136). En este apartado, se pretende clarificar alguna notación *compacta*, utilizada en **C** al emplear punteros asociados a vectores. Veamos la interpretación de alguna de ellas:

- **aux = *ptr++**. Es equivalente a la expresión: **aux = *(ptr++)**. Se interpreta como:
  1. **aux = *ptr**.
  2. **ptr = ptr + 1**.

Código 6.6 – ***ptr++***

```
1   #include <stdio.h>
2
3   #define DIM 3
4   int main(void)
5   {
6         int i, aux;
7         int vector[DIM] = {28, 29, 30};
8         int *ptr;
9
10        FILE *fi;
11        fi = fopen("Datos.dat", "w");
12
```

```
13          ptr = &vector[0];
14          fprintf(fi, " Inicial  %u <-> %2d;\n\n ", ptr, *ptr);
15          for (i = 0; i <= DIM-1; i++){
16              fprintf(fi, "ptr = %u <-> ", ptr);
17              aux = *ptr++;
18              fprintf(fi, "*ptr++ = %2d; ", aux);
19              }
20          fprintf(fi, "\n\n");
21
22          fclose(fi);
23          return 0;
24  }
```

La salida que se obtiene es:

```
Inicial 2293280 <->  28;

ptr = 2293280 <-> *ptr++ =  28; ptr = 2293284 <-> *ptr++ =  29; ptr = 2293288 <-> *ptr++ =  30;
```

- **`aux = *++ptr`**. Es equivalente a la expresión: **`aux = *(++ptr)`**. Se interpreta como:
  1. **`ptr = ptr + 1`**.
  2. **`aux = *ptr`**.

Código 6.7 – ***++ptr***

```
1   #include <stdio.h>
2
3   #define DIM 3
4   int main(void)
5   {
6          int i, aux;
7          int vector[DIM] = {28, 29, 30};
8          int *ptr;
9
10         FILE *fi;
11         fi = fopen("Datos.dat", "w");
12
13         ptr = &vector[0];
14         fprintf(fi, " Inicial  %u <-> %2d;\n\n ", ptr, *ptr);
15         for (i = 0; i <= DIM-1; i++){
16             fprintf(fi, "ptr = %u <-> ", ptr);
17             aux = *++ptr;
18             fprintf(fi, "*++ptr = %2d; ", aux);
19             }
20         fprintf(fi, "\n\n");
21
22         fclose(fi);
23         return 0;
24  }
```

La salida que se obtiene es:

```
Inicial 2293280 <->  28;

ptr = 2293280 <-> *++ptr =  29; ptr = 2293284 <-> *++ptr =  30; ptr = 2293288 <-> *++ptr =  0;
```

En este caso, el programa muestra el contenido de la siguiente dirección de memoria. En el último, como sobrepasa el tamaño del vector, muestra un contenido, posiblemente anterior.

- **aux = (*ptr)++**. Se interpreta como:
  1. **aux = *ptr**.
  2. ***ptr = *ptr + 1**.

Código 6.8 – *(*ptr)++*

```
#include <stdio.h>

#define DIM 3
int main(void)
{
       int i, aux;
       int vector[DIM] = {28, 29, 30};
       int *ptr;

       FILE *fi;
       fi = fopen("Datos.dat", "w");

       ptr = &vector[0];
       fprintf(fi, " Inicial %u <-> %2d;\n\n ", ptr, *ptr);
       for (i = 0; i <= DIM-1; i++){
           fprintf(fi, "ptr = %u <-> ", ptr);
           aux = (*ptr)++;
           fprintf(fi, "(*ptr)++ = %2d; ", aux);
           }
       fprintf(fi, "\n\n");

       fclose(fi);
       return 0;
}
```

La salida que se obtiene es:

```
Inicial 2293280 <->  28;

ptr = 2293280 <-> (*ptr)++ =  28; ptr = 2293280 <-> (*ptr)++ =  29; ptr = 2293280 <-> (*ptr)++
    =  30;
```

En este caso, el programa no cambia el valor de **ptr**, únicamente el valor que contiene. Así pues, queda modificado el valor de **vector[0]**, que resulta ser ahora 30.

- **aux = ++(*ptr)**. Se interpreta como:
  1. ***ptr = *ptr + 1**.

2. **aux = *ptr**.

En esta situación, es interesante aplicar esta notación con la del ejemplo anterior, para observar su resultado. Para ello se propone el siguiente programa:

Código 6.9 – *++(*ptr)*

```
1  #include <stdio.h>
2
3  #define DIM 3
4  int main(void)
5  {
6        int i, aux;
7        int vector[DIM] = {28, 29, 30};
8        int *ptr;
9
10       FILE *fi;
11       fi = fopen("Datos.dat", "w");
12
13       ptr = &vector[0];
14       fprintf(fi, " Inicial %u <-> %2d;\n\n ", ptr, *ptr);
15       for (i = 0; i <= DIM-1; i++){
16           fprintf(fi, "ptr = %u <-> ", ptr);
17           aux = (*ptr)++;
18           fprintf(fi, "(*ptr)++ = %2d; ", aux);
19           }
20       fprintf(fi, "\n\n");
21
22       ptr = &vector[0];
23       fprintf(fi, " Inicial %u <-> %2d;\n\n ", ptr, *ptr);
24       for (i = 0; i <= DIM-1; i++){
25           fprintf(fi, "ptr = %u <-> ", ptr);
26           aux = ++(*ptr);
27           fprintf(fi, "++*ptr = %2d; ", aux);
28           }
29       fprintf(fi, "\n\n");
30
31
32       fclose(fi);
33       return 0;
34 }
```

La salida sería ahora,

```
1  Inicial 2293280 <->  28;
2
3  ptr = 2293280 <-> (*ptr)++ =  28; ptr = 2293280 <-> (*ptr)++ =  29; ptr = 2293280 <-> (*ptr)++
       =  30;
4
5  Inicial 2293280 <->  31;
6
7  ptr = 2293280 <-> ++*ptr =  32; ptr = 2293280 <-> ++*ptr =  33; ptr = 2293280 <-> ++*ptr =  34;
```

De nuevo, no cambia el valor de **ptr**. Sin embargo, en la LÍNEA 5 de la salida muestra un valor para ***ptr** de 31. Ello es debido a posición *sufija* del operador *incremento* en la LÍNEA 17 del código 6.9, que

una vez que imprime el valor de 30, se incrementa en una unidad, con lo cual en la LÍNEA 22 del mismo programa, el valor de **`vector[0]`** y por tanto, el de **`*ptr`** es 31.

6.4. Matrices y punteros

El concepto de matriz ya se introdujo en la sección 5.4 (pág. 113), ahora se trata de relacionar la estructura de matriz con la idea de puntero. Para entender claramente este procedimiento es importante conocer el mecanismo que utiliza **C** para operar con las matrices.

Lo mismo que sucede con los *vectores*, existen dos formas para acceder a los elementos de las matrices: mediante índices o empleando punteros. En cuanto a la primera, como ya se estudió, la forma de designar un elemento es mediante la expresión `a[i][j]`: el primer índice indica la *fila* y el segundo la *columna*. Realmente para el **C**, una matriz es un ***vector*** de una única fila de elementos contiguos (cuyo número coincide con el número de *filas*), en la que cada uno de estos elementos es a su vez, un vector de tamaño igual al número de *columnas*. Así pues, al hacer variar el índice de la derecha (*columnas*), el acceso a los valores de la matriz es mucho más rápido, que si se realiza variando primeramente el índice de la izquierda (*filas*). Por esta razón, se dice que en **C**, los elementos de una matriz se ***almacenan por filas***. Para localizar sobre este ***vector***, un elemento $a_{i,j}$ de una matriz $\mathbf{A} \in \mathcal{M}_{n\times m}$, se utiliza la transformación

$$a_{i,j} \iff P(i,j) \quad \text{donde } P(i,j) = i*m+j \quad \text{con } 0 \leqslant i \leqslant n;\ 0 \leqslant j \leqslant m$$

Como consecuencia, mientras que es crucial conocer el *número total de columnas* de la matriz, ya que junto con los índices, permite la localización del elemento sobre este ***vector***, saber el *número total de filas*, resulta accesorio. Ello conduce a definir un *puntero* a una matriz de tamaño igual al número de columnas. Veamos esta situación con un simple ejemplo. Se trata de asociar el vector de enteros

$$\mathbf{V} = [1,2,3,4]$$

a la matriz **`PtrV`** $\in \mathcal{M}_{1\times 4}$. Esta matriz debe tener por tanto, 4 columnas. Se define entonces el *puntero a matriz de* 4 *enteros*:

```
int (*PtrV)[4];
```

que va a configurar la matriz *fila*. Esta expresión, también puede interpretarse como un *puntero a un entero de tamaño* 4. En efecto, **`V`** es un *vector* de cuatro enteros, con lo cual se puede identificar al vector **`V`** como una única variable *entera que tiene a su vez, un tamaño de* 4 *enteros*. Realmente es así como **C** configura las matrices. Consecuentemente, tiene sentido la asignación:

```
PtrV = &V;
```

Obsérvese que dicha asignación **no** tendría sentido si **`PtrV`** se hubiese declarado como un *puntero a entero*:

```
int *PtrV;
```

tal y como se ha comentado en la sección 6.3 (pág. 138). Así pues, ahora **no** tienen sentido asignaciones del tipo **`PtrV = &V[0]`** o equivalentemente **`PtrV = V`**; de hecho da error de compilación. Como consecuencia de este proceso de asignación, se deducen las siguientes igualdades lógicas (en el sentido aritmético):

`*PtrV = PtrV[0] = V = &V[0]`

por tanto: **PtrV[0] + 1** es equivalente a **V + 1** y así sucesivamente. O bien, utilizando la notación de índices se tendría la equivalencia entre **PtrV [0][1]** y **V[1]** y así sucesivamente con todas las componentes del vector. El código del programa que ejecuta la asociación del vector **V** con la matriz de una *fila* y 4 columnas es el siguiente:

Código 6.10 – *Vector y puntero a matriz de enteros*

```
#include <stdio.h>
/* Ejemplo de programa de vector a puntero */
#define COLUM 4
int main(void)
{
       int i, j;
       int V[COLUM] ={1, 2, 3, 4};
       int (*PtrV)[COLUM];


       FILE *fi;
       fi = fopen("Datos.dat", "w");

       fprintf(fi, "&V: %u; V: %u\n\n", &V, V);
       for (i = 0; i < COLUM; i++)
           fprintf(fi, " V[%1d] = %7d; ", i, V[i]);
       fprintf(fi, "\n");
       for (i = 0; i < COLUM; i++)
           fprintf(fi, "&V[%1d] = %u; ", i, &V[i] );
       fprintf(fi,"\n\n");

       PtrV = &V;
       fprintf(fi,"Asignación de dirección de memoria: PtrV = &V\n\n");
       fprintf(fi, "&PtrV: %u; PtrV: %u; PtrV[0]: %u;\n", &PtrV, PtrV, PtrV[0]);
       for (i = 0; i < COLUM; i++)
           fprintf(fi, " PtrV[0][%1d] = %8d; ", i, PtrV[0][i] );
       fprintf(fi,"\n");
       for (i = 0; i < COLUM; i++)
            fprintf(fi, " *(PtrV + %1d) = %9u; ", i, *(PtrV  + i) );
       fprintf(fi, "\n");
       for (i = 0; i < COLUM; i++)
            fprintf(fi, " PtrV[0] + %1d = %u; ", i, PtrV[0]  + i );
       fprintf(fi, "\n");


     fclose(fi);

     return 0;
}
```

La salida se almacena el en fichero `Datos.dat` y el resultado es:

```
&V: 2293296; V: 2293296

 V[0] =       1;  V[1] =       2;  V[2] =       3;  V[3] =       4;
&V[0] = 2293296; &V[1] = 2293300; &V[2] = 2293304; &V[3] = 2293308;

Asignación de dirección de memoria: PtrV = &V
```

```
&PtrV: 2293288; PtrV: 2293296; PtrV[0]: 2293296;
 PtrV[0][0] =        1;  PtrV[0][1] =        2;  PtrV[0][2] =        3;  PtrV[0][3] =        4;
 *PtrV + 0 =   2293296;  *PtrV + 1 =   2293300;  *PtrV + 2 =   2293304;  *PtrV + 3 =   2293308;
 PtrV[0] + 0 = 2293296;  PtrV[0] + 1 = 2293300;  PtrV[0] + 2 = 2293304;  PtrV[0] + 3 = 2293308;
```

La interpretación gráfica del proceso puede verse en la figura 6.3. Es interesante comparar esta figura con la 6.2 en la pág. 140. En este caso, debido a la asignación planteada (LÍNEA 22), la *flecha*, en la figura 6.3, sale de la casilla **`&V`**.

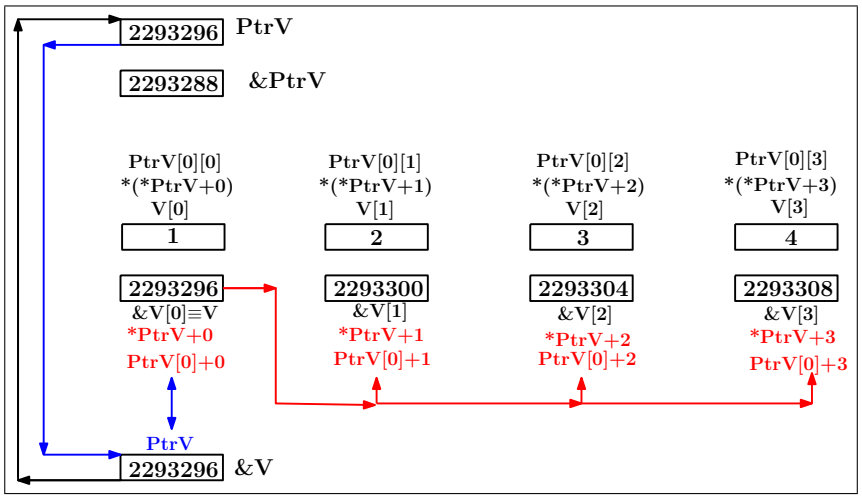


Figura 6.3 – *Asociación de un vector a un* puntero a vector

El **lenguaje C** permite una declaración alternativa a la que se ha utilizado para el puntero **`PtrV`**. Se puede también declarar de la forma

```
int PtrV[][4];
```

Tal vez esta sea más intuitiva, en el sentido de al estar *vacía* la primera componente (*número de filas*), comunique un cierto efecto de *variabilidad*; mientras que la segunda (*número de columnas*), que es importante tal y como se ha comentado anteriormente, se deja fija.

El siguiente paso, consiste en asociar una matriz dada, a un puntero. Se ha explicado anteriormente, que resulta fundamental conocer el *número total de columnas* de la matriz. Siguiendo con el planteamiento del ejemplo anterior, se considera ahora la matriz *entera*

$$\mathbf{Mat} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{bmatrix}$$

La idea es asociarla a un *puntero*. Para ello se define, de manera análoga al caso anterior, el puntero:

```
int (*PtrMat)[4]; /* o bien: int PtrMat[][4]; */
```

Este puntero señalará el comienzo de cada fila de la matriz. Así pues, en este caso tiene sentido la asignación

```
PrtMat = &Mat[0];
```

No debe olvidarse, que **C** entiende que la matrices son vectores, cuyas componentes son a su vez variables pero de tamaño coincidente con el número de columnas. De esta manera, **`Mat[0]`** es el nombre de la variable que almacena los valores de la primera fila. Por tanto, **`Mat[0]`** representa a una única variable *entera que contiene a su vez,* 4 *enteros*, al igual que sucedía con el vector **`V`** del ejemplo anterior y queda justificada la asignación anterior. Así pues, si

`PtrMat = &Mat[0]`

entonces como consecuencia, ocurre la igualdad (aritmética)

`*PtrMat = Mat[0]`

Teniendo en cuenta las identificaciones entre vectores y punteros planteadas anteriormente (véase sección 6.3, pág. 138), entonces se dan las siguientes igualdades lógicas (en el sentido aritmético):

`*PtrMat = PtrMat[0] = Mat[0] = &Mat[0][0]`

`*(PtrMat+1) = PtrMat[1] = Mat[1] = &Mat[1][0]`

`*(PtrMat+2) = PtrMat[2] = Mat[2] = &Mat[2][0]`

y

`*(PtrMat+3) = PtrMat[3] = Mat[3] = &Mat[3][0]`

A la vista de este proceso, se tendrán las siguientes equivalencias

(6.3) **`Mat[i][j]`** ≡ **`*( PtrMat[i] + j )`** ≡ **`*( *(PtrMat+i) + j )`** ≡ **`PtrMat[i][j]`**

con la diferencia que **`Mat`** es una *constante*, mientras que **`PrtMat`** es una *variable*. El resultado del programa 6.11 permite observar estas asignaciones con direcciones de memorias concretas:

Código 6.11 – *Matriz y puntero a matriz de enteros*

```
#include <stdio.h>
/* Ejemplo de programa de matrices */
#define FILA 4
#define COLUM FILA
int main(void)
{
    int i, j;
    int Mat[FILA][COLUM] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13,
       14, 15, 16}};
    int (*PtrMat)[COLUM];


    FILE *fi;
    fi = fopen("Datos.dat", "w");

    fprintf(fi,"Mat: %u; Mat[0]: %u; ", Mat, Mat[0]);
    fprintf(fi," &Mat[0]: %u; &Mat[0][0]: %u\n\n", &Mat[0], &Mat[0][0]);

    fprintf(fi, "&PtrMat: %u; PtrMat: %u\n\n", &PtrMat, PtrMat);
```

```
19      fprintf(fi,"Se establece la asignación: PtrMat = Mat ó PtrMat = &Mat[0]\n\n"
            );
20
21      PtrMat = Mat; /*PtrMat = &Mat[0]*/
22
23      fprintf(fi, "&PtrMat: %u; PtrMat: %u\n\n", &PtrMat, PtrMat);
24      for(i = 0; i <  FILA; i++){
25          fprintf(fi, "PtrMat + %i = %i\n", i, PtrMat + i);
26          for ( j = 0; j <  COLUM; j++)
27              fprintf(fi, "PtrMat[%1d][%1d]=%8d; ", i, j, PtrMat[i][j] );
28          fprintf(fi, "\n");
29          for ( j = 0; j < COLUM; j++)
30                  fprintf(fi, "PtrMat[%1d]+%1d = %u; ", i, j, PtrMat[i] + j );
31          fprintf(fi, "\n\n");
32      }
33      fprintf(fi,"\n\n");
34
35      fclose(fi);
36      return 0;
37  }
```

La salida que se obtiene es

```
Mat: 2293248; Mat[0]: 2293248;  &Mat[0]: 2293248; &Mat[0][0]: 2293248

&PtrMat: 2293240; PtrMat: 121

Se establece la asignación: PtrMat = Mat ó PtrMat = &Mat[0]

&PtrMat: 2293240; PtrMat: 2293248

PtrMat + 0 = 2293248
PtrMat[0][0]=       1; PtrMat[0][1]=       2; PtrMat[0][2]=       3; PtrMat[0][3]=       4;
PtrMat[0]+0 = 2293248; PtrMat[0]+1 = 2293252; PtrMat[0]+2 = 2293256; PtrMat[0]+3 = 2293260;

PtrMat + 1 = 2293264
PtrMat[1][0]=       5; PtrMat[1][1]=       6; PtrMat[1][2]=       7; PtrMat[1][3]=       8;
PtrMat[1]+0 = 2293264; PtrMat[1]+1 = 2293268; PtrMat[1]+2 = 2293272; PtrMat[1]+3 = 2293276;

PtrMat + 2 = 2293280
PtrMat[2][0]=       9; PtrMat[2][1]=      10; PtrMat[2][2]=      11; PtrMat[2][3]=      12;
PtrMat[2]+0 = 2293280; PtrMat[2]+1 = 2293284; PtrMat[2]+2 = 2293288; PtrMat[2]+3 = 2293292;

PtrMat + 3 = 2293296
PtrMat[3][0]=      13; PtrMat[3][1]=      14; PtrMat[3][2]=      15; PtrMat[3][3]=      16;
PtrMat[3]+0 = 2293296; PtrMat[3]+1 = 2293300; PtrMat[3]+2 = 2293304; PtrMat[3]+3 = 2293308;
```

Una interpretación gráfica de este planteamiento y con a las direcciones resultantes obtenidas a partir del programa 6.11, puede verse en la figura 6.4.

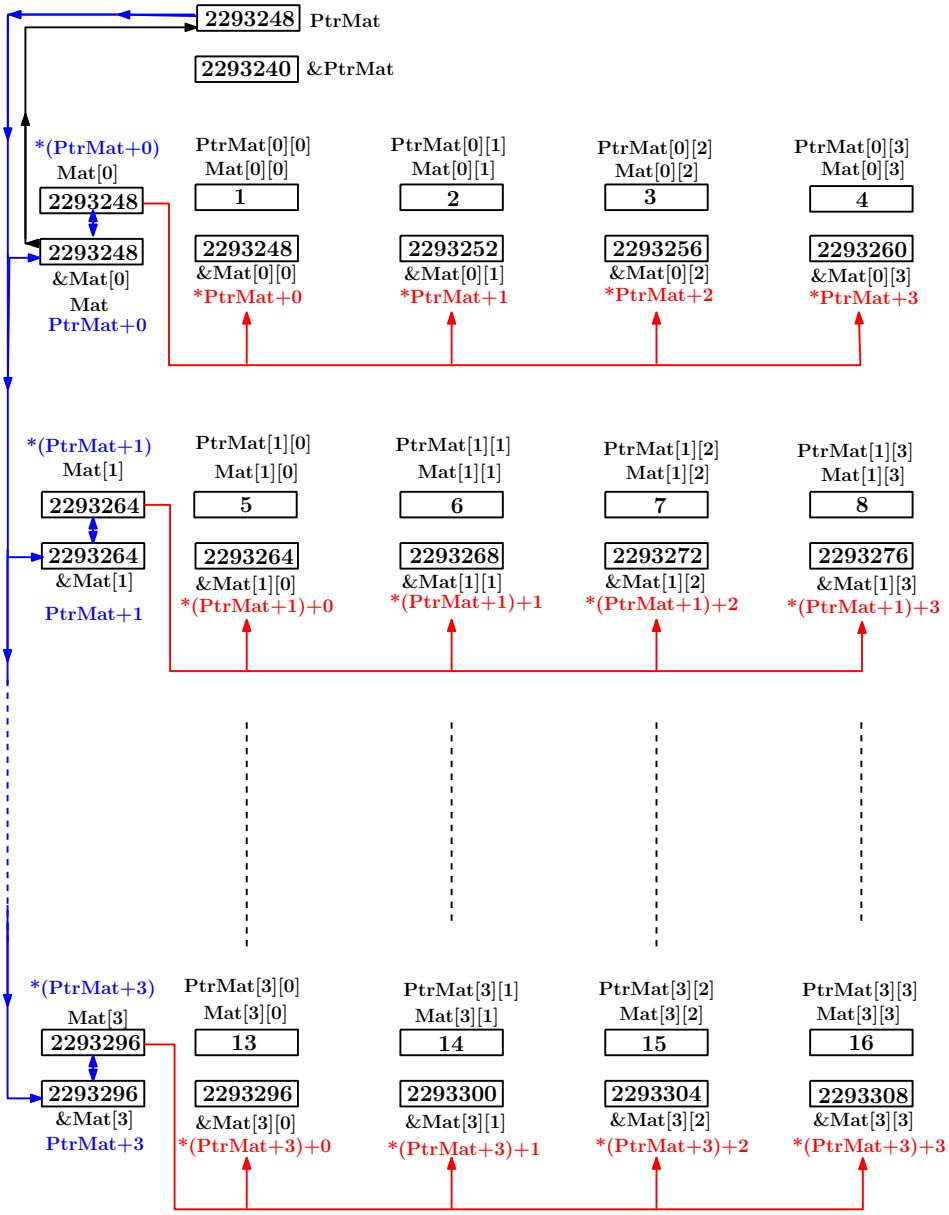


Figura 6.4 – *Asociación de una matriz a un* puntero matricial

Es inmediato, que el caso anterior se puede generalizar a matrices de cualquier tamaño, cambiando el número de columnas y de cualquier *tipo*, sustituyendo **`int`** por **`char, float, double,`** ...y de forma respectiva al *puntero* en cuestión.

6.5. Estructuras y punteros

Al igual que ocurre con las variables, también existe la posibilidad de asociar un puntero a una *estructura*. La declaración de *puntero a estructura* es como la de *puntero* a cualquier otra variable. Se declara el objeto apuntado (véase la sección 5.5, pág. 116):

```
    struct alumno *Ptr1, *Ptr2;
```

Se puede acceder a los miembros de una estructura a través de un puntero de dos maneras indistintamente:

1. ***Accediendo a la estructura***:

```
(*Ptr1).N_mat;
```

   Los paréntesis son necesarios porque el operador `.` tiene una mayor precedencia que la *dirección* `*`.

2. ***Directamente desde el puntero***. Se utiliza el *operador:* `->`.

```
Ptr1 -> N_mat;
```

   Es un procedimiento más intuitivo que el anterior y por esta razón es más utilizado.

Como ejemplo, se presenta el programa 6.12 que es una modificación del código 5.14, para emplear un *puntero a estructura*.

Código 6.12 – *Estructuras*

```
#include <stdio.h>
/* Ejemplo de programa con estructuras */
main()
{
      struct alumno{
             int N_mat;
             char Apellidos[35];
             char Nombre[15];
             };
      struct alumno estudiante, *Ptr;

      FILE *fi;
      fi = fopen("Datos.dat", "w");

      Ptr = &estudiante;
      printf("Introduce el Número de matrícula -> ");
      scanf("%d", &Ptr -> N_mat);
      while (getchar() != '\n'); /* Para vaciar el buffer */
      printf("\n\n");
      printf("Introduce los Apellidos -> ");
      gets(Ptr -> Apellidos);
      printf("\n\n");
      printf("Introduce el Nombre -> ");
      gets(Ptr ->  Nombre);

      fprintf(fi, " Los datos introducidos son: \n\n");
      fprintf(fi, "* Número de matrícula: %d\n", Ptr -> N_mat);
      fprintf(fi, "* Apellidos: %s \n", Ptr -> Apellidos);
      fprintf(fi, "* Nombre: %s \n", Ptr -> Nombre);

      fclose(fi);
}
```

La salida es análoga a la obtenida para el código 5.14, dependiendo de los datos introducidos:

```
 Los datos introducidos son:

* Número de matrícula: 23456
* Apellidos: Torralba Campos
* Nombre: Miriam
```

6.6. Dimensionamiento dinámico y punteros

En las secciones anteriores, la relación establecida entre vectores, matrices y estructuras con los punteros se ha realizado a partir de un dimensionamiento estático de estos *datos estructurados*. No obstante, el hecho de utilizar un procedimiento de ***dimensionamiento dinámico*** (véase sección 5.7, pág. 125) para definir vectores, matrices o estructuras, no afecta al comportamiento del *puntero* asociado a dicho dato. Por tanto, todo lo explicado en estas secciones es aplicable tanto para *datos estructurados* dimensionados *estáticamente*, como para aquellos dimensionados dinámicamente.

6.7. Gestión dinámica

El concepto de ***gestión dinámica*** es más general que el de ***dimensionamiento dinámico*** ya comentado (véase sección 5.7, pág. 125). Con la *gestión dinámica* no sólo se puede dimensionar un vector, una matriz o una estructura dinámicamente a través del *direccionamiento* o reserva de la memoria, sino también cabe la posibilidad de gestionar su tamaño (aumentarlo o reducirlo) durante la ejecución del programa, proceso que resultaba imposible si los datos estructurados hubieran estado dimensionados estáticamente o dinámicamente, previamente (véase, de nuevo sección 5.7, pág. 125).

El **lenguaje C** ofrece una serie de funciones para poder *direccionar* y gestionar la memoria durante la ejecución de un programa: **`malloc()`**, **`calloc()`**, **`realloc()`** y **`free()`**, cuyas definiciones y declaraciones se encuentran en la librería **`stdlib.h`**.

6.7.1. `malloc()`

Es la manera más habitual de reservar o *direccionar* bloques de memoria de manera dinámica. La función genera o asigna un bloque de memoria en *bytes*, cuyo tamaño es el argumento de la función. La función devuelve un puntero **`*void`** al bloque de memoria asignado, por tanto hay que hacer una conversión al *tipo* del puntero requerido. El prototipo de la función es de la forma

```
    void* = malloc(unsigned int tamaño_bytes);
```

Un ejemplo para reservar dinámicamente un bloque para un *vector* de 10 enteros, sería

```
    int *PtrInt
    PtrInt = (int *)malloc( 10 * sizeof(int) );
```

La función **`sizeof`**, que ya fue utilizada anteriormente (véase sección 3.2 en la pág. 25 o bien en el código 6.2, pág. 135), indica el tamaño en *bytes*, del *tipo* **`int`**.

En caso que haya problemas en direccionar los bloques de memoria, **`malloc()`** devuelve el valor `NULL`. Es por tanto, conveniente preguntar después de una operación de asignación de memoria, si todo está correcto.

```
    int *PtrInt
    PtrInt = (int *)malloc( 10 * sizeof(int) );
```

```
    if( PtrInt == NULL ){
        printf("No hay memoria disponible\n");
        exit(1);
        }
```

La función **exit** es una función proporcionada por **C** para manejar posibles errores. No devuelve ningún valor, pero necesita un argumento numérico. La opción **exit(0)** indica una salida normal. Sin embargo, cualquier otro valor distinto de 0, señala una situación anormal. La ventaja de la utilización de esta función es que antes de *salir* del programa, cierra todos los ficheros y vacía el *buffer*. Con lo cual se obtiene una *salida ordenada* del programa. El programa 6.13 muestra un ejemplo de direccionamiento dinámico de un vector, cuyo objetivo consiste en determinar el valor máximo de las componentes del vector

Código 6.13 – *Valor máximo de la componente de un vector (función `malloc()`)*

```
#include <stdio.h>
#include <stdlib.h>
/* Ejemplo de programa con malloc() */
#define MAX(X,Y) ((X) > (Y) ? (X) : (Y) )
int main(void)
{
    int i;
    int Dim;
    float *Vect;
    float MaxVect;


    FILE *fi;
    fi = fopen("Datos.dat", "w");

    printf("Introduce la dimensión del vector -> ");
    scanf("%d", &Dim);
    while (getchar() != '\n');

    Vect = (float *) malloc( Dim * sizeof(float));
    if (Vect == NULL ) {
        printf("Error en la asignación de memoria\n");
        getchar();
        exit(1);
    }

    printf("\n\n Introduce las componentes del vector: \n");
    for(i = 0; i < Dim; i++){
        printf("V[%d] = ", i+1);
        scanf("%f", Vect + i); // o bien scanf("%f", &Vect[i]);
        while(getchar() != '\n');
    }

    fprintf(fi, "El vector de dimensión %2d introducido es: \n\n", Dim);
    for (i = 0; i < Dim; i++)
        fprintf(fi,"Vec[%1d] = %.4f ; ", i, *(Vect +i));
    fprintf(fi, "\n\n");

    for(MaxVect = Vect[0], i=1; i < Dim; i++)
        MaxVect = MAX(MaxVect, Vect[i]);
```

```
41
42      fprintf(fi, "El valor máximo es: %.4f\n", MaxVect);
43
44      fclose(fi);
45
46      return 0;
47  }
```

El resultado para los datos introducidos es:

```
El vector de dimensión  4 introducido es:

Vec[0] = -12.3400 ; Vec[1] = 8.5200 ; Vec[2] = -0.4500 ; Vec[3] = 6.9300 ;

El valor máximo es: 8.5200
```

Al igual que sucede con los *vectores*, las matrices también pueden *direccionarse dinámicamente*. Hasta ahora, las matrices se asociaban con *punteros a variables* de un determinado *tipo*, cuyo tamaño coincidía con el número de columnas de la matriz (véase LÍNEA 8 del código 6.10 (pág. 146) o LÍNEA 9 del código 6.11 (pág. 148)). Sin embargo, a través de la función **`malloc()`** una disposición de datos matricial se puede asociar a un *doble puntero*, mediante dos pasos:

1. Se reserva espacio en la memoria para un *vector* de *punteros* que señalarán las direcciones de los *vectores fila*, mediante la instrucción:

   `MatReal = (double **) malloc( Filas * sizeof(double *));`

   donde **`MatReal`** es un *doble puntero a double*: **`double **MatReal`**. Un ejemplo completo se presenta en el programa 6.14.

Código 6.14 – *Direccionamiento de una matriz con* `malloc`

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   /* Ejemplo de programa con malloc() */
4   int main(void)
5   {
6       int i,j;
7       int Filas, Columnas;
8       double **MatReal;
9
10
11      FILE *fi;
12      fi = fopen("Datos.dat", "w");
13
14      printf("Introduce las Filas y Columnas -> ");
15      scanf("%d %d", &Filas, &Columnas);
16      while (getchar() != '\n');
17
18      fprintf(fi, "Filas: %3d, Columnas: %3d\n\n", Filas, Columnas);
19      fprintf(fi,"sizeof(double)=%d; sizeof(double *)=%d; sizeof(double **)=%d
            \n", sizeof(double), sizeof(double*), sizeof(double **));
20      fprintf(fi, "\n\n");
21
22      // *** Direcciona las filas ***
```

```
23      MatReal = (double **) malloc( Filas  * sizeof(double *));
24
25      if (MatReal == NULL ) {
26          printf("Error en la asignación de memoria\n");
27          getchar();
28          exit(1);
29      }
30      fprintf(fi, "&MatReal: %u; MatReal: %u; \n\n", &MatReal, MatReal);
31      fprintf(fi, "MatReal: %u; Direcciona: %d bytes, ", MatReal, Filas*sizeof
            (double *));
32      fprintf(fi, "es decir %d punteros a (double *)\n\n", Filas);
33      fprintf(fi, "\n\n");
34      fprintf(fi, "Dirección de las filas:\n");
35      for(i = 0; i < Filas; i++){
36          fprintf(fi, "&MatReal[%1d]: %u; ", i, &MatReal[i]);
37          fprintf(fi, "MatReal+%1d: %u; ", i, MatReal+i);
38          fprintf(fi, "MatReal[%1d]: %u\n", i, MatReal[i]);
39      }
40
41      fclose(fi);
42
43      return 0;
44  }
```

El resultado de este programa se almacena en el fichero `Datos.dat`:

```
Filas:   3, Columnas:   3

sizeof(double) = 8; sizeof(double *): 8; sizeof(double **): 8


&MatReal: 2293296; MatReal: 3014400;

MatReal: 3014400; Direcciona: 24 bytes, es decir 3 punteros a (double *)


Dirección de las filas:
&MatReal[0]: 3014400; MatReal+0: 3014400; MatReal[0]: 8084048
&MatReal[1]: 3014408; MatReal+1: 3014408; MatReal[1]: 8061272
&MatReal[2]: 3014416; MatReal+2: 3014416; MatReal[2]: 0
```

En la salida se observa que las variables **`MatReal[0]`**, **`MatReal[1]`** y **`MatReal[2]`**, tienen una serie de valores almacenados. Estos valores no han sido asignados por el programa sino que que son datos que tenía previamente almacenados en esa posición de memoria. La figura 6.5 muestra una representación gráfica del proceso de asignación realizada por el programa 6.14.

2. En el *paso segundo*, se reserva espacio en la memoria para almacenar los vectores a los que apuntan las direcciones anteriormente establecidas. Es decir se reserva memoria para las *columnas*. La instrucción para direccionar las columnas de la fila i es

   ```
   MatReal[i] = (double *) malloc( Columnas * sizeof(double) );
   ```

   El proceso completo se describe en el programa 6.15.

Código 6.15 – *Direccionamiento de una matriz con* `malloc`

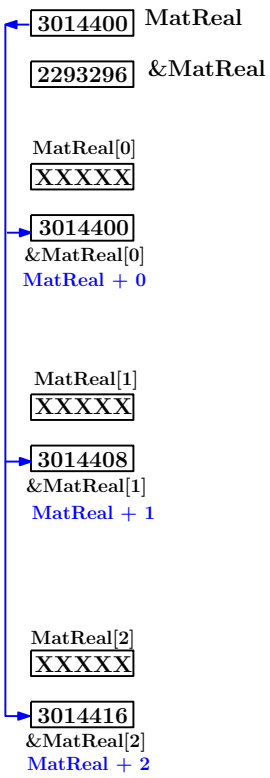


Figura 6.5 – *Utilización de la función `malloc()` para direccionar un doble puntero*

```c
#include <stdio.h>
#include <stdlib.h>
/* Ejemplo de programa con malloc() */
int main(void)
{
    int i,j;
    int Filas, Columnas;
    double **MatReal;

    FILE *fi;
    fi = fopen("Datos.dat", "w");

    printf("Introduce las Filas y Columnas -> ");
    scanf("%d %d", &Filas, &Columnas);
    while (getchar() != '\n');

    fprintf(fi, "Filas: %3d, Columnas: %3d\n\n", Filas, Columnas);
    fprintf(fi," sizeof(double) = %d; sizeof(double *): %d; sizeof(double
        **): %d\n", sizeof(double), sizeof(double*), sizeof(double **));
    fprintf(fi, "\n\n");

    // *** Direccionamiento de las filas ****
    MatReal = (double **) malloc( Filas  * sizeof(double *));
    if (MatReal == NULL ) {
        printf("Error en la asignación de memoria\n");
        getchar();
```

```
26          exit(1);
27      }
28
29      fprintf(fi, "&MatReal: %u; MatReal: %u; \n\n", &MatReal, MatReal);
30      fprintf(fi, "MatReal: %u; Direcciona: %d bytes, ", MatReal, Filas*sizeof
            (double *));
31      fprintf(fi, "es decir %d punteros a (double *)\n\n", Filas);
32      fprintf(fi, "\n\n");
33      fprintf(fi, "Dirección de las filas:\n");
34      for(i=0; i<Filas; i++){
35          fprintf(fi, "&MatReal[%1d]: %u; ", i, &MatReal[i]);
36          fprintf(fi, "MatReal+%1d: %u; ", i, MatReal+i);
37          fprintf(fi, "MatReal[%1d]: %u\n", i, MatReal[i]);
38      }
39      fprintf(fi, "\n\n");
40
41      for(i=0; i<Filas; i++)
42          fprintf(fi, "MatReal + %1d: %u; *(MatReal + %1d) = %u;\n", i,
                MatReal+i, i, *(MatReal +i));
43      fprintf(fi,"\n\n");
44
45      //*** Direccionamiento de las columnas ***
46      fprintf(fi, "Asignación para direccionar las columnas\n\n");
47      for(i=0; i<=Filas; i++){
48          MatReal[i] = (double *) malloc( Columnas * sizeof(double) );
49          if (MatReal[i] == NULL ) {
50              printf("Error en la asignación de memoria\n");
51              getchar();
52              exit(1);
53          }
54      }
55      for(i=0; i<Filas; i++){
56          fprintf(fi, "** &MatReal[%d]: %u;", i, &MatReal[i]);
57          fprintf(fi," MatReal[%1d] = %u;\n ", i, MatReal[i]);
58          for(j = 0; j<Columnas; j++){
59              fprintf(fi, "MatReal[%1d] + %1d = %u;", i,j,MatReal[i]+j);
60              fprintf(fi, " MatReal[%1d][%1d] = %d ", i,j,MatReal[i][j]);
61              fprintf(fi, "\n\n");
62          }
63      }
64      fprintf(fi, "\n\n");
65
66      fclose(fi);
67
68      return 0;
69  }
```

La salida obtenida es:

```
Filas:   3, Columnas:   4

 sizeof(double) = 8; sizeof(double *): 8; sizeof(double **): 8


&MatReal: 2293280; MatReal: 6553344;
```

```
MatReal: 6553344; Direcciona: 24 bytes, es decir 3 punteros a (double *)

&MatReal[0]: 6553344; MatReal+0: 6553344; MatReal[0]: 3693136
&MatReal[1]: 6553352; MatReal+1: 6553352; MatReal[1]: 3670360
&MatReal[2]: 6553360; MatReal+2: 6553360; MatReal[2]: 0


MatReal + 0: 6553344; *(MatReal + 0) = 3693136;
MatReal + 1: 6553352; *(MatReal + 1) = 3670360;
MatReal + 2: 6553360; *(MatReal + 2) = 0;


Asignación para direccionar las columnas

** &MatReal[0]: 6553344; MatReal[0] = 6553376;
 MatReal[0] + 0 = 6553376; MatReal[0][0] = 3693136

MatReal[0] + 1 = 6553384; MatReal[0][1] = 3670360

MatReal[0] + 2 = 6553392; MatReal[0][2] = 0

MatReal[0] + 3 = 6553400; MatReal[0][3] = 0

** &MatReal[1]: 6553352; MatReal[1] = 6553424;
 MatReal[1] + 0 = 6553424; MatReal[1][0] = 3693136

MatReal[1] + 1 = 6553432; MatReal[1][1] = 3670360

MatReal[1] + 2 = 6553440; MatReal[1][2] = 0

MatReal[1] + 3 = 6553448; MatReal[1][3] = 0

** &MatReal[2]: 6553360; MatReal[2] = 6553472;
 MatReal[2] + 0 = 6553472; MatReal[2][0] = 3693136

MatReal[2] + 1 = 6553480; MatReal[2][1] = 3670360

MatReal[2] + 2 = 6553488; MatReal[2][2] = 0

MatReal[2] + 3 = 6553496; MatReal[2][3] = 0
```

La representación gráfica de la asignación completa se observa en la figura 6.6

Es importante destacar que para **C**, el *doble puntero* (en este caso **`MatReal`**), no se asocia a la idea de matriz tal y como se planteó en la sección 5.4 (pág. 113). Para el **lenguaje C**, el único puntero a una matriz es un *puntero a variable* de un determinado *tipo*, cuyo tamaño coincide con el número de columnas de la matriz, tal y como se ha planteado en la sección 6.4 (pág. 145). Sin embargo, el *doble puntero* definido a través de la función **`malloc()`**, permite asociar un puntero, a una disposición de datos de doble entrada y que de esta forma, su manejo resulta muy cómodo.

6.7.2. free()

Cuando se termina de utilizar un bloque de memoria, direccionado previamente por cualquier función de asignación dinámica, se debe liberar el espacio de memoria para dejarlo disponible para otros procesos. Este procedimiento lo realiza la función **`free()`**. El prototipo que define dicha función es

```
void free(void *ptr );
```

donde **`*ptr`** es el puntero que hace referencia al bloque se hay que liberar. Si **`ptr`** es `NULL`, entonces no hace nada.

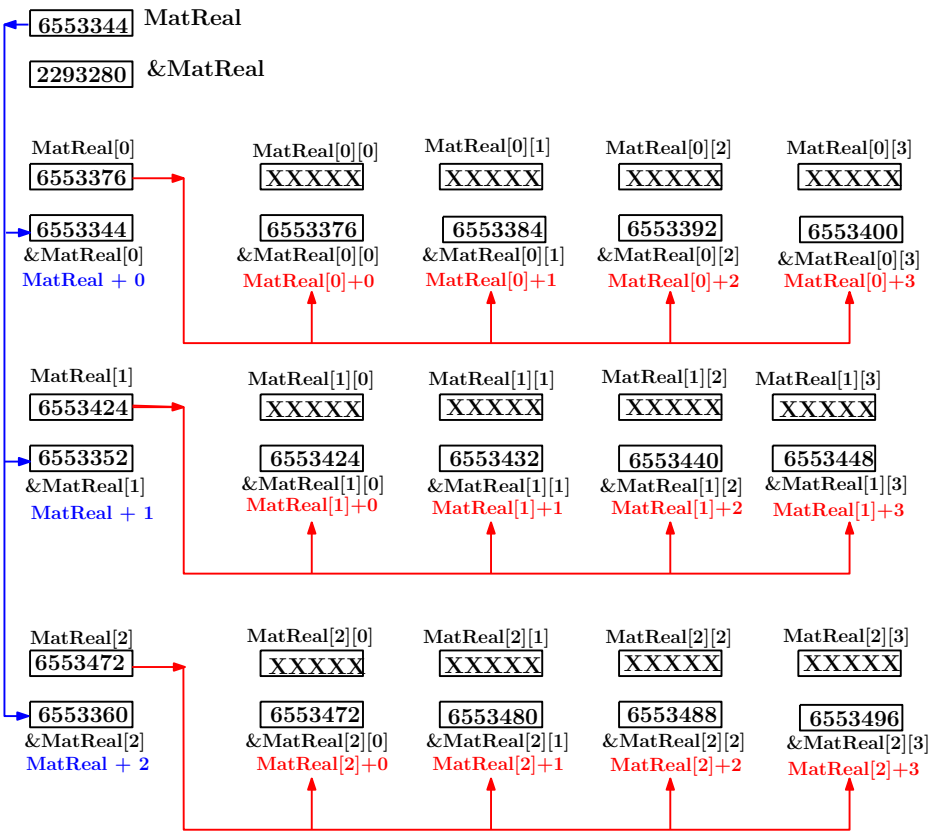


Figura 6.6 – *Utilización de la función `malloc()` para direccionar una matriz*

En el programa anterior, para liberar la memoria asignada a la matriz **`MatReal`**, se haría también en dos pasos:

1. Se liberaría la memoria asignada a los vectores que direccionan las columnas:

```
    for (i=0; i<Filas; i++)
        free(MatReal[i] );
```

2. Por último se libera la memoria del vector de punteros:

```
    free(MatReal);
```

6.7.3. calloc()

Se trata de otra función que permite direccionar memoria dinámicamente. Es muy similar a **`malloc()`**, devuelve un puntero **`void*`** que hace referencia al bloque de memoria direccionado o `NULL` si no existe memoria suficiente para asignar. La forma de utilización es

```
puntero = (tipo*) calloc( número_elementos, tamaño tipo );
```

Por ejemplo,

```
int *PtrInt;
PtrInt = (int *) calloc(25, sizeof(int));
```

genera una dirección, que almacena en **PtrInt**, para 25 variables del *tipo entero*. Tiene unas características similares a la función **malloc()**.

6.7.4. realloc()

Se trata de una función más técnica que las anteriores. Permite cambiar el tamaño de un bloque de memoria asignado previamente. Es una función que devuelve un puntero **void***. Su forma de utilización es:

```
puntero = (tipo*) realloc( ptr_anterior, nuevo tamaño );
```

Tiene también unas características similares a la función **malloc()**.

6.8. Problemas

1. Basándote en el código 6.1 (pág. 134), realiza un programa que declare e inicialice una variable de *tipo* **double**. A continuación declara un *puntero a double*, que apunte a la variable anterior. A través del puntero, suma y resta una cantidad constante. En un fichero, escribe el valor inicial de la variable con su dirección y el resultado final con su dirección.

2. Dado el vector real $\mathbf{V} = [-2.45, 0.23, -45.98, -23.25, 1.89]$, se pide que realices un programa que asigne un puntero al vector y mediante este puntero, opera adecuadamente para incrementar cada componente en 25 unidades. En un fichero de salida se debe escribir el resultado final, con la dirección de memoria de cada componente. Primero hazlo con un *tipo* **float** y después con un *tipo* **double**. Observa el valor de las direcciones de memoria.

3. ***Ejemplo de* *++ptr**. Basándote en el código 6.7 (pág. 142), se pide que asignes el puntero **ptr** a la cadena de caracteres: **Me gusta programar en C** y a continuación imprime en un fichero, la salida como consecuencia de operar: ***++ptr**.

4. ***Ejemplo de* (*ptr)--**. Basándote en el código 6.8 (pág. 143), se pide que asignes el puntero **ptr** a la cadena de caracteres: **Me gusta programar en C** y a continuación imprime en un fichero, la frase escrita del revés.

5. Dada la matriz real

$$\mathbf{A} = \begin{bmatrix} -1.56 & 5.0 & -23.764 \\ 34.257 & 12.34 & -25.074 \\ 95.72 & -84.73 & 0.34 \end{bmatrix}$$

se pide que le asocies un puntero y a través de este puntero escribas los valores de las componentes junto con las respectivas direcciones de memoria.

6. Escribe un programa que permute las filas dadas de una matriz real dada.

7. En este ejercicio se trata de ilustrar la diferencia del concepto de matriz en **C**, con el de *doble puntero* (**** PtrMat**).

a) Declara e inicializa la matriz

$$\mathbf{A} = \begin{bmatrix} -1.56 & 5.0 & -23.764 \\ 34.257 & 12.34 & -25.074 \\ 95.72 & -84.73 & 0.34 \end{bmatrix}$$

con el tipo **`double`**.

b) Declara un *doble puntero a double*

c) Asigna la dirección de la matriz anterior a este puntero (lo más probable es que dé error).

d) Declara un *puntero matricial a double* (de manera similar a cómo se ha hecho en sección 6.4, pág. 145). Asocia la dirección de la matriz a este puntero (ahora, no debería dar error).

La conclusión es que el *doble puntero* se puede utilizar para realizar una distribución matricial de datos, mediante la función **`malloc`**, pero **no** es una matriz desde el punto de vista *riguroso* de **C**. Es interesante comparar la figura 6.4 (pág. 150) con la figura 6.6 (pág. 6.6). En el primer caso, las direcciones de las filas y su valor coinciden, mientras que en el segundo no es así.

8. Crea una matriz dinámicamente utilizando la función **`malloc`**, de manera que se pueda introducir la matriz

$$\mathbf{A} = \begin{bmatrix} -1.56 & 5.0 & -23.764 \\ 34.257 & 12.34 & -25.074 \\ 95.72 & -84.73 & 0.34 \end{bmatrix}$$

después escribe en un fichero esta matriz, con las direcciones de memoria de cada una de sus componentes.

CAPÍTULO 7

Funciones

7.1. Introducción

La noción de *función* en un programa está relacionando con la idea de *modularidad*. Es habitual que los programas informáticos tengan decenas e incluso cientos de miles de líneas de código fuente. A medida que se van desarrollando estos programas su tamaño aumenta y se convierten rápidamente en códigos poco manejables. Para evitar esta situación surge el concepto de *modularidad*, que consiste en dividir el programa en partes más pequeñas con una finalidad más concreta. A estos módulos se les suele llamar: *subprogramas, subrutinas, procedimientos, etc.* En el **lenguaje C** se emplea la palabra ***función***. Ya se ha utilizado la idea de *función* en los programas presentados hasta ahora (pero no se ha mencionado). En todos ellos, el *cuerpo* del código, debe escribirse a partir de la instrucción **`int main(void)`** y entre las llaves: { y }. Realmente **`main`** consiste en una *función*, cuya utilización es obligatoria, por ser la *principal*. La estructura de las funciones en **C** es similar.

Un ejemplo muy básico de función en **C**, aquella que no recibe ni devuelve datos, es la función **`profesor()`**, que se ilustra en el siguiente código:

Código 7.1 – *Ejemplo de función*

```
#include <stdio.h>

void profesor(void);

int main(void)
{

    printf("Quiero aprender C. \n\n");
    profesor();
    printf("Muchas gracias, seguiré sus consejos\n");

    return 0;

}

void profesor(void)
```

```
17 {
18     printf(" Profesor: estudia y haz los ejercicios ");
19     printf("de programación. No te desanimes\n\n");
20
21     return;
22 }
```

La salida en pantalla sería:

```
Quiero aprender C.

 Profesor: estudia y haz los ejercicios de programación. No te desanimes

Muchas gracias, seguiré sus consejos
```

La función definida se llama `profesor` y tal como expresa el *prototipo de la función* (LÍNEA 3) es una función que no contiene argumentos (igual que la función `main`), ni devuelve ningún valor debido a la palabra **`void`**, antes del nombre de **`profesor`**. La definición de la función está en el *fichero fuente*, a partir de la LÍNEA 16. La *llamada a la función* se hace por su nombre, en la LÍNEA 9, en el código principal. En la LÍNEA 3, se presenta lo que se llama *prototipo de la función*. Indica al compilador que se va a definir una función, en este caso, sin argumentos y sin ningún valor de retorno.

Es claro que este tipo de funciones tienen una utilidad más que discutible. Las funciones más interesantes serán aquellas que permitan transmitir valores de variables, para que sean manipuladas y después devuelvan sus nuevos valores al programa principal.

En este capítulo se analizarán los siguientes conceptos relacionados con el concepto de *función*:

Modularidad. Ya se ha comentado, la función o *subrutina* o *subprograma* es la herramienta básica para tratar de fragmentar los programas y por tanto hacerlos más legibles y eficientes. Una función proporciona una forma conveniente de encapsular algunos cálculos, que se pueden emplear después, en distintos programas.

Alcance local de las variables. En **C**, las variables definidas en el programa principal y las funciones son *locales*. Esto quiere decir, que **sólo** son visibles en aquellas partes en dónde se definen, aunque tengan el mismo nombre.

Intercambio de datos. Una función se *activa* mediante una *llamada* del programa principal o de otra función que esté a un nivel superior. En **C** se hace a través del nombre de la función. Para intercambiar información entre variables de distintas funciones, se puede hacer a través de los argumentos de la función o mediante el tipo de valor que puede retornar una función.

7.2. Aspectos fundamentales

En **C**, una función tiene la siguiente estructura:

*Estructura de una función en **C***

```
tipo_de_retorno nombre_función (lista de argumentos con los tipos)
{
       declaración de funciones y/o
       declaración e inicialización de variables

       sentencias ejecutables
```

```
            ...
            ...
        sentencias ejecutables

        return (expresión);
}
```

La primera línea se conoce como el ***encabezamiento*** (*header*) de la función y todo aquello que está entre llaves, recibe el nombre de ***cuerpo*** (*body*). Se muestra con más detalle, algunos de los elementos que forman la *estructura* de una función.

7.2.1. Tipos de las funciones

En la primera línea o *encabezamiento* se hace referencia al

```
tipo_de_retorno
```

Resulta que en **C**, cuando una función es ejecutada puede devolver al programa o función que la ha llamado, un valor (***valor de retorno***) cuyo *tipo*, debe ser especificado en el encabezamiento de la función y se llama ***tipo de retorno*** de la función. Si no se establece el *tipo de retorno* de la función, entonces se considera que la función devolverá un valor **entero**. Es decir, el *tipo de retorno* es **`int`**, mientras no se especifique lo contrario. Si se quiere establecer *explícitamente* que la función **no** devuelva ningún valor, entonces el *tipo de retorno* que debe declararse es **`void`**.

7.2.2. Sentencia `return`

La sentencia **`return`**, puede ir seguida de una **`expresión`** (los paréntesis son opcionales), en cuyo caso, cuando ésta es evaluada, el valor resultante, queda asignado como *valor de retorno* de la función. Por otra parte, permite devolver el control a la siguiente sentencia del programa o función que ha hecho la llamada. Si no existe ningún **`return`**, el control se devuelve una vez llegado al final del *cuerpo* de la función.

7.2.3. Declaración de funciones y prototipos

De la misma manera que se necesitan declarar todas las variables es conveniente (aunque no obligatorio) declarar la función explícitamente, antes de ser utilizada en la función o programa que realiza la llamada. Esta es una práctica recomendable y segura que debe hacerse siempre[1] (para evitar conflictos con los *tipos* en el desarrollo del programa). La declaración de la funciones se hace mediante su ***prototipo***, que coincide esencialmente con la primera línea de definición de la propia función (lo que sería el *encabezamiento*). Esto es,

```
tipo_de_retorno nombre_función (lista de los tipos de argumentos)
```

Sólo es necesario incluir los *tipos* de las variables que formarán el argumento. No se necesitan la identificación de las variables. Los *prototipos* de las funciones permiten al compilador que realice correctamente la conversión del *tipo del valor del retorno*. Este modelo de declaración suele hacerse al comienzo del fichero, después de las directivas `#define` y `#include`.

[1] Por ejemplo, en **C++**, sí es obligatorio

7.2.4. LLamadas a funciones

Las *llamadas a funciones* se realizan incluyendo su nombre en una expresión o sentencia del programa principal o de otra función. Este nombre debe ir seguido de una lista de *argumentos* separados por comas y encerrados entre paréntesis. A los argumentos incluidos en la llamada se les denomina ***argumentos reales***, frente a los ***argumentos formales*** que son los que aparecen en el *encabezamiento* de la definición de la función. Los *argumentos reales* pueden ser, no sólo variables y/o constantes sino también expresiones. Cuando el programa encuentra el nombre de la función (*llama a la función*), evalúa los *argumentos reales*, los convierte, si es necesario, al *tipo* de los *argumentos formales* y ***pasa copias de dichos valores*** (*paso de argumentos por valor*) a la función, junto con el control de la ejecución.

7.2.5. Primeros ejemplos

Lo más directo para familiarizarse con las funciones en **C** es ver algunos ejemplos sencillos y observar cómo se adaptan a la estructura anterior.

Ejemplo: suma de los primeros N números naturales

Una función un poco más elaborada, consiste en aquella que tiene algún dato de entrada, o dicho de manera más formal, que tiene algún argumento de entrada. Se quiere construir una función que sume los primeros n números naturales. Este algoritmo ya se había planteado anteriormente (véase sección 2.4, pág. 19), pero ahora el interés está en construir una función, cuyo argumento de entrada sea el valor n. Una posible solución se muestra en el código 7.2

Código 7.2 – *Suma de los primeros números naturales*

```
#include <stdio.h>

void SumaNaturales(int ); //** Prototipo

int main(void)
{
    int N;
    int chk;

    // **** Entrada de datos ***********
    do {
        printf("\n\n Introduce un número natural (N > 1) ->  ");
        chk = scanf("%i", &N);
        while(getchar() != '\n');
    } while(chk != 1 || N < 2 );
    printf("\n\n");

   // ***** LLamada  a la función ***
    SumaNaturales(N);  //*** Argumento REAL

    return 0;
}

void SumaNaturales(int K)  // *** Argumento FORMAL
{
    int Suma;
    int i;
```

```
28
29      for (i = 0, Suma = 0; i <= K; i++){
30          Suma += i;
31      }
32      printf("La suma de los %i primeros números naturales es %i\n\n", K, Suma);
33
34      return;
35  }
```

La salida en pantalla sería de la forma

```
Introduce un número natural (N > 1) -> 15

La suma de los 15 primeros números naturales es 120
```

En este ejemplo, resulta importante destacar la LÍNEA 3 que es en dónde se define el *prototipo de la función*. Posteriormente en la LÍNEA 19 se hace la llamada a la función **`SumaNaturales(N)`** (con el *argumento real* **`N`**), que *lleva el control del programa* a la LÍNEA 24 en donde empieza la definición de la función.

Una posible mejora del programa es que la salida se produzca en un fichero de datos. Para ello se introduce un nuevo argumento en la función **`SumaNaturales`** que sea el *puntero a fichero*. En el siguiente código 7.3 se muestra el procedimiento:

Código 7.3 – *Suma de los primeros números naturales*

```
1   #include <stdio.h>
2
3   void SumaNaturales(FILE *, int ); //** Prototipo
4
5   int main(void)
6   {
7       int N;
8       int chk;
9
10      FILE *fi;
11
12      fi = fopen("Datos.dat", "w");
13
14
15      // **** Entrada de datos ************
16      do {
17          printf("\n\n Introduce un número natural (N > 1) ->  ");
18          chk = scanf("%i", &N);
19          while(getchar() != '\n');
20      } while(chk != 1 || N < 2 );
21      printf("\n\n");
22
23      // *** LLamada a la función ***
24      SumaNaturales(fi, N); //** Argumentos REALES
25
26      fclose(fi);
27
28      return 0;
29  }
30  void SumaNaturales(FILE *fo, int K) //** Argumentos FORMALES
```

```
{
    int Suma;
    int i;

    for (i = 0, Suma = 0; i <= K; i++){
        Suma += i;
    }
    fprintf(fo, "La suma de los %i primeros números naturales es %i\n\n",
                  K, Suma);

    return;
}
```

en la pantalla aparecería

```
Introduce un número natural (N > 1) -> 15
```

y en el fichero **Datos.dat**, la frase

```
La suma de los 15 primeros números naturales es 120
```

Otra alternativa sería que el valor de la suma *retornara* al programa principal y ahí, imprimirlo. Para ello hay que dotar a la función **SumaNaturales** de un *tipo* de *retorno*. Es lo que se ha llamado *tipo de la función*. En el código 7.4 se observa este recurso:

Código 7.4 – *Suma de los primeros números naturales*

```
#include <stdio.h>

int SumaNaturales(int ); //** Prototipo

int main(void)
{
    int N, Suma = 0;
    int chk;

    FILE *fi;

    fi = fopen("Datos.dat", "w");


    // **** Entrada de datos ************
    do {
        printf("\n\n Introduce un número natural (N > 1) ->  ");
        chk = scanf("%i", &N);
        while(getchar() != '\n');
    } while(chk != 1 || N < 2 );
    printf("\n\n");

    // *** LLamada a la función
    Suma = SumaNaturales(N);  //*** Argumento REAL

    fprintf(fi, "La suma de los %i primeros números naturales es %i\n\n",
                                          N, Suma);
```

```
28
29      fclose(fi);
30      return 0;
31  }
32
33  int SumaNaturales(int K) // *** Argumento FORMAL
34  {
35      int Resultado;
36      int i;
37
38      for (i = 0, Resultado = 0; i <= K; i++){
39          Resultado += i;
40      }
41
42
43      return (Resultado); //Devuelve el valor "Resultado"
44  }
```

de nuevo, en la pantalla aparecería

```
Introduce un número natural (N > 1) -> 15
```

y en el fichero **Datos.dat**, la frase

```
La suma de los 15 primeros números naturales es 120
```

Merece la pena, detenerse un momento en la definición de la función **SumaNaturales**, situada en las LÍNEAS 33-44. En el *encabezamiento* de la función (LÍNEA 33) ya aparece el *tipo* de la función, que corresponde a un **int**. Además se define la variable **Resultado** (LÍNEA 35) con el objetivo de almacenar el resultado final de la suma. Por último, su valor se *devuelve* al programa principal, a través de la instrucción **return** y se asigna a la variable **Suma** (LÍNEA 24) en el programa principal.

Es importante hacer notar que **lenguaje C** asigna distintas posiciones de memoria a las variables definidas en las *subrutinas o funciones*, respecto de las definidas en el programa principal. Esto es, aunque en la definición de la función **SumaNaturales** se hubiese utilizado el nombre de **Suma** en lugar de **Resultado**, para **C** la posición de memoria sería distinta a la ocupada por la variable **Suma** del programa principal y por tanto, distintas. Este tema se abordará con más detalle en la sección 7.3 (pág. 176).

Ejemplo: valor absoluto

Al describir el *Operador condicional* (véase sección 4.2.2, pág. 71) ya se planteó el código de la función *valor absoluto* en forma de *macro*. (véase el código 4.3, pág. 71). Sin embargo, debido a su sencillez, resulta de interesante su descripción como función de **C**. Una primera programación, siguiendo los aspectos anteriormente considerados, se muestra en el código 7.5:

Código 7.5 – *Función: valor absoluto*

```
1  #include <stdio.h>
2
3  /* Ejemplo de programación de la función 'absoluto' */
4
5  int absoluto(int );  /*** Prototipo de la función definida ***/
6
```

```
7  int main(void)
8  {
9
10     int a = 10, b = 0, c = -22;
11     int d, e, f;
12
13     FILE *fi;
14     fi = fopen("Datos.dat", "w");
15
16     d = absoluto(a);
17     e = absoluto(b);
18     f = absoluto(c);
19     fprintf(fi, "Los valores absolutos de %d; %d; %d ", a, b, c);
20     fprintf(fi, " son: %d; %d; %d\n", d, e, f);
21
22     fclose(fi);
23
24     return 0;
25 }
26
27 /* Función valor absoluto */
28 int absoluto(int x)  // *** Argumento FORMAL
29 {
30         int y;
31         y = ( x < 0 ) ? -x : x; /* Operador condicional */
32         return (y); /* Devuelve el valor de y a main()*/
33 }
```

cuya salida, almacenada en el fichero `Datos.dat` tendría la forma:

```
Los valores absolutos de 10; 0; -22  son: 10; 0; 22
```

La definición de la función **absoluto** está entre las LÍNEAS 28-33. En el *encabezamiento* se ha realizado una declaración específica sobre el *tipo de retorno*, en este caso **int**. Se insiste de nuevo, que el argumento: **int** x, que aparece en la definición, corresponde al *argumento formal*. Sin embargo, en las *llamadas a la función* **absoluto** (LÍNEAS 16–18), las variables `a, b, c` son *argumentos reales*, respectivamente. En la LÍNEA 5 se encuentra definido el *prototipo* de la función **absoluto**.

Tal y como se ha planteado el código anterior, la función **absoluto** sólo está definida para valores enteros. Es lógico, ampliar el espectro de esta función, considerando también números reales que puedan introducirse por teclado. Además la definición se puede compactar, eliminado la declaración de la variable y. Otra versión un poco más realista sería:

Código 7.6 – *Función: valor absoluto*

```
1  #include <stdio.h>
2
3  /* Ejemplo de programación de la función 'absoluto' */
4
5  double absoluto(double );  /* Prototipo de la función definida */
6
7  int main(void)
8  {
9
```

```
10      double N;
11      int chk;
12
13      // **** Entrada de datos ************
14      do {
15          printf("\n\n Introduce un número para determinar su valor absoluto ->  "
                );
16          chk = scanf("%lf", &N);
17          while(getchar() != '\n');
18      } while(chk != 1);
19      printf("\n\n");
20
21      printf("El valor absoluto de %.4f es %.4f\n\n", N, absoluto(N));
22
23      return 0;
24  }
25
26  /* Función valor absoluto */
27  double absoluto(double x)
28  {
29      return ( (x < 0) ? -x : x  );
30  }
```

cuya salida en pantalla podría ser:

```
El valor absoluto de -34.7000 es 34.7000;
```

Ejemplo: potencia de 2

Otro ejemplo, consiste en definir una función que calcule la potencia entera de 2. La función se define con *tipo* **long long** y tiene como argumentos la *base*, que se toma como constante el valor de 2 y un valor entero, que es el exponente. El resultado se imprime en un fichero de datos. La codificación puede verse en el programa 7.7.

Código 7.7 – *Función: potencia de 2*

```
1   #include <stdio.h>
2
3   /* Función potencia entera de 2 */
4
5   long long f_potencia(unsigned, int);
6
7   int main(void)
8   {
9
10      const unsigned int BASE = 2;
11      int N;
12      int chk;
13
14      FILE *fi;
15
16      fi = fopen("Datos.dat", "w");
17
```

```
18          // **** Entrada de datos ************
19      do {
20          printf("\n\n Introduce un número natural (N > 1) ->  ");
21          chk = scanf("%i", &N);
22          while(getchar() != '\n');
23      } while(chk != 1 || N < 2 );
24      printf("\n\n");
25
26      fprintf(fi, "La potencia %i de 2 es %i\n\n", N, f_potencia(BASE, N));
27
28      fclose(fi);
29      return 0;
30
31  }
32  long long f_potencia(unsigned k, int b)
33  {
34      int incremento;
35      long long potencia;
36
37      for(incremento = 1, potencia = 1.; incremento <= b; incremento++){
38          potencia = potencia * k;
39      }
40      return (potencia);
41  }
```

La entrada se realiza por pantalla, de la forma:

```
Introduce un número natural (N > 1) ->  25
```

y la salida se escribe en el fichero de datos:

```
La potencia 25 de 2 es 33554432
```

Ejemplo: factorial

El cálculo del factorial de un número, ya tratado en ejercicios anteriores (véase por ejemplo, sección 4.6 de Problemas, pág. 90), se puede calcular a través de una función, muy fácilmente. En este ejemplo, se añade una función que escriba en un fichero de salida, los datos del alumno junto con el valor del factorial. Un posible programa se muestra en el código 7.8.

Código 7.8 – *Función: factorial*

```
1   #include <stdio.h>
2
3   /** Función factorial  **/
4
5   void Presentacion(FILE * );  // *** Prototipos ***
6   long long Factorial(int );   //
7
8   int main(void)
9   {
10      int N;
11      unsigned long long Fact;
```

```
12      unsigned int incremento;
13      int chk;
14
15      FILE *fi;
16
17      fi = fopen("Datos.dat", "w");
18
19      Presentacion(fi); // *** LLamada a la función Presentación
20
21      // **** Entrada de datos ***********
22
23      do {
24          printf("\n\n Introduce un número natural (1 < N < 21) ->  ");
25          chk = scanf("%i", &N);
26          while(getchar() != '\n');
27      } while(chk != 1 || N < 2 );
28      printf("\n\n");
29
30      // *** Imprime el resultado a través de la función Factorial
31      fprintf(fi, "El factorial de %i es %i! = %i", N, N, Factorial(N));
32
33      fclose(fi);
34
35      return 0;
36
37  }
38
39  void Presentacion(FILE *fo)
40  {
41      fprintf(fo, "\n **** Apellidos: AAAAAAA \n");
42      fprintf(fo, "\n **** Nombre: AAAAA\n");
43      fprintf(fo, "\n **** Numero Mat: 111111\n");
44      fprintf(fo, "\n **************************\n\n");
45
46      return;
47  }
48
49  long long Factorial (int K)
50  {
51      long long Fact;
52      int incremento;
53
54      for(Fact = 1ULL, incremento = 1; incremento <= K; incremento++){
55          Fact *= incremento; //
56      }
57
58      return (Fact);
59  }
```

Cuando se ejecuta el programa, en la pantalla se obtiene

```
Introduce un número natural (1 < N < 21) ->  15
```

y el resultado se imprime en el fichero **Datos.dat**, de la forma:

```
 **** Apellidos: AAAAAAA

 **** Nombre: AAAAA

 **** Numero Mat: 111111

 *************************
El factorial de 15 es 15! = 20043100
```

Ejemplo: raíz cuadrada

El **lenguaje C** permite trabajar con *funciones anidadas*. Esto es, funciones que en su definición aparecen otras funciones, definidas previamente. Como ejemplo sencillo para ilustrar esta situación se propone calcular la *raíz cuadrada* de un número positivo, utilizando el *algoritmo de Newton–Raphson*. La idea es muy simple: dado $x \geqslant 0$ se quiere calcular $\sqrt{x}$, o lo que es lo mismo, dado $x \geqslant 0$, buscar un valor t tal que:

$$t^2 = x \iff t^2 - x = 0$$

A partir de aquí, se puede considerar la función

$$f(t) = t^2 - x$$

y por tanto, la ecuación

$$f(t) = 0$$

El *algoritmo de Newton-Raphson* se asocia a un *algoritmo iterativo de punto fijo* a través de la función de iteración

$$g(t) = t - \frac{f(t)}{f'(t)} = \frac{t^2 + x}{2t} = \frac{t}{2} + \frac{x}{2t} = \frac{1}{2}\left(t + \frac{x}{t}\right)$$

con lo cual, la solución se obtiene mediante el proceso iterativo

$$t_{n+1} = g(t_n)$$

que genera una sucesión convergente a la solución[2] (véase por ejemplo, Mathews-Fink [14, 80]). Para establecer una aproximación a la solución, se considera como criterio de parada:

$$|t_n^2 - x| < \varepsilon$$

donde ε es un valor pequeño, escogido previamente (por ejemplo, $\varepsilon = 10^{-6}$).

Una posible codificación de este algoritmo, se muestra en el código 7.9

Código 7.9 – *Raíz cuadrada de un número*

```
1 #include <stdio.h>
2
3 /* Cálculo de la raíz cuadrada mediante
4 el algoritmo de Newton--Raphson */
5
6 double Raiz2(double);      // *** Prototipos de
7 double absoluto(double ); // *** funciones
```

[2] En general para otro tipo de funciones las condiciones para convergencia pueden cambiar. Véase Mathews-Fink [14, pág. 77].

```
8
9   int main(void)
10  {
11
12      double Real, Raiz;
13      int chk;
14
15      FILE *fi;
16
17      fi = fopen("Datos.dat", "w");
18
19          // **** Entrada de datos ************
20      /*do {
21          printf("\n\n Introduce un número real NO negativo (R >= 0) ->  ");
22          chk = scanf("%lf", &Real);
23          while(getchar() != '\n');        //Alternativa a la entrada de datos
24      } while(chk != 1 || Real < 0 );
25      printf("\n\n");*/
26
27      //Raiz = Raiz2(Real); // Alternativa a la llamada de la función
28
29      fprintf(fi, "La raiz cuadrada de 3 es %.8f\n\n", Raiz2(3.));
30      fprintf(fi, "La raiz cuadrada de 225 es %.8f\n\n", Raiz2(225.));
31      fprintf(fi, "La raiz cuadrada de 32.32 es %.8f", Raiz2(32.32));
32
33      fclose(fi);
34      return 0;
35
36  }
37
38  double absoluto(double x)  // *** Argumento FORMAL
39  {
40      return ( (x < 0) ? -x : x  ); /* Devuelve el valor de y a main()*/
41  }
42
43  double Raiz2(double x) // *** Argumento FORMAL
44  {
45      double t_i = 1.;
46      const double epsilon = .000001;
47
48
49      // *** Se utiliza la función absoluto()
50      while ( epsilon <= absoluto(t_i * t_i - x) ){
51          t_i = (t_i + x/t_i) / 2.;
52      }
53      return (t_i);
54  }
```

Si se ejecuta el programa sin introducir los datos por pantalla, el resultado en el fichero **`Datos.dat`** es

```
La raiz cuadrada de 3 es 1.73205081

La raiz cuadrada de 225 es 15.00000000

La raiz cuadrada de 32.32 es 5.68506816
```

En la LÍNEA 50 la función **Raiz2** hace uso de la función **absoluto** que también se ha definido previamente. Para el compilador, no es importante el orden en la que se hayan definido. Lo importante es que estén *localizables*. En este caso en el mismo *fichero fuente* que el programa principal.

7.3. Visibilidad de las variables

Cada función puede disponer de sus propias variables, declaradas al comienzo de su código. Como ya se ha comentado (véase sección 7.1, pág. 163 o el párrafo anterior, al ejemplo **Ejemplo: valor absoluto**, pág. 169), estas variables, mientras no se especifique lo contrario, son sólo *visibles*, dentro del bloque en el que han sido definidas. Para **C** estas variables son de *visibilidad **auto*** (*automatic*). Aunque no es necesario, se puede poner **auto** en la declaración de las variables para hacerlo explícitamente. Un ejemplo, del comportamiento de este tipo de variables se puede seguir en el código 7.10.

Código 7.10 – *Variables automáticas*

```
1  #include <stdio.h>
2
3  /* Programa que muestra el alcance o visibilidiad de
4  las variables automáticas */
5
6  void Prueba(FILE *, int); //Prototipo
7
8  int main(void)
9  {
10
11     auto int Dato; // *** NO es necesario
12     auto int chk;  // *** la palabra: auto
13
14     FILE *fi;
15
16     fi =fopen("Salida.dat", "w");
17
18         // **** Entrada de datos ************
19     do {
20         printf("\n\n Introduce un número natural (N > 1) ->  ");
21         chk = scanf("%i", &Dato);
22         while(getchar() != '\n');
23     } while(chk != 1 || Dato < 2 );
24     printf("\n\n");
25
26     fprintf(fi, "Antes de entrar en la función\n");
27     fprintf(fi, "Dato = %i, Dirección: %u", Dato, &Dato);
28     fprintf(fi, "\n\n");
29
30     Prueba(fi, Dato); //*** Argumento REAL
31
32     fprintf(fi,"\n\n");
33     fprintf(fi, "Estoy de nuevo, en el programa principal\n");
34     fprintf(fi, "Dato = %i, Dirección: %u", Dato, &Dato);
35     fprintf(fi, "\n\n");
36
```

```
37      fclose(fi);
38      return 0;
39
40  }
41  void Prueba(FILE *fo, int Dato) //*** Argumento FORMAL
42  {
43      fprintf(fo, "Estoy dentro de la función Prueba\n");
44      fprintf(fo, "Dato = %i, Dirección: %u\n", Dato, &Dato);
45      Dato  = Dato + 20;
46      fprintf(fo, "Dato + 20 = %i, Dirección: %u", Dato, &Dato);
47      return;
48  }
```

Cuando se ejecuta el programa, en la pantalla aparece,

```
Introduce un número natural (N > 1) ->  15
```

y el resultado se imprime en el fichero **Salida.dat**:

```
Antes de entrar en la función
Dato = 15, Dirección: 2293312

Estoy dentro de la función Prueba
Dato = 15, Dirección: 2293288
Dato + 20 = 35, Dirección: 2293288

Estoy de nuevo, en el programa principal
Dato = 15, Dirección: 2293312
```

En este código se declara la variable **Dato** en el programa principal (LÍNEA 11). Posteriormente, por teclado se le asigna un valor de 15. Con este valor entra en la función **Prueba** (LÍNEA 30), en la que también existe una variable con el mismo nombre de **Dato** (LÍNEA 41), que recibe el valor de 15. En esta función se modifica el valor de la variable **Dato** a 35 (LÍNEA 45) y se devuelve el *control* al programa principal. Posteriormente se imprime el valor de la variable **Dato** y tiene el valor de 15 (LÍNEA 34). Esto es así, porque como se observa en la salida del programa, impresa en fichero **Salida.dat**, la variable **Dato**, definida en el programa principal y la variable **Dato**, declarada en la función **Prueba** tienen direcciones de memoria *distintas*. Esto implica que para **C**, aunque tienen el mismo nombre son variables *distintas*[3].

Una interpretación gráfica de este proceso puede verse en la figura 7.1

[3]Los humanos también somos capaces de distinguir personas conocidas, con el mismo nombre.

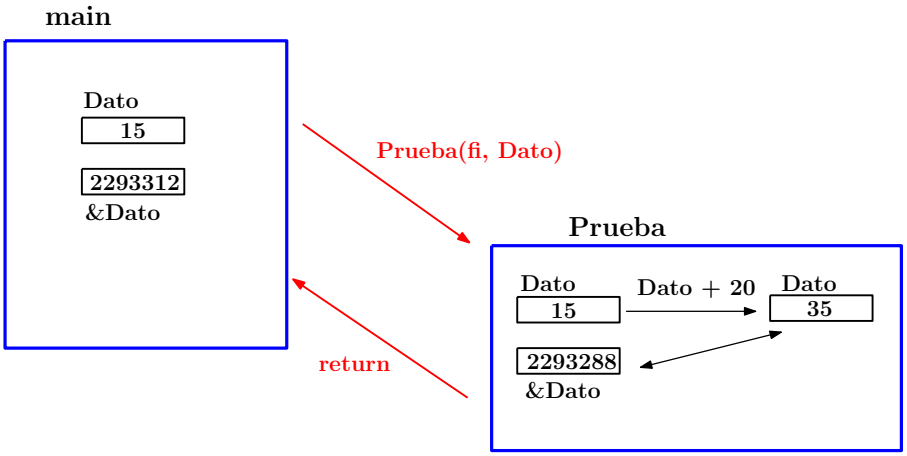


Figura 7.1 – *Visibilidad de variables automáticas*

Si las variables se declaran como ***static*** (*estáticas*), entonces aunque tienen la misma *visibilidad* que las ***auto***, se diferencian en que su valor permanece en memoria. Esto es, su valor no desaparece cuando la función que las contiene, *finaliza su trabajo*; el programa recuerda su valor, si la función vuelve a ser llamada. Las variables que son *visibles* en todo momento y en toda parte del programa se llaman *globales* y en **C** se declaran como ***extern*** (*externas*). El programa 7.11 muestra el efecto de los tres tipos de *visibilidad* en las variables.

Código 7.11 – *Diferencias entre variables auto, static y extern*

```
#include <stdio.h>

/* Programa para observar el efecto de
distintos visibilidades de variables */

void Auto_static_extern(FILE *); //*** Prototipo de la función.

int Var_extern; //** Declaración variable GLOBAL

int main(void)
{

    int i;
    FILE *fi;

    fi = fopen("Datos.dat", "w");

    for (Var_extern = 1, i = 0; i < 7; i++) {
        Var_extern++;
        Auto_static_extern(fi);
    }


    fclose(fi);
    return 0;

}

void  Auto_static_extern(FILE *fo)
```

```
30  {
31      int Var_auto = 1;
32      static int Var_static = 1;
33      extern int Var_extern;
34
35      fprintf(fo, "Var_auto = %i;  Var_static = %i;  Var_extern = %i\n\n",
36                   Var_auto, Var_static, Var_extern);
37      Var_auto++;
38      Var_static++;
39      Var_extern++;
40
41      return;
42  }
```

Cuando se ejecuta el programa, en el fichero **`Datos.dat`** aparece:

```
Var_auto = 1;  Var_static = 1;  Var_extern = 2

Var_auto = 1;  Var_static = 2;  Var_extern = 4

Var_auto = 1;  Var_static = 3;  Var_extern = 6

Var_auto = 1;  Var_static = 4;  Var_extern = 8

Var_auto = 1;  Var_static = 5;  Var_extern = 10

Var_auto = 1;  Var_static = 6;  Var_extern = 12

Var_auto = 1;  Var_static = 7;  Var_extern = 14
```

En este ejemplo se observa claramente el efecto de los tres tipos de variables. Es interesante destacar la definición de la variable **`Var_extern`**. Su declaración se realiza en la LÍNEA 8, *antes* de la función **`main`**. Posteriormente se inicializa en la LÍNEA 18 y dentro del bloque **`for`** se aumenta en una unidad. El efecto de su *globalidad* es inmediato al entrar en la función **`Auto_static_extern`**, pues mantiene su valor y además aumenta en una unidad antes de regresar al programa principal. Seguidamente vuelve a aumentar en una unidad y antes de entrar de nuevo en la función **`Auto_static_extern`** y vuelve a mantener este valor dentro de la función. Por esto en la salida, su incremento es de dos unidades en dos unidades. También se observa claramente el efecto de considerar una variable *static*. Una vez declarada e inicializada en la función mantiene su valor cada que se vuelve a entrar en la función.

En principio, las variables tipo *static* y *extern* no serán utilizadas en este documento.

7.4. Regiones de la memoria

Durante la ejecución de un programa, resulta básico la gestión y distribución eficiente y óptima, de los distintos elementos del código (sentencias, variables, funciones), en la memoria del computador.

Como se comentó en el capítulo: *Elementos de un ordenador* (en la pág. 1), el primer responsable de la gestión de la memoria es el sistema operativo.

Generalmente, al ejecutar un programa (una vez generado el fichero `.exe`), un módulo del sistema operativo, llamado *cargador*, asigna y direcciona cierta cantidad de memoria, además de cargar el código a ejecutar, en dichas direcciones. Por tanto, el código del programa resultante de la compilación, debe organizarse de forma que haga uso de este bloque, por lo que el compilador debe incorporar al *fichero objeto*, todo el código necesario para ello.

Una vez que el programa se está ejecutando, se plantea la necesidad de la *gestión de memoria en tiempo de ejecución*. Las técnicas de esta *gestión de la memoria en tiempo de ejecución* difieren de unos lenguajes de programación a otros, e incluso de unos compiladores a otros. En el caso del lenguaje **C**, como en la mayoría de los lenguajes prodecimentales e imperativos (FORTRAN, PASCAL, MÓDULA-2, COBOL, ...), la memoria asignada se divide en tres regiones, áreas o *segmentos*, básicamente:

- Área de tamaño fijo.
- Pila (*Stack*)
- Área de gestión dinámica (*Montón o Heap*)

Esta distinción es un tanto arbitraria. Otros autores la descomponen en dos grupos únicamente: *área de tamaño fijo* y *área tamaño variable*. En esta última, agrupan la *pila* y la *gestión dinámica*. En este documento, se seguirá la planteada inicialmente, que suele ser la más habitual (véase la figura 7.2).

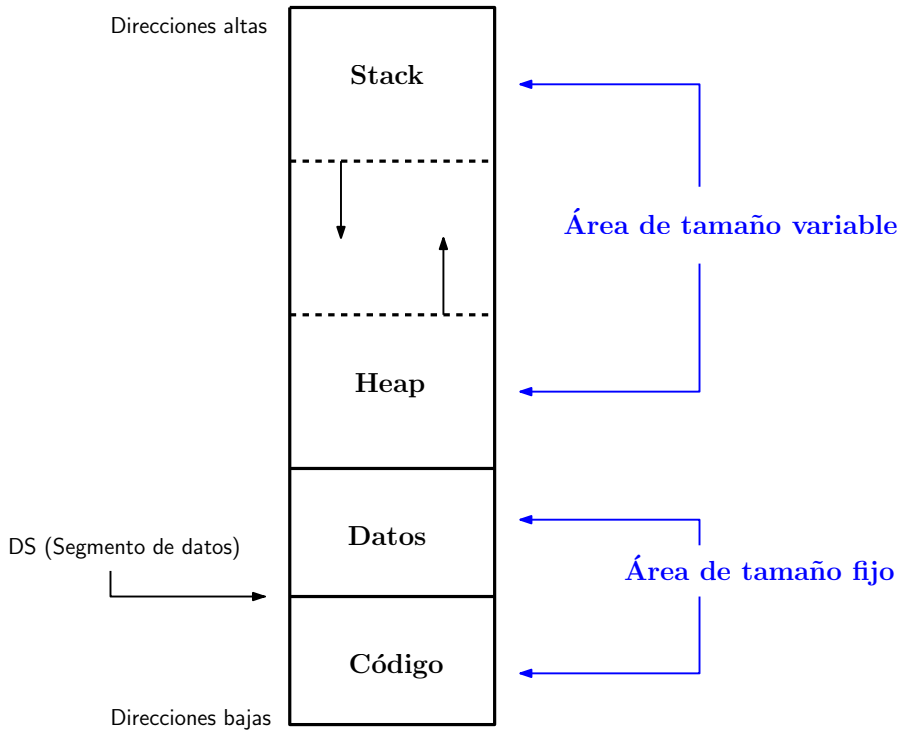


Figura 7.2 – *Disposición de los segmentos de memoria durante el proceso de ejecución*

Área de tamaño fijo. Se divide a su vez en dos grupos:

- **Segmento de código.** Es la zona donde se almacenan las instrucciones del programa ejecutable en código máquina, y también el código correspondiente a las funciones que utiliza. Su tamaño y su contenido se establecen en tiempo de compilación y por ello se dice que su tamaño es fijo en tiempo de ejecución. De esta forma, el compilador, a medida que va generando código, determina las posiciones de memoria consecutivas necesarias para situar el código en esta zona, delimitando convenientemente el inicio de cada función, junto el del programa principal. Este código ejecutable también recoge información acerca de las necesidades de memoria en *tiempo de ejecución*, tanto para variables de tamaño fijo, como aquellas de tamaño variable. Esta información será utilizada por el *cargador* en el momento de hacer la carga del *fichero ejecutable* en memoria principal, para proceder a su ejecución.

- **Segmento de datos.** En esta región el compilador permite almacenar constantes y variables globales y estáticas (aquellas declaradas con **`static`**). Si en un *fichero fuente* se encuentra una variable declarada fuera del ámbito de las funciones (incluyendo la principal, `main`), entonces el compilador la considera *global* y le asigna un espacio determinado, con las referencias necesarias, en esta área. Los datos almacenados aquí, tienen ciertas características:
  - El tamaño de las variables queda determinado durante la compilación y no puede ser cambiado durante la ejecución del programa.
  - El tiempo de vida es la duración del programa.
  - Las variables son visibles para todas las funciones definidas después de ellas, en el archivo en que se definan.

  Es inmediato darse cuenta que la gestión de este tipo de memoria resulta sencilla. De hecho, algunos lenguajes de programación antiguos, como las primeras versiones de FORTRAN, únicamente tenían reserva de *memoria estática*. Las principales ventajas son la sencillez de implementaciónn y que las necesidades de memoria del programa son conocidos en el momento de la compilación. Sin embargo, existen ciertos inconvenientes importantes:
  - El tamaño de los objetos debe ser conocido en tiempo de compilación. Esto implica que no se puede trabajar con objetos de longitud variable. Por lo general, es difícil para el programador definir el tamaño de las estructuras que va a usar. En efecto, si son demasiado grandes, se desperdicia memoria; si son pequeñas, no puede utilizar el programa en todos los casos.
  - No se pueden implementar procedimientos recursivos.

Los inconvenientes planteados para las variables situadas en el segmento *estático*, hace que sea necesario implementar una gestión dinámica en la memoria durante el proceso de ejecución. Esto es lo que hace el **C**, en dos regiones llamadas *Pila* y *Montón* o *Área de gestión dinámica*, que se estudian a continuación.

Pila (Stack). Como ya se ha visto, en **C** una variable definida en una función (sin especificador o bien con el de **`auto`**), no es *visible* en la función previa que ha realizado la llamada a la primera. Es decir, que la variable *desaparece* cuando la ejecución del programa *sale* de la función en donde ha sido definida.

Las variables con estas características se conocen como *automáticas* y en **C** se construyen con el especificador **`auto`** o bien, sin añadir ninguno. Debido a la naturaleza de estas variables, su gestión debe ser dinámica pues, hay momentos en los que el programa puede acceder a ellas y en otros no.

Por esta razón, existe una región de memoria específica, llamada ***Pila o Stack***, para almacenar estas variables. Los argumentos y las variables locales, son asignados y desasignados de manera dinámica durante la ejecución de las correspondientes funciones. Esto se realiza de manera automática por el código generado por el compilador, sin que el programador tenga acceso a la gestión.

La mecánica de esta gestión de los datos se ajusta a un proceso de naturaleza LIFO (*Last Input– First Output*), que resulta muy eficiente y rápido. Permite direccionar eficazmente variables que serán usadas frecuentemente y a la vez posibilita ahorrar espacio de direccionamiento, ya que puede reutilizar el espacio de memoria dedicado a la función, cuando ésta termina. Además posibilita el diseño de funciones recursivas y reentrantes, asociando un espacio diferente para las variables por cada llamada de la función.

Segmento de gestión dinámica (Heap). Se trata de un área de memoria al que tiene acceso el programador mediante las función **`malloc`** y **`calloc`**, a través de punteros. Se accede a esta memoria en tiempo de ejecución, lo cual hace que los programas sean muy flexibles, pero tiene el inconveniente que el acceso puede ralentizarse, ya que durante la ejecución deben buscarse las direcciones libres. Es por ello, que resulta fundamental liberar los espacios que dejan de ser utilizados durante la ejecución, mediante la función **`free`**.

Desde un punto de vista de la programación, con este tipo de variables, resulta crucial la utilización adecuada de **free**. En efecto, hay que destacar (véase el libro de Prata [15, pág. 547]) que la cantidad de memoria asignada al *área de tamaño fijo* durante el tiempo de ejecución es fija (como su propio nombre indica), además la cantidad de memoria utilizada por las *variables automáticas* puede crecer y disminuir automáticamente durante la ejecución del programa. Sin embargo, la cantidad de memoria utilizada por el *segmento de gestión dinámica*, únicamente crece a menos que se utilice la función **free**. Si no se realiza este proceso se puede llegar a una situación conocida como *falta de memoria del sistema (memory leaks)*.

7.4.1. Gestión dinámica y direccionamiento dinámico

Como ya se comentó en la sección: *Gestión dinámica* (en la pág. 152), la gestión dinámica resulta más flexible a la hora de direccionar un *array* que simplemente el *direccionamiento dinámico*. Pero una vez analizadas, las diferentes regiones de memoria que gestiona un programa en **C**, esta diferencia incluso, se aprecia más claramente. En efecto, en el caso de la *gestión dinámica*, las variables creadas con este procedimiento se almacenan durante la ejecución en el *área de gestión dinámica* o *Heap*, mientras que las variables creadas mediante un proceso de direccionamiento dinámico (véase sección: *Dimensionamiento dinámico (Variable Length Arrays – VLA)*, en la pág. 125) son variables *automáticas* almacenadas, por tanto, en la *Pila* o *Stack*. Esta diferencia es crucial y tiene también sus consecuencias si se utilizan este tipo de variables como argumentos de funciones, como se verá más adelante.

7.5. Paso de argumentos

Como ya se ha comentado anteriormente (véase sección: **LLamadas a funciones**, pág. 166), el *paso de información* sobre los argumentos se realiza en el momento de la *llamada a la función*. Esto implica que a los *argumentos formales* se les envía una ***copia del valor*** de los *argumentos reales*, junto con el control de la ejecución del programa. Esto es lo que se llama, el *paso de argumentos* ***por valor***. Este procedimiento tiene unas consecuencias importantes, cuando se quiere compartir el valor de las variables entre funciones y el programa principal. Este efecto se puede apreciar en el código 7.12, en el que se define la función **Intercambio**, cuyo objetivo es intercambiar los valores asignados, en el *programa principal*, de dos variables.

Código 7.12 – *Intercambio de valores entre dos variables*

```
1  #include <stdio.h>
2
3  /* Programa para intercambiar los valores
4     de dos variables */
5
6  void Intercambio(FILE *, int, int);
7
8
9  int main(void)
10 {
11
12     int a, b;
13     FILE *fi;
14
15     fi = fopen("Datos.dat", "w");
16
17     a = 12;
18     b = -35;
```

```
19      fprintf(fi, "En el programa principal se han asignado los valores\n");
20      fprintf(fi, "a = %3i &a = %8u\n", a, &a);
21      fprintf(fi, "b = %3i &b = %8u", b, &b);
22
23      fprintf(fi, "\n\n");
24      Intercambio(fi, a,b);
25      fprintf(fi, "\n\n");
26
27      fprintf(fi, "En el programa principal los nuevos valores son:\n");
28      fprintf(fi, "a = %3i &a = %8u\n", a, &a);
29      fprintf(fi, "b = %3i &b = %8u\n", b, &b);
30
31
32      fclose(fi);
33      return 0;
34
35  }
36
37  void  Intercambio(FILE *fo, int x, int y)
38  {
39      int temp;
40
41      fprintf(fo, "   Valores recibidos del programa principal:\n");
42      fprintf(fo, "     a = %3i    &a = %8u\n", x, &x);
43      fprintf(fo, "     b = %3i    &b = %8u\n", y, &y);
44
45      temp = x;
46      x = y;
47      y = temp;
48
49      fprintf(fo, "   Una vez realizado el intercambio sus valores son:\n");
50      fprintf(fo, "     a = %3i    &a = %8u\n", x, &x);
51      fprintf(fo, "     b = %3i    &b = %8u\n", y, &y);
52      fprintf(fo, "   Se devuelve el control al programa principal");
53
54      return;
55  }
```

Cuando se ejecuta el programa, en el fichero de salida se muestra:

```
En el programa principal se han asignado los valores
a =  12 &a =  2293316
b = -35 &b =  2293312

   Valores recibidos del programa principal:
     a =  12    &a =  2293288
     b = -35    &b =  2293296
   Una vez realizado el intercambio sus valores son:
     a = -35    &a =  2293288
     b =  12    &b =  2293296
   Se devuelve el control al programa principal

En el programa principal los nuevos valores son:
a =  12 &a =  2293316
b = -35 &b =  2293312
```

Tal y como está desarrollado el código, **no** se obtiene el resultado propuesto, porque como se observa en las

dos últimas líneas del fichero de salida, los valores de las variables **a** y **b**, **no** han sido intercambiados. Sin embargo, sí puede apreciarse en el fichero de salida, que los valores fueron intercambiados *dentro* de la función **Intercambio**.

Esto es consecuencia de la naturaleza *automática* de las variables, como ya se comentó en la sección 7.3, llamada **Visibilidad de las variables** (pág. 176). Las variables declaradas en las funciones tienen posiciones de memoria distintas, a las declaradas en otro programa o en el *programa principal* y por tanto son variables ***diferentes***. Esta circunstancia puede apreciarse de nuevo, en el fichero **Datos.dat**.

Para solucionar esta contrariedad, se plantea no pasar los *valores de las variables*, sino las *direcciones de las variables*, como se muestra ahora en programa 7.13.

Código 7.13 – *Intercambio de valores entre dos variables*

```
#include <stdio.h>

/* Programa para intercambiar los valores
   de dos variables */

void Intercambio(FILE *, int *, int *);


int main(void)
{

    int a, b;
    FILE *fi;

    fi = fopen("Datos.dat", "w");

    a = 12;
    b = -35;
    fprintf(fi, "En el programa principal se han asignado los valores\n");
    fprintf(fi, "a = %3i &a = %8u\n", a, &a);
    fprintf(fi, "b = %3i &b = %8u", b, &b);

    fprintf(fi, "\n\n");
    Intercambio(fi, &a, &b);
    fprintf(fi, "\n\n");

    fprintf(fi, "En el programa principal los nuevos valores son:\n");
    fprintf(fi, "a = %3i &a = %8u\n", a, &a);
    fprintf(fi, "b = %3i &b = %8u\n", b, &b);


    fclose(fi);
    return 0;

}

void  Intercambio(FILE *fo, int *p1, int *p2)
{
    int temp;

    fprintf(fo, "   Valores recibidos del programa principal:\n");
```

```
42      fprintf(fo, "      a = %3i     &a = %8u\n", *p1, p1);
43      fprintf(fo, "      b = %3i     &b = %8u\n", *p2, p2);
44
45      temp = *p1;
46      *p1 = *p2;
47      *p2 = temp;
48
49      fprintf(fo, "   Una vez realizado el intercambio sus valores son:\n");
50      fprintf(fo, "      a = %3i     &a = %8u\n", *p1, p1);
51      fprintf(fo, "      b = %3i     &b = %8u\n", *p2, p2);
52      fprintf(fo, "   Se devuelve el control al programa principal");
53
54      return;
55  }
```

Una vez ejecutado el programa, en el fichero **`Datos.dat`** se tiene:

```
En el programa principal se han asignado los valores
a =  12 &a =  2293316
b = -35 &b =  2293312

   Valores recibidos del programa principal:
     a =  12    &a =  2293316
     b = -35    &b =  2293312
   Una vez realizado el intercambio sus valores son:
     a = -35    &a =  2293316
     b =  12    &b =  2293312
   Se devuelve el control al programa principal

En el programa principal los nuevos valores son:
a = -35 &a =  2293316
b =  12 &b =  2293312
```

Ahora ya se ha realizado correctamente el intercambio. Como se observa en la definición de la función **`Intercambio`**, los argumentos *formales* son *punteros a variables*. Así pues, en la llamada a la función (LÍNEA 24) deben ponerse las direcciones de las variables y no, los valores. De esta manera, al operar con direcciones la información se transmite correctamente cuando se devuelve el control al programa principal. Esto es debido a que las operaciones se hacen *siempre* sobre las direcciones de memoria de las variables. Como consecuencia, en la salida impresa en el fichero, se observa que las direcciones de memoria de las variables ***no*** han cambiado, a diferencia del resultado del programa anterior 7.12.

Hay lenguajes (por ejemplo, FORTRAN), que en la llamada a las funciones, basta con el nombre de las variables, para transmitir automáticamente las direcciones de memoria de estas variables. Se llama *paso de argumentos* ***por referencia***. Como ya se ha comentado, el **C** no lo permite y hay que especificar explícitamente que se transmite una dirección (por medio de punteros). Por eso se dice, que en **C**, el *paso de argumentos* a las funciones se hace siempre ***por valor***.

7.6. Funciones y vectores

Hasta ahora, se ha estado trabajando con variables y funciones. El siguiente paso es construir funciones cuyos argumentos sean vectores. La cuestión clave es recordar que los nombres de los vectores son a su vez, *punteros* (véase sección: Vectores y punteros, pág. 138). Esto facilita enormemente la comunicación entre las funciones y el programa principal: no es necesario *identificar* las direcciones de todas las componentes del vector, si no que únicamente, basta con indicar la dirección de la primera componente del vector.

7.6.1. Ejemplos

Para ilustrar esta idea, se plantean unos ejemplos en los que se opera con funciones en cuyos argumentos aparecen direcciones de vectores.

Ejemplo: valor mínimo de las componentes de un vector

En el programa 7.14, se muestra como aplicación, la utilización de una función que determine el valor mínimo de las componentes de un vector dado.

Código 7.14 – *Valor mínimo de las componentes de un vector*

```
#include <stdio.h>

/* Programa que calcula el valor mínimo de las
componentes de un vector */

double VecMin(double *, int ); // *** Prototipo de la función


int main(void)
{

    //double Vector[5] = {-23.45, 467, -123, 23, 0}
    const int DIM = 5;
    double Minimo;
    int chk, i;

    FILE *fi;

    fi = fopen("Datos.dat", "w");

    double Vector[DIM];

    Vector[0] = -23.45, Vector[1] = 467,
    Vector[2] = -123,   Vector[3] = 23,
    Vector[4] = 0;

    fprintf(fi, "El vector introducido  es: \n");
    for(i = 0; i < DIM; i++){
        fprintf(fi, "V[%i] = %.2f; ", i+1, Vector[i]);
    }
    fprintf(fi, "\n\n");


    Minimo = VecMin(Vector, DIM); //*** Argumentos REALES

    fprintf(fi,"El valor más pequeño es  %.5f\n", Minimo);


    fclose(fi);
    return 0;

}
```

```
43
44  double VecMin(double *P, int N) // *** Argumentos FORMALES
45  {
46      double Min;
47      int i;
48
49      for (Min = P[0], i = 1; i < N; i++){
50          if (P[i] < Min){
51              Min = P[i];
52          }
53      }
54
55      return (Min);
56  }
```

Cuando se ejecuta el programa, en el fichero de salida se escribe:

```
El vector introducido  es:
V[1] = -23.45; V[2] = 467.00; V[3] = -123.00; V[4] = 23.00; V[5] = 0.00;

El valor más pequeño es -123.00000
```

Como se observa, una vez dimensionado e inicializado el vector (LÍNEAS 21-25), en la llamada a la función (LÍNEA 34) se introduce el nombre (**Vector**), que junto con el tamaño del vector (**DIM**), permite acceder a cualquier componente, dentro de la función. Es importante destacar, que en la definición de la función **VecMin** (LÍNEAS 44–56), se utiliza no un vector, sino un *puntero a double* (**P**) cuyo cometido es almacenar la dirección de la primera componente (véase (6.2), pág. 139). Gracias a este hecho, se establecen las equivalencias o igualdades (6.1) (pág. 139), lo que permite, en el programa **VecMin**, expresar el puntero **P** como si se tratase de un vector.

Ejemplo: ordenación de las componentes de un vector

En este caso el objetivo es diseñar una función que permita ordenar las componentes de un vector y devuelva, al programa principal, el vector ordenado. En el código 5.10 (pág. 111), ya se utilizó un procedimiento de ordenación, para colocar adecuadamente *cadenas de caracteres*. En este caso, el algoritmo de ordenación que se pretende programar es el mismo. En el programa 7.15 se muestra un posible ejemplo con dicho procedimiento.

Código 7.15 – *Ordena las componentes de un vector*

```
1   #include <stdio.h>
2
3   /* Programa que ordena las componentes
4   de un vector, de menor a mayor */
5
6   void VecOrden(double *, int );
7
8
9   int main(void)
10  {
11
12      //double Vector[5] = {-23.45, 467, -123, 23, 0}
13      const int DIM = 5;
14      int chk, i;
```

```
15
16      FILE *fi;
17
18      fi = fopen("Datos.dat", "w");
19
20      double Vector[DIM];
21
22      Vector[0] = -23.45, Vector[1] =  467,
23      Vector[2] = -123,   Vector[3] = 23,
24      Vector[4] = 0;
25
26      fprintf(fi, "El vector introducido  es: \n");
27      for(i = 0; i < DIM; i++){
28          fprintf(fi, "V[%i] = %.2f; ", i+1, Vector[i]);
29      }
30      fprintf(fi, "\n\n");
31
32      VecOrden(Vector, DIM); //** Argumentos REALES
33
34      fprintf(fi, "El vector ordenado  es: \n");
35      for(i = 0; i < DIM; i++){
36          fprintf(fi, "V[%i] = %.2f; ", i+1, Vector[i]);
37      }
38      fprintf(fi, "\n\n");
39
40      fclose(fi);
41      return 0;
42
43  }
44
45  void VecOrden(double *P, int N) //** Argumentos FORMALES
46  {
47      double Aux;
48      int i, j;
49
50      for ( i = 0; i < N-1; i++){
51          for( j = i + 1; j < N; j++){
52              if( P[i] > P[j]){
53                  Aux = P[i];
54                  P[i] = P[j];
55                  P[j] = Aux;
56              }
57          }
58      }
59
60      return;
61  }
```

Una vez ejecutado, el resultado que se obtiene en el fichero **`Datos.dat`** es

```
El vector introducido  es:
V[1] = -23.45; V[2] = 467.00; V[3] = -123.00; V[4] = 23.00; V[5] = 0.00;

El vector ordenado  es:
V[1] = -123.00; V[2] = -23.45; V[3] = 0.00; V[4] = 23.00; V[5] = 467.00;
```

A diferencia del ejemplo anterior, ahora la función **VecOrden** no devuelve ningún valor. Sin embargo, el procedimiento para operar con el vector es el mismo, que el del programa previo 7.14. Como *argumento formal* se toma un *puntero a double* que se opera como si fuese un vector. Al operar sobre las *direcciones de memoria*, el resultado se verá reflejado en el *argumento real*, en este caso, la variable vectorial **Vector**.

7.6.2. Funciones, cadenas de caracteres y estructuras

Como ya se ha comentado (véase sección 5.3, pág. 107), las *cadenas de caracteres* son vectores de caracteres. Por tanto, todo lo comentado sobre vectores es de aplicación directa a este tipo de variables multidimensionales.

Así mismo, las *estructuras* (véase sección 5.5, pág. 116) pueden dimensionarse y tratarse como vectores. No obstante, en este caso los *punteros a estructuras* (véase sección: Estructuras y punteros, pág. 150) tienen una representación un poco diferente, pero su comportamiento es análogo, al caso vectorial, una vez dimensionado.

7.7. Funciones y matrices

La forma de vincular matrices definidas en el programa principal con otras funciones, se hace de manera análoga al caso de los vectores: *a través de punteros*. Sin embargo, es claro que estos punteros son distintos a los utilizados en el caso de los vectores (véase sección: Matrices y punteros, pág. 145).

Así pues, para enlazar una matriz definida en el programa principal con una función, como *argumento real* se indica la *dirección* de la primera componente, mientras que como *argumento formal* en la definición de la función, se sitúa un puntero de *naturaleza matricial*.

Para ilustrar este procedimiento, se plantea definir una función que calcule el *producto de matrices*, cuyo algoritmo fue visto en la sección 5.4.2 (pág. 114). En el programa 7.16 se define la función **ProdMat** a través de punteros.

Código 7.16 – *Producto de matrices enteras*

```
1  #include <stdio.h>
2
3  /* Realiza el producto de matrices a
4    través de una función */
5
6  #define N 3
7  #define P 2
8  #define M N
9
10 //*** Prototipo
11 void ProdMat(int, int, int, int (* )[], int (* )[], int (* )[]);
12
13 int main(void)
14 {
15     int i, j, k;
16
17
18     int A[N][P] = {{1, 2}, {3,  4}, {5, 6}};
19     int B[P][M] = {{-4, 3, -7}, {-1, 11, 2}};
20     int C[N][M] = {0};
21
```

```
22      FILE *fi;
23
24      fi = fopen("Datos.dat", "w");
25
26      fprintf(fi,"Matriz A: \n");
27      for (i = 0; i < M; i++){
28          for(k = 0; k < P; k++){
29              fprintf(fi, "%4i", A[i][k]);
30          }
31          fprintf(fi, "\n");
32      }
33
34      fprintf(fi, "Matriz B: \n");
35      for (k = 0; k < P; k++){
36          for (j = 0; j < M; j++){
37              fprintf(fi, "%4i", B[k][j]);
38          }
39          fprintf(fi, "\n");
40      }
41
42      fprintf(fi, "Matriz producto C, inicial: \n");
43      for (i = 0; i < N; i++){
44          for(j = 0; j < M; j++){
45              fprintf(fi, "%4i", C[i][j]);
46          }
47          fprintf(fi, "\n");
48      }
49
50      ProdMat(N, M, P, A, B, C); // **** Argumentos REALES
51
52      //Salida de resultados
53      fprintf(fi,"\n ********* RESULTADO ******* \n\n");
54      fprintf(fi, "Matriz Producto C:\n");
55      for(i = 0; i < N; i++){
56          for(j = 0; j < M; j++){
57              fprintf(fi, "%4i", C[i][j]);
58          }
59          fprintf(fi, "\n");
60      }
61
62      fclose(fi);
63
64      return 0;
65  }
66  //*** Definición de la función con argumentos FORMALES
67  void ProdMat(int F, int C, int S,
68               int (*MatA)[S], int (*MatB)[C], int (*MatC)[C])
69  {
70      int i, j, k;
71
72      for (i = 0; i < F; i++){
73          for(j = 0; j < C; j++){
74              for(MatC[i][j] = 0, k = 0; k < S; k++){
75                  MatC[i][j] += MatA[i][k] * MatB[k][j];
```

```
76                  }
77              }
78          }
79          return;
80      }
```

Una vez ejecutado el programa, en el fichero **Datos.dat** se tiene:

```
Matriz A:
   1   2
   3   4
   5   6
Matriz B:
  -4   3  -7
  -1  11   2
Matriz producto C, inicial:
   0   0   0
   0   0   0
   0   0   0

 ********* RESULTADO *******

Matriz Producto C:
  -6  25  -3
 -16  53 -13
 -26  81 -23
```

En la definición de la función **ProdMat** (LÍNEAS 67–80), se utilizan *punteros matriciales* que pueden expresarse en forma matricial (véase (6.3), pág. 148), lo que simplifica la programación del algoritmo. Por otra parte, en la llamada a la función (LÍNEA 50), la dirección de la matriz *producto* **C**, entra como un argumento más. Esto permite que cuando se realice el producto matricial en la función, los valores se sitúen en las direcciones de la matriz **C**, obteniéndose el resultado deseado, cuando se devuelve el control de la ejecución al programa principal.

7.8. Punteros a funciones

Como ya se explicado, un *puntero* señala a una dirección de memoria. Hasta ahora se ha supuesto que en esta dirección estaba contenida el valor de una variable. Sin embargo, la memoria aparte de datos, contiene código y no existe una diferencia cualitativa entre las áreas de memoria que contiene datos y aquellas que almacenan instrucciones. Así pues, un *puntero* puede, igualmente, señalar una zona de memoria que almacene código o datos. El **C** tiene la particularidad de poder señalar *código* mediante punteros. En particular, permite definir *punteros* que señalan la posición de funciones: son los ***punteros a funciones***. El nombre de una función, para **C** es realmente un *puntero*. La manera más general de *declarar* un *puntero a función* es de la forma:

```
tipo_de_retorno (*nombre_puntero) (lista de argumentos)
```

El primer paréntesis es ***fundamental*** porque si no, la declaración

```
tipo_de_retorno *nombre (lista de argumentos)
```

indica que se trata de una función que devuelve un *puntero* al tipo declarado. Por tanto, no es lo mismo. La ventaja de utilizar *punteros a funciones* es que permite utilizar las funciones como parámetros de otras funciones. Ejemplos de esta situación se encuentran frecuentemente en *Cálculo numérico*. Un método de integración o un método de resolución de ecuaciones depende de la función o de la ecuación a resolver. Los *punteros a función*

permiten escribir una función genérica, que reciba como parámetro un puntero a la función o ecuación con la que ha de operarse.

Veamos primeramente el ejemplo del *Método de la secante*, que es un procedimiento básico en *Cálculo numérico*, para la resolución de una ecuación no lineal (véase Mathews y Fink [14]). En este caso es necesario introducir la ecuación. En el código 7.17 la función asociada, entra como parámetro:

Código 7.17 – *Método de la secante*

```
#include <stdio.h>

/* Ejemplo del método de la secante */

double fun(double );
double f_secante(FILE *, double , double , double (* )(double ) );

int main(void)
{
      double p1, p0;
      double solfin = 0.;


      FILE *fi;
      fi = fopen("Datos.dat", "w");

      printf("Introduce p0 -> ");
      scanf("%lf", &p0);
      while(getchar() != '\n');  /* Vacía el buffer */
      printf("\n\n");
      printf("Introduce p1 -> ");
      scanf("%lf", &p1);
      while(getchar() != '\n');  /* Vacía el buffer */

      fprintf(fi, " p1: %f y p0: %f\n\n", p1, p0);
      solfin = f_secante(fi, p0, p1, fun);

      fprintf(fi, "\n\n");
      fprintf(fi, "La solución es: %f\n", solfin);

      fclose(fi);

      return 0;

}
/* Función asociada a la ecuación */
double fun(double x)
{
       double y;

       y = pow(x,3) - 2.*pow(x,2)- x + 2;
       return (y);
}
/* Función que codifica el método de la secante*/
double f_secante(FILE *fo, double x0, double x1, double (* f)(double ))
{
```

```
47          int iter = 0;
48          int MaxIter = 500;
49          double delta = 1.e-9, epsilon = 1.e-9;
50          double dif = 1., val = 1.;
51          double sol = 0.;
52
53          while(delta < dif && epsilon < val && iter++ <= MaxIter){
54               sol = x1 - f(x1)*(x1-x0)/( f(x1)-f(x0) );
55               dif = fabs(sol-x1);
56               val = fabs(f(sol));
57               x0 = x1;
58               x1 = sol;
59               fprintf(fo,"dif: %.20f; val: %.20f; ", dif, val);
60               fprintf(fo, "sol: %.10f; Iter: %d\n", sol, iter);
61          }
62          return (sol);
63     }
```

La salida se almacena en el fichero `Datos.dat` y resulta

```
 p1: 1.500000 y p0: 0.000000

dif: 0.35714285714285721000; val: 0.26239067055393578000; sol: 1.1428571429; Iter: 1
dif: 0.25843503230437892000; val: 0.24297010925184734000; sol: 0.8844221106; Iter: 2
dif: 0.12425180295477500000; val: 0.01727193764225795300; sol: 1.0086739135; Iter: 3
dif: 0.00824643603209218590; val: 0.00085477213578550519; sol: 1.0004274775; Iter: 4
dif: 0.00042935692747880783; val: 0.00000375890759652059; sol: 0.9999981205; Iter: 5
dif: 0.00000187985400035284; val: 0.00000000080393647317; sol: 1.0000000004; Iter: 6


La solución es: 1.000000
```

Se trata de resolver la ecuación

$$x^3 - 2x^2 - x + 2 = 0 \tag{7.1}$$

que tiene tres raíces: $x = -1$, $x = 1$ y $x = 2$. La función asociada a esta ecuación es:

$$f(x) = x^3 - 2x^2 - x + 2,$$

que queda definida en el programa en las LÍNEAS 37-43, mediante el nombre de **`fun`**. Para establecer el código del *método de la secante* se utiliza la función **`f_secante`** en las LÍNEAS 45-63. En el último argumento de la definición de esta función (LÍNEA 45), se establece un *puntero a función* de tipo *double*, mediante la expresión

```
          double (* f)(double )
```

Posteriormente, en la llamada (véase LÍNEA 26), el nombre de la función: **`fun`** actúa como puntero.

En algunos casos es necesario introducir más de una función, como sucede en el *método de Newton–Raphson*, también un algoritmo básico, para la resolución de ecuaciones no lineales. En este procedimiento es necesario disponer de la función que define la ecuación y su derivada. Es posible resolver esta situación de manera análoga al caso anterior. Sin embargo, puede resultar más útil, en ciertas ocasiones definir un vector de *punteros a funciones*. Seguiremos este procedimiento. Para ello se establece el programa 7.18:

Código 7.18 – *Método de Newton–Raphson*

```c
#include <stdio.h>
/* Ejemplo del método de Newton-Raphson */

double fun(double );
double dfun(double );
double f_NR(FILE *, double, double (* [])(double ) );

int main(void)
{
      double  p0;
      double solfin = 0.;
      double (*PtrF[2])();


      FILE *fi;
      fi = fopen("Datos.dat", "w");

      printf("Introduce p0 -> ");
      scanf("%lf", &p0);
      while(getchar() != '\n');  /* Vacía el buffer */

      fprintf(fi, "El valor p0 es: %f\n\n", p0);

      PtrF[0] = fun;
      PtrF[1] = dfun;
      solfin = f_NR(fi, p0, PtrF);

      fprintf(fi, "\n\n");
      fprintf(fi, "La solución es: %f; \n", solfin);

      fclose(fi);

      return 0;

}

double fun(double x)
{
       double y;

       y = pow(x,3) - 2.*pow(x,2)- x + 2;
       return (y);
}

double dfun(double x)
{
       double y;

       y = 3*pow(x,2) - 4*x -1;
       return (y);
}

double f_NR(FILE *fo, double x0, double (* f[2])(double ))
{
```

```
55          int iter = 0;
56          int MaxIter = 500;
57          double delta = 1.e-9, epsilon = 1.e-9;
58          double dif = 1., val = 1.;
59          double sol = 0.;
60
61          while(delta < dif && epsilon < val && iter++ <= MaxIter){
62               sol = x0 - f[0](x0)/f[1](x0);
63               dif = fabs(sol-x0);
64               val = fabs(f[0](sol));
65               x0 = sol;
66               fprintf(fo,"dif: %.15f; val: %.15f; ", dif, val);
67               fprintf(fo, "sol: %.10f; Iter: %d\n", sol, iter);
68          }
69          return (sol);
70    }
```

con el siguiente resultado, almacenado en el fichero `Datos.dat`

```
El valor p0 es: -0.500000

dif: 1.071428571428571; val: 5.247813411078717; sol: -1.5714285714; Iter: 1
dif: 0.413412953605880; val: 1.076883866081182; sol: -1.1580156178; Iter: 2
dif: 0.140676029477473; val: 0.105546050034807; sol: -1.0173395883; Iter: 3
dif: 0.017094421471253; val: 0.001471301792507; sol: -1.0002451669; Iter: 4
dif: 0.000245116800518; val: 0.000000300440693; sol: -1.0000000501; Iter: 5
dif: 0.000000050073445; val: 0.000000000000012; sol: -1.0000000000; Iter: 6


La solución es: -1.000000;
```

Siguiendo con la ecuación anterior (7.1), la función asociada a la ecuación y su derivada, se definen a través de las funciones: **fun** y **dfun**, respectivamente. El objetivo ha sido definir un *vector de punteros a función*, con dos componentes: una que apunte a **fun** y otra que señale a **dfun**. La declaración de este vector, se plantea en la LÍNEA 12 mediante la sentencia

```
       double (*PtrF[2])();
```

Posteriormente, a este vector se le asigna las direcciones (mediante el nombre) de las funciones **fun** y **dfun** (LÍNEAS 24-25). El vector de *punteros a funciones*: **PtrF** entra como tercer argumento en la llamada a la función **f_NR** (véase LÍNEA 26), que resuelve el *algoritmo de Newton-Rapshon*. En la declaración de variables para el subprograma **f_NR** (LÍNEA 53) se define el vector de dos componentes de *punteros a función*, mediante la expresión:

```
           double (* f[2])(double )
```

7.9. Recursividad

Un método se dice ***recursivo*** cuando está basado en un procedimiento que al aplicarse reiteradamente la resolución termina aplicándose a situaciones muy simples. La soluciones a ciertos problemas se pueden plantear de manera *recursiva*. Esto significa por tanto, que su solución se apoya en la solución del mismo problema, pero para un caso más fácil.

En matemáticas es frecuente encontrar situaciones en los que es posible definir conceptos en términos de sí mismo. Un ejemplo clásico es la definición de *factorial de un número natural*. En este caso se tiene

$$n! = \begin{cases} 1 & \text{si } n = 0 \\ n \cdot (n-1)! & \text{si } n > 0 \end{cases}$$

Se trata de un proceso *recursivo* porque se está utilizando el propio concepto de factorial (aparece $(n-1)!$) en la propia definición. De esta manera se tendría:

$$n! = n \cdot (n-1)! = n \cdot (n-1) \cdot (n-2)! = \cdots = n \cdot (n-1) \cdots 2 \cdot 1$$

No todos los lenguajes permiten codificar algoritmos recursivos, pero en el caso del **C**, sí. Para la función *factorial* su codificación puede verse en el programa 7.19.

Código 7.19 – *Función: factorial*

```c
#include <stdio.h>
/* Ejemplo de programa recursivo */

int factorial(int );  /* Prototipo de la función definida */

int main(void)
{

      int N;

      FILE *fi;
      fi = fopen("Datos.dat", "w");

      printf("Introduce el valor de N -> ");
      scanf("%d", &N);
      fprintf(fi, " El factorial de %d es %d", N, factorial(N));


      fclose(fi);

      return 0;
}

/* Función factorial */
int factorial(int x)
{
        if (x == 0)
           return (1);
        return (x * factorial(x-1));
}
```

La salida para el valor $n = 5$ introducido sería:

```
El factorial de 5 es 120
```

Otro ejemplo interesante es la *Sucesión de Fibonacci*. Consiste en una sucesión de números naturales que comienza con los números 1 y 1 y a partir de estos, *cada término es la suma de los dos anteriores*. Se puede

escribir de manera *recursiva* de la siguiente forma:

$$F(n) = \begin{cases} 1 & \text{si } n = 1 \\ 1 & \text{si } n = 2 \\ F(n-1) + F(n-2) & \text{si } n > 2 \end{cases}$$

El código para este caso sería

Código 7.20 – *Función: Sucesión de Fibonacci*

```
#include <stdio.h>
/* Ejemplo de programa recursivo */

int fibonacci(int );  /* Prototipo de la función definida */

int main(void)
{

      int N;

      FILE *fi;
      fi = fopen("Datos.dat", "w");

      printf("Introduce el valor de N (>=1) -> ");
      scanf("%d", &N);
      fprintf(fi, " El término %d de Fibonacci es %d", N, fibonacci(N));


      fclose(fi);

      return 0;
}

/* Sucesión de Fibonacci */
int fibonacci(int x)
{
        if (x == 1 || x == 2)
           return (1);
        else
           return (fibonacci(x-1) + fibonacci(x-2));
}
```

y la salida para $N = 14$ sería

```
El término 14 de Fibonacci es 377
```

Cualquier problema que puede resolverse de manera *recursiva*, admite solución mediante un proceso *iterativo*. A veces, el procedimiento de pasar un algoritmo recursivo a uno iterativo, no es directo y puede dar lugar a códigos más complejos de seguir. No obstante, los algoritmos recursivos suelen ser menos eficientes (en tiempo) que los equivalentes iterativos. Esto es debido, por lo general, a la sobrecarga debida al funcionamiento interno de recursividad (es necesario definir la *pila* en donde se almacenan los resultados intermedios) y a la posible solución recursiva.

El hecho que la recursividad presente cierta sobrecarga (en tiempo de ejecución del algoritmo) frente a la iteración, no significa que no se deba utilizar, sino que se debe emplear cuando sea apropiado. Cuando el problema a resolver sea relativamente simple, se recomienda utilizar la solución iterativa ya que, como se ha comentado, es más eficiente. Sin embargo, las soluciones a ciertos problemas más complejos pueden ser mucho más fáciles si se utiliza la *recursividad*. En este caso, la claridad y simplicidad del algoritmo recursivo prevalece frente al tiempo extra que puede consumir en la ejecución. Un ejemplo de esta situación aparece con el problema de las *Torres de Hanoi*.

7.9.1. Ejemplo: Torres de Hanoi

Las *Torres de Hanoi* constituyen un claro ejemplo en el que la solución *recursiva* resulta mucho más simple que la correspondiente solución iterativa. El problema consiste en lo siguiente:

> *Sean tres estacas verticales y n discos de distintos radios que pueden insertarse en las estacas formando torres. Inicialmente los n discos están todos situados en la primera estaca, por orden decreciente de radios (veáse figura 7.3). Se trata de pasar los n discos de la primera estaca a la última siguiendo las reglas:*

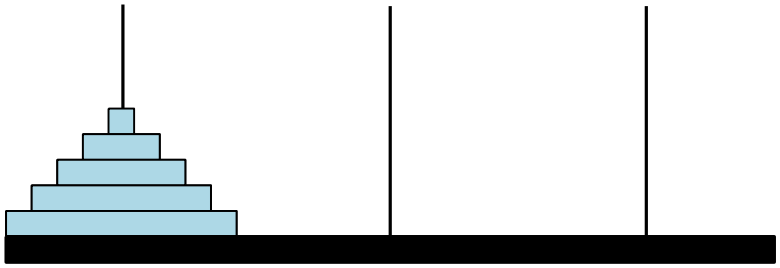


Figura 7.3 – *Torres de Hanoi*

1. *En cada paso se mueve un único disco.*
2. *En ningún paso se puede colocar un disco, sobre otro de radio menor.*
3. *Puede usarse la estaca central como auxiliar.*

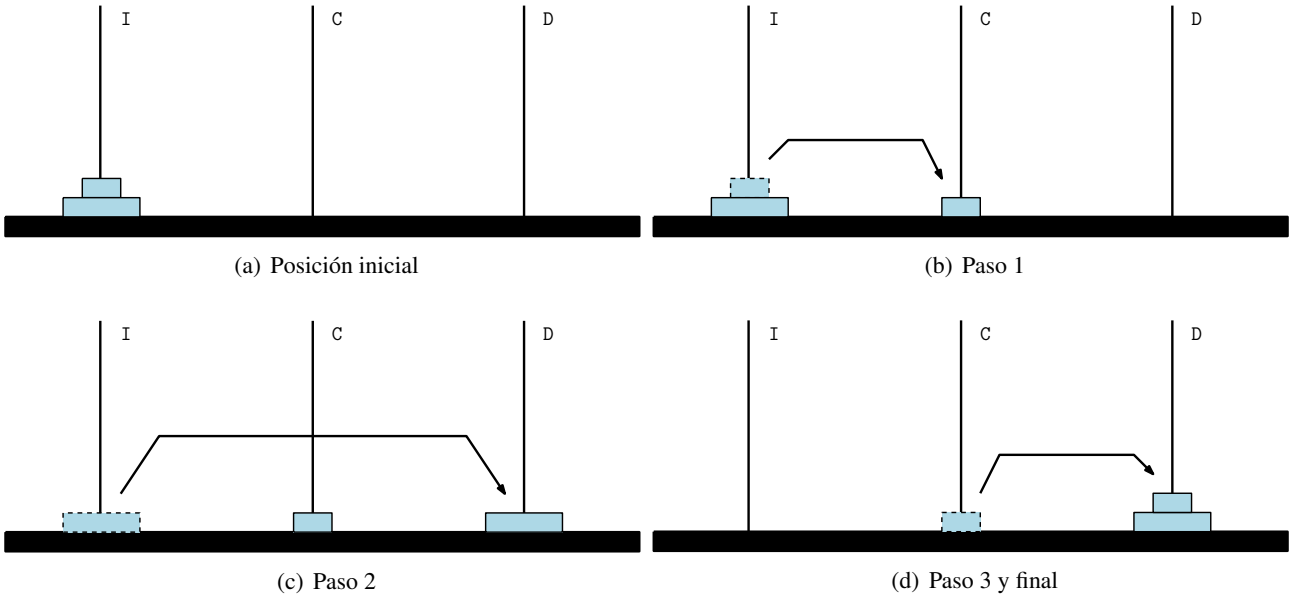
Para una torre de altura uno, la solución es trivial, como puede verse en la figura 7.4

(a) Posición inicial (b) Paso 1 y final

Figura 7.4 – *Movimiento para una única torre*

La solución para una torre de dos discos, consta de sólo tres pasos, como puede visualizarse en la figura 7.5.

A la vista de las dos situaciones anteriores, cabe la posibilidad de generalizar la solución para n discos. En efecto, si se supone el problema resuelto para una torre de altura $n-1$ discos (es decir, se sabe mover $n-1$ discos de una estaca a otra, utilizando la que queda como auxiliar), el problema de pasar n discos desde la estaca `I` o *izquierda* a la `D` o *derecha*, queda resuelto del siguiente modo (véase figura 7.6):



(a) Posición inicial (b) Paso 1

(c) Paso 2 (d) Paso 3 y final

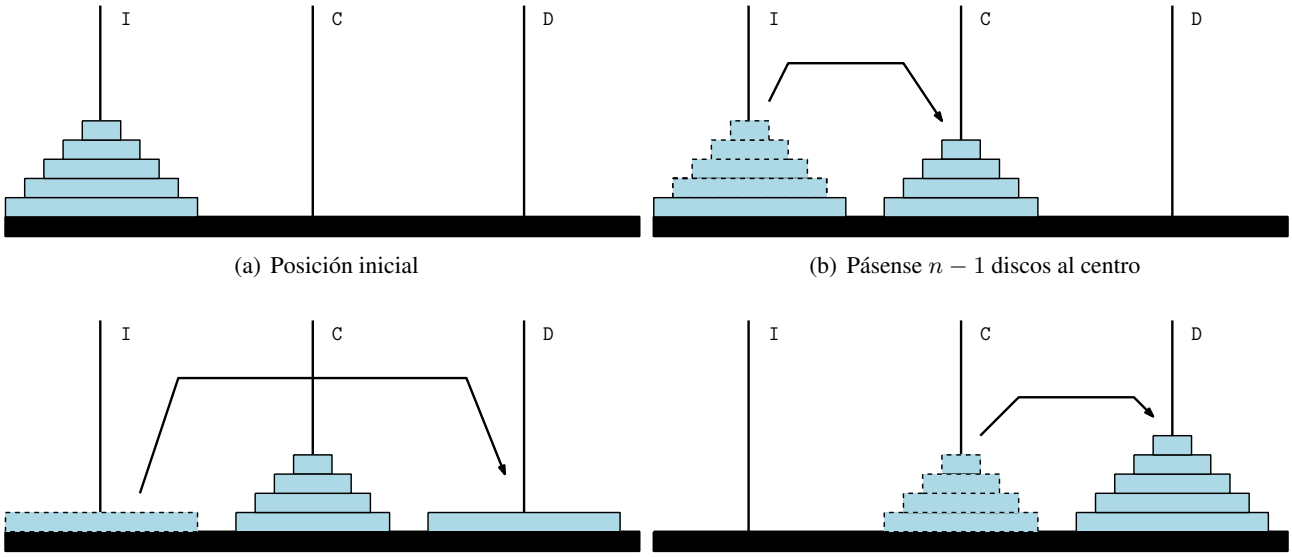
Figura 7.5 – *Movimiento de 2 torres*

1. Pásense $n-1$ discos de la estaca **I** o *izquierda* a la estaca **C** o *central*, utilizando la estaca **D** o *derecha* como auxiliar.
2. Muévase el disco de mayor tamaño de la estaca **I** o *izquierda* a la estaca **D** o *derecha*.
3. Pásense $n-1$ discos de la estaca *central* o **C** a la estaca *derecha* o **D**, utilizando la estaca *izquierda* o **I** como auxiliar.

Este es el proceso *recursivo* que se programa en el código 7.21

Código 7.21 – *Algoritmo de las Torres de Hanoi*

```
#include <stdio.h>

void T_hanoi(FILE *, int, char, char, char );

int main(void)
{
      char ini = 'I', aux = 'C', fin = 'D';
      int N;

      FILE *fi;

      fi = fopen("Datos.dat", "w");

      printf("\nNúmero de discos -> ");
      scanf("%d",&N);

      fprintf(fi, "Los movimientos para mover %d discos son: \n", N);

      T_hanoi(fi, N, ini, aux, fin);

      fclose(fi);
```



(a) Posición inicial

(b) Pásense $n-1$ discos al centro

(c) Pásese el disco n a la derecha

(d) Pásense $n-1$ discos a la derecha

Figura 7.6 – *Movimiento de n torres*

```
22
23          return 0;
24      }
25
26      void T_hanoi(FILE *fo, int N, char ini, char aux, char fin)
27      {
28          if(N == 1)
29              fprintf(fo, "%c -> %c", ini, fin);
30          else{
31            T_hanoi(fo, N-1, ini, fin, aux); /* Mueve n-1 discos al centro */
32            fprintf(fo, "\n%c -> %c\n", ini, fin); /* Mueve el disco mayor a la decha
                  . */
33            T_hanoi(fo, N-1, aux, ini, fin); /* Mueve n-1 discos del centro a la
                  decha.*/
34          }
35
36          return;
37      }
```

La salida se almacena en el fichero `Datos.dat`. Para $N = 3$ discos, el resultado es:

```
Los movimientos para mover 3 discos son:
I -> D
I -> C
D -> C
I -> D
C -> I
C -> D
I -> D
```

La cuestión clave está en la definición de la función *recursiva* **`T_hanoi`**, que muestra en pantalla los movimientos para mover `N` discos de la estaca identificada con la letra **`I`** (*Izda.*) a la estaca **`D`** (*Dcha.*) y que utiliza la estaca **`C`** (*Centro*), como auxiliar. La posición de los argumentos es muy importante para fijar los movimientos de los discos.

La función tiene como solución *simple*, cuando la torre está formada por un disco ($N = 1$). En este caso, basta realizar un único movimiento: mover la torre de la estaca **I** a la **D**. En otro caso, si la torre está formada por más de un disco, entonces

1. Se deben trasladar los $N - 1$ discos desde la estaca **I** a la estaca **C**. Para ello se utiliza la estaca **D**, como auxiliar. Esto se consigue utilizando la función **T_hanoi**, pero cambiando el orden de los argumentos, establecidos en la definición. Así pues, en la LÍNEA 29 se llama a la función de la forma

```
T_hanoi(fo, N-1, ini, fin, aux);
```

   en donde se ha cambiado el orden de los argumentos. Esto quiere decir ahora, que los $N - 1$ discos se trasladan desde `ini` (que contiene a **I**) a `aux` (que contiene a **C**), utilizando a `fin` (que contiene a **D**) como estaca auxiliar.

2. En la LÍNEA 30 se establece que se traslade el disco que queda, de la estaca **I** a la **D**.

3. La última etapa consiste en trasladar los $N - 1$ discos de la estaca del *centro* (**C**) a la *derecha* (**D**). Esto se consigue con la sentencia de la LÍNEA 31. En ella se toma la estaca *izda.* (**I**) como auxiliar, cambiando de nuevo, el orden de los argumentos en la función **T_hanoi**. Por esta razón, primero aparece la variable `aux` (que contiene **C** y es desde donde se van a trasladar los discos), después aparece la variable `ini` (que contiene a **I** que es la estaca que se utiliza como auxiliar) y por último está la variable `fin` (que contiene **D** que es la estaca de destino).

Obsérvese que el número de pasos necesarios para trasladar los N discos de la estaca A a la estaca C será de $2^N - 1$.

7.10. Problemas

1. Haz un programa en el que se defina la función **Presentacion()** cuyo objetivo es que escriba en pantalla, el número de matrícula, el nombre y los apellidos. Mejora esta función para que los mismos datos se escriban en un fichero de salida.

2. A partir de la función **f_potencia** ya vista en el código 7.7 (pág. 171) realiza un programa en la que se defina y utilice una función que calcule la potencia de dos números enteros. Esto es, dados m y n, que calcule mediante una función, el valor: m^n y lo imprima en un fichero.

3. Utiliza la función definida anteriormente para calcular las *variaciones con repetición de m elementos tomados de n en n* ya planteado en la sección 4.6 de Problemas, pág. 90.

4. Utiliza la función **Factorial** definida en el código 7.8 (pág. 172), para calcular las *variaciones sin repetición de m elementos tomados de n en n* y para determinar el número combinatorio

$$\binom{m}{n} = \frac{m!}{n!(m-n)!}$$

conceptos ya vistos en la sección 4.6 de Problemas, pág. 90. Una vez visto el concepto de *recursividad*, reescribe la función **Factorial** para que el cálculo se realice de forma recursiva.

5. Escribe un programa que dado un número $m \in \mathbb{N}$, calcule todos los números combinatorios siguientes:

$$\binom{m}{0}, \binom{m}{1}, \binom{m}{2}, \ldots, \binom{m}{m}$$

y muestre todos ellos en un fichero de salida, en el mismo orden. En este ejercicio y a partir del problema anterior, es obligado que se defina y utilice una función **`combinatorio`** cuyo argumento de entrada sean dos números positivos y la salida su número combinatorio correcto.
(Propuesto en examen: curso – 2014/15)

6. Se llama *binomio de Newton* a la igualdad:

$$(a+b)^m = \sum_{k=0}^{m} \binom{m}{k} a^{m-k}\, b^k$$

a) Se pide que se escriba un programa que dado un número natural $m \geqslant 1$ calcule 2^m, **SIN** utilizar la función `pow(x,y)` de la biblioteca de **C**. Por tanto, se tendrá que definir la función correspondiente.

b) Además el programa debe evaluar el *binomio de Newton* (esto es, programar el miembro de la derecha de la igualdad anterior), para comprobar que el cálculo anterior es correcto y mostrarlo en un fichero de salida. Después deben analizarse si ambos resultados son iguales. Si los son, la salida debe ser

`Se cumple la igualdad establecida por el Binomio de Newton`

si no,

`Uff!! algo falla en el programa`.

(Propuesto en examen: curso – 2014/15)

7. Escribe una función que describa el *Algoritmo de Horner* y utilízala para determinar el valor de un polinomio dado en el punto x_0.

8. Dado un vector real de doble precisión:

$$\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$$

se pide, que realices un programa que permita:

a) Calcular el valor de las normas:

$$\|\boldsymbol{x}\|_2 = \sqrt{\boldsymbol{x}\cdot\boldsymbol{x}}; \qquad \|\boldsymbol{x}\|_1 = \sum_{i=1}^{n} |x_i|; \qquad \|\boldsymbol{x}\|_\infty = \max_{1\leqslant i\leqslant n} |x_i|$$

b) Invertir las componentes del vector inicial.

c) Ordenar de menor a mayor, las componentes del vector inicial.

Es obligatorio que se definan y utilicen funciones para cada cuestión planteada. Por tanto debe haber 5 funciones: **`Norma_2`**, **`Norma_1`**, **`Norma_inf`**, **`Invierte`** y **`Ordena`**. Las salidas para las normas debe ser real, con al menos 13 cifras significativas. Una ejecución típica del programa debe ser:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 1111111
**************************

Introduce la dimensión del vector -> -1
Por favor introduce un número positivo !!!! -> 5
```

```
Introduce V[1] = 0
Introduce V[2] = -1
Introduce V[3] = 1
Introduce V[4] = 3
Introduce V[5] = -6
```

La salida típica en un fichero de datos debe ser:

```
El vector introducido  es:
V[1] = 0.00; V[2] = -1.00; V[3] = 1.00; V[4] = 3.00; V[5] = -6.00;
La norma 2 es: 6.85565

La norma 1 es: 11.00000

La norma infinito es: 6.00000

El vector invertido es:
V[1] = -6.00; V[2] = 3.00; V[3] = 1.00; V[4] = -1.00; V[5] = 0.00;

El vector ordenado es:
V[1] = -6.00; V[2] = -1.00; V[3] = 0.00; V[4] = 1.00; V[5] = 3.00;
```

(Propuesto en examen: curso 2014/15)

9. Teniendo en cuenta la *tabla de números primos* realizada en la sección: **Problemas** de la pág. 126, haz un programa que construya una tabla con la descomposición en factores primos de un número natural dado N. Para ello, debe construirse una función que determine los números primos menores que N y utilizar este resultado, para obtener la descomposición en factores primos.

10. Se llama *número de Armstrong* al número que verifica la propiedad de que *la suma de cada uno de sus dígitos elevado a la potencia del número de dígitos, coincide con su valor*. Por ejemplo el número 371 es un *número de Armstrong* ya que cumple

$$3^3 + 7^3 + 1^3 = 371$$

o bien, el número 92727

$$9^5 + 2^5 + 7^5 + 2^5 + 7^5 = 92727$$

Otros *números de Armstrong* son: $1, 153, 370, 371, 407, 1634, 8208, 9474, 54748, 92727, 93084, 548834$. Se pide que hagas una función de manera que dado un número natural N, determine si es de *Armstrong* o no. A partir de esta función, haz un programa que calcule los *números de Armstrong* menores de 10^8 (hay 27 y menores a 10^9, 31) y los imprima en un fichero.

11. Haz una función tal que dada una matriz permita permutar dos filas de la matriz.

12. Haz una función tal que dada una matriz $\mathbf{A} \in \mathcal{M}_{n\times m}(\mathbb{R})$, se obtenga su transpuesta.
(Sugerencia: constrúyase la matriz transpuesta sobre una matriz cuadrada de tamaño $\max(n, m)$).

13. Dada una matriz $\mathbf{A} \in \mathcal{M}_{n\times m}$, se dice que uno de sus coeficientes es un *punto de silla* si es el máximo de su fila y mínimo de su columna o bien, si es el mínimo de su fila y máximo de su columna. Se pide que hagas un programa que determine los posibles puntos de silla de una matriz entera dada $\mathbf{A} \in \mathcal{M}_{n\times m}(\mathbb{Z})$. Se deben definir y utilizar al menos 3 funciones: **`Maximo`**, **`Minimo`** y **`PuntoSilla`** desde la cual, se debe imprimir la solución. Una ejecución típica del programa debe ser:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 1111111
**************************

Introduce las filas de la matriz -> 3
Introduce las columnas de la matriz -> 3

Introduce A[1][1] = 1
Introduce A[1][2] = 2
Introduce A[1][3] = 3

Introduce A[2][1] = 5
Introduce A[2][2] = 6
Introduce A[2][3] = 7

Introduce A[3][1] = 4
Introduce A[3][2] = 1
Introduce A[3][3] = 9
```

La salida típica en un fichero de datos debe ser:

```
La matriz matriz introducida es
 A[1][1] = 1;  A[1][2] = 2;  A[1][3] = 3;
 A[2][1] = 5;  A[2][2] = 6;  A[2][3] = 7;
 A[3][1] = 4;  A[3][2] = 1;  A[3][3] = 9;

 El coeficiente A[1][3] = 3 es punto de silla.
 El coeficiente A[2][1] = 5 es punto de silla.
```

(Propuesto en examen: junio – 2015).

14. Haz un programa que calcule el área de un polígono *simple*, mediante la utilización de una función **`area`** (véase apéndice 9.1.2, pág. 215). Un argumento de entrada de la función debe ser un *puntero al vector* del *tipo* **`punto`**, que contiene las coordenadas de los vértices.

15. Haz un programa que calcule el *m.c.d.* de dos números naturales mediante el *algoritmo de Euclides*, definida en una función cuya codificación debe estar realizada de manera recursiva.

16. En un curso de álgebra lineal se define la *Traza de una matriz* $\boldsymbol{A} \in \mathcal{M}_{m\times n}$ como la suma de las componentes de la diagonal principal. Esto es

$$\boldsymbol{A} = \begin{bmatrix} a_{1,1} & \dots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \dots & a_{m,n} \end{bmatrix} \Longrightarrow Tr(\boldsymbol{A}) = \sum_{i=1}^{k} a_{ii} \qquad \text{donde } k = \min(n, m)$$

En el caso de matrices del mismo tamaño, el concepto de *traza* se puede asociar al de un *operador lineal*.

En este ejercicio se pide que dada una matriz *simétrica aleatoria* $\boldsymbol{A} \in \mathcal{M}_{n\times n}$ calcules su *componente desviatoria* esto es

$$\boldsymbol{A}_D = \boldsymbol{A} - \frac{1}{n} Traza(\boldsymbol{A})\boldsymbol{I}$$

donde $\boldsymbol{I}$ es la matriz identidad. El concepto de *componente desviatoria* es una generalización matemática de la *componente desviatoria* del *tensor de tensiones*, que indica cuánto se aparta el estado tensional un sólido, respecto de un estado hidrostático o isotrópico.

Para realizar el ejercicio se debe:

a) definir una función llamada `MatSimAl` que obtenga la matriz simétrica aleatoria, en la cual se pida los extremos del intervalo en donde estarán los números aleatorios, así como la semilla para generarlos.

El proceso para generar esta matriz es muy sencillo. No obstante, se puede seguir la siguiente sugerencia

> Como argumento de la función debe estar el puntero a una matriz cuadrada. A continuación, en la parte triangular superior de la matriz se situarán los números aleatorios (el proceso es similar al realizado en clase de prácticas, para imprimir figuras triangulares de ejercicios planteados en el capítulo de *Sentencias de Control*). Es decir,
>
> $$a_{i,\,j} = \alpha \qquad \text{cumpliendo } i \leqslant j \text{ y donde } \alpha \text{ es un número aleatorio, que irá variando}$$
>
> Una vez asignados estos valores, se rellenan las componentes de la parte triangular inferior con la regla:
>
> $$a_{j,\,i} = a_{i,\,j}$$
>
> (ya planteado en el ejercicio 6 del capítulo de *Datos estructurados*).

b) definir una función llamada `Traza` que determine la traza de una matriz cualquiera.

c) definir una función llamada `MatDesv` que calcule la *componente desviatoria* de la matriz dada $\boldsymbol{A}$, quedando almacenada en la propia matriz $\boldsymbol{A}$.

d) escribir, desde el programa principal, la *componente desviatoria* en un fichero.

Una ejecución típica del programa con salida en pantalla sería:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 1111111
**************************

Introduce el nombre del fichero -> Salida.dat

Introduce el tamaño de la matriz ( > 1) -> -7
Introduce el tamaño de la matriz ( > 1) -> 3


Introduce los extremos del intervalo (enteros) -> 5 -1

Introduce la semilla -> -1

Se genera una matriz  de números aleatorios de tamaño 3 x 3

El intervalo es [-1, 5]
La semilla es -1

La matriz aleatoria simétrica es:
```

```
3.00;  -1.00;   2.00;
-1.00;   3.00;  -1.00;
2.00;  -1.00;   1.00;


La componente desviatoria  es:
0.67;  -1.00;   2.00;
-1.00;   0.67;  -1.00;
2.00;  -1.00;  -1.33;
```

(Propuesto en examen: curso 15/16).

17. En un curso de álgebra lineal se define la *Traza de una matriz* $\boldsymbol{A} \in \mathcal{M}_{m\times n}$ como la suma de las componentes de la diagonal principal. Esto es

$$\boldsymbol{A} = \begin{bmatrix} a_{1,1} & \dots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \dots & a_{m,n} \end{bmatrix} \Longrightarrow Tr(\boldsymbol{A}) = \sum_{i=1}^{k} a_{ii} \qquad \text{donde } k = \text{mín}(n, m)$$

En el caso de matrices del mismo tamaño, el concepto de *traza* se puede asociar al de un *operador lineal*.

En este ejercicio se pide que dado un vector *aleatorio real* $\boldsymbol{v} \in \mathbb{R}^n$, se calcule el valor de la *traza* del producto diádico del vector $\boldsymbol{v}$ por sí mismo. Esto es

$$Tr(\boldsymbol{v} \otimes \boldsymbol{v})$$

Para ello se debe:

a) definir una función llamada `VectorAleatorio` que genere el vector aleatorio $\boldsymbol{v}$, en la cual se pida los extremos del intervalo en donde estarán los números aleatorios, así como la semilla para generarlos.

b) definir una función llamada `ProdTensorial`, que a partir del vector $\boldsymbol{v}$ se genere la matriz correspondiente al *producto diádico o tensorial* del vector $\boldsymbol{v}$ por sí mismo.

c) definir una función llamada `Traza` que determine la traza de una matriz cualquiera y sea utilizada para obtener el resultado pedido. Su aplicación debe hacerse desde el programa principal.

Una ejecución típica del programa con salida en pantalla sería:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 1111111
*************************

Introduce el nombre del fichero -> Salida.dat

Introduce la cantidad de números a generar ( > 0) -> -7
Introduce la cantidad de números a generar ( > 0) -> 5

Introduce los extremos del intervalo (enteros) -> 2 -1

Introduce la semilla -> -3
```

```
******** Salida de resultados ********

El intervalo es [-1, 2]
La semilla es -3
El vector aletorio es de tamaño 5 con valores:
-0.99744; -0.24540; -0.96219;  1.61400; -0.02228;

La matriz del producto diádico es:
0.99488;   0.24477;   0.95972;  -1.60986;   0.02222;
0.24477;   0.06022;   0.23612;  -0.39607;   0.00547;
0.95972;   0.23612;   0.92580;  -1.55297;   0.02144;
-1.60986;  -0.39607;  -1.55297;   2.60500;  -0.03596;
0.02222;   0.00547;   0.02144;  -0.03596;   0.00050;


El valor de la traza es 4.58640
```

(Propuesto en examen: curso 15/16).

18. Dado un PVI (problema de valor inicial)

$$y' = e^{-y^2} P(x), \qquad y_0 = y(x_0) = y(0) = P(0)$$

siendo $P(x) = a_0 x^n + a_1 x^{n-1} + a_2 x^{n-2} + \cdots + a_n$ un polinomio de grado n con coeficientes enteros. Se desea aproximar el valor de la solución $y(x)$ de la edo en el punto $x = 1$ mediante la sucesión definida por

$$y_{k+1} = y_k + \frac{e^{-y_k^2} P(x_k)}{M} \quad k = 0, 1, \ldots, M-1, \quad \text{siendo } x_i = \frac{i}{M} \qquad i = 0, 1, \ldots, M$$

En este ejercicio se pide que dado un polinomio con coeficientes enteros $P(x)$ obtengas la sucesión que genera el método para aproximar $y(1) \approx y_{\text{M}}$.

Para realizar el ejercicio se debe:

a) introducir desde teclado el nombre de un fichero en el que se escribirán los datos de salida.

b) introducir el número de pasos, M, para generar la sucesión que aproxima $y(1)$, así como el grado del polinomio $P(x)$, n.

c) definir una función llamada `GenPol` que obtenga los coeficientes enteros aleatorios del polinomio $P(x)$, en la cual se pida los extremos del intervalo en donde estarán los números aleatorios, así como la semilla para generarlos. Los coeficientes del polinomio resultante serán **reales de doble precisión**, se imprimirán por pantalla y en el fichero desde el programa principal.

d) definir una función llamada `Pol` que calcule, y devuelva, el valor del polinomio $P(x)$ para un valor real cualquiera x, que deberá ser **real de doble precisión**.

e) definir una función llamada `Euler` que genere la sucesión de valores $y_i \quad i = 0, 1, \ldots, M$ que será devuelta al programa principal en un vector.

f) por último, desde el programa principal debe imprimirse en el fichero y por pantalla la sucesión de puntos $(x_i, y_i) \quad i = 0, 1, 2, \ldots, M$, siendo $y_{\text{M}} \approx y(1)$

Una ejecución típica del programa con salida en pantalla sería:

```
***** Nombre: Jdkdkdkd
****  Apellidos: Skdkdkd Rdkdkdkd
**** Matricula: 1111111
**************************

Introduce el nombre del fichero -> Salida.dat

Introduce el grado del polinomio P(x), n>0 -> -7
Introduce el grado del polinomio P(x), n>0 -> 3

Introduce el número de pasos M (entero positivo) a realizar para aproximar y(1

Introduce los extremos del intervalo (enteros) -> 5 -2

Introduce la semilla -> -1

******** Salida de resultados ********

Coeficientes (vector V ) generados->
coef.de grado 0-> V[0]=1.00  coef.de grado 1-> V[1]=1.00
coef.de grado 2-> V[2]=4.00  coef.de grado 3-> V[3]=3.00

Sucesión generada:
Salida de los puntos x(i) y sus aproximaciones y(i)
i   x(i)     y(i)
0   0.0000   1.0000
1   0.1000   1.0368
2   0.2000   1.0758
3   0.3000   1.1193
4   0.4000   1.1690
5   0.5000   1.2259
6   0.6000   1.2899
7   0.7000   1.3598
8   0.8000   1.4336
9   0.9000   1.5091
10  1.0000   1.5842
```

Propuesto en examen: junio – 2016).

19. Se pide que realices un programa que determine el vector $\boldsymbol{y} \in \mathbb{R}^n$, resultado del producto de la matriz $\mathbf{A} \in \mathcal{M}_{n\times n}$ con el vector $\boldsymbol{x} \in \mathbb{R}^n$. Matricialmente:

$$\begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} a_{1,1} & \dots & a_{1,n} \\ \vdots & \ddots & \vdots \\ a_{n,1} & \dots & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} \iff y_i = \sum_{j=1}^{n} a_{i,j}\, x_j \qquad \text{con} \quad i = 1, 2, \dots, n$$

Para ello se debe:

a) definir una función llamada **`VecMatInt`** en la que se establezcan las componentes del vector $\boldsymbol{x}$ y de la matriz cuadrada $\mathbf{A}$. Si la introducción se realiza mediante números aleatorios, en esta función se debe introducir el intervalo en donde estarán los números aleatorios, así como la semilla. Si se opta por introducir las componentes por teclado, deberá programarse de manera clara y ordenada para que el usuario sepa qué componentes del vector $\boldsymbol{x}$ y de la matriz cuadrada $\mathbf{A}$ está introduciendo.

b) definir una función llamada **`Prod`** en la que se realice el producto pedido.

c) imprimir, desde el programa principal, el vector $\boldsymbol{y}$ resultado del producto obtenido en la función **`Prod`**.

Una ejecución típica del programa con salida en pantalla sería:

```
Introduce el nombre del fichero -> Salida.dat

Introduce la dimension del vector y de la matriz ( > 0) -> -7
Introduce la dimension del vector y de la matriz ( > 0)  -> 3

Introduce los extremos del intervalo (enteros) -> 5 -2

Introduce la semilla -> -1
```

La salida en el fichero debe ser

```
***** Salida de resultados *******

**** Nombre: Jalslsl
**** Apellidos: Kskdiekdkm Rosldoe
**** Número de matrícula: 111111
**********************************

El intervalo es [-2, 5]
La semilla es -1
El vector aletorio es de tamaño 3 con valores:
1.00;  1.00;  4.00;

La matriz aleatoria es:
3.00   2.00   4.00
3.00   3.00   0.00
-2.00  -1.00  -2.00


El vector producto es:
21.00;  6.00; -11.00;
```

Propuesto en examen: julio – 2016).

CAPÍTULO 8

Iniciación a Dev-C++

8.1. Introducción

Para realizar y ejecutar los programas, en este curso de *Introducción a la programación en C*, se utiliza el *entorno* DEV–C++ 5.11–TDM-GCC 4.9.2 que se puede descargar gratuitamente de

http://orwelldevcpp.blogspot.com.es/

DEV–C++ utiliza como compilador una adaptación a *Windows* del GCC, llamado TDM-GCC 4.9.2. Con esta versión del compilador se pueden emplear novedades de los estándares **C99** y **C11**. Por otra parte, este entorno tiene la ventaja que funciona correctamente en WINDOWS 7 y WINDOWS 8.1 siendo capaz de ajustarse a la naturaleza de 32 o 64 *bits* del propio sistema sistema operativo.

En este capítulo se trata de enseñar a cómo se puede realizar un programa fuente, compilarlo y ejecutarlo, dentro de este entorno de programación.

8.2. Creación, compilación y ejecución de un fichero fuente

8.2.1. Introducción

Es importante señalar (véase sección 1.5, pág. 9), que el nombre de un *fichero fuente* en **C** debe tener la extensión `.c`. Esto es, debe ser de la forma:

`NombreDelFichero.c`

Es muy recomendable que el **`NombreDelFichero`**, **no** tenga espacios en blanco. Es decir, un nombre de la forma:

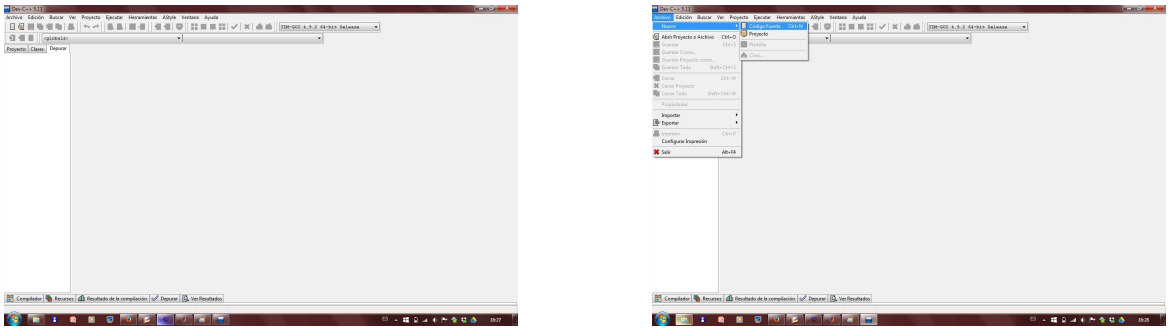
`Esto es un fichero de prueba.c` o así: **`Prueba 1.c`**

no debe hacerse. Es más conveniente poner el nombre así:

`Prueba1.c` o así: **`Prueba_1.c`** o así: **`Prueba01.c`**

8.2.2. Creación

Una vez, que se tiene instalado el *entorno* DEV–C++, con el puntero del ratón se hace *click* dos veces, en el icono del entorno DEV–C++ que debe estar sobre el *escritorio* de WINDOWS. Una vez realizado, aparece una pantalla como muestra la figura 8.1(a) A continuación se selecciona ***Archivo*** y dentro del submenú ***Nuevo*** se

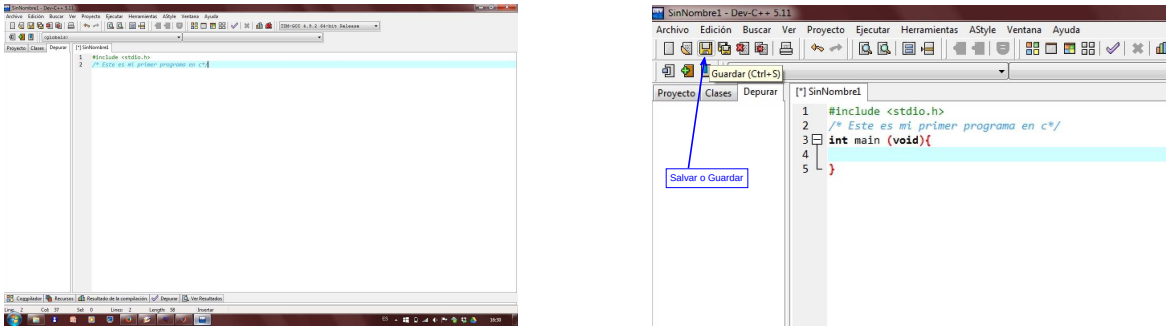


(a) Pantalla de inicio de DEV–C++ (b) Creación del código fuente

Figura 8.1 – *Creación de un fichero fuente*

opta por ***Código fuente***, tal y como muestra la figura 8.1(b).

En la pestaña superior aparece **`SinNombre1`** y ya se puede escribir texto de **C**, tal y como muestra la figura 8.2(a) Durante el proceso de escritura se puede ir *salvando* el fichero, para no perder lo escrito. Para ello



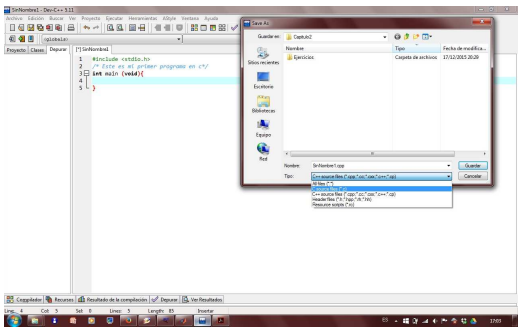
(a) Creación del código fuente (b) Guarda el texto tecleado

Figura 8.2 – *Creación de un fichero fuente*

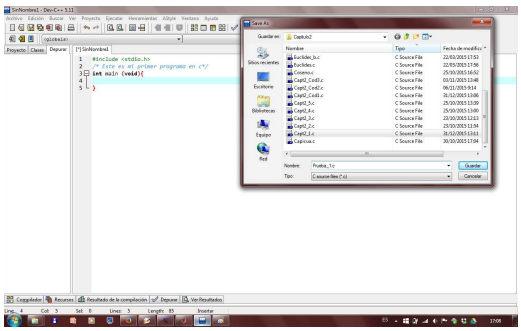
se recurre al icono del *disco* tal y como muestra la figura 8.2(b). En el momento en que se hace *click* con el puntero del *ratón*, se abre un menú desplegable como muestra el gráfico 8.3(a). Hay que seleccionar la opción: **`C source files (*.c)`**. El entorno DEV-C++ toma *por defecto* ficheros fuente para el **C++**. Puesto que nos interesa ficheros fuente para el **C**, hay que indicarlo explícitamente. Una vez seleccionada esta opción, ya se puede teclear el nombre del fichero, tal y como muestra la figura 8.3(b). En el momento que se seleccione **`Guardar`** ya aparece en la pestaña superior el nombre seleccionado **`Prueba_1`**.

8.2.3. Compilación

Una vez terminado el programa, como se observa en la figura 8.4(a), se opta por *compilar* el programa. Para ello se utilizan los iconos que se muestran en la figura 8.4(b). Se puede utilizar el de ***compilar*** o bien el de ***compilar y ejecutar***.

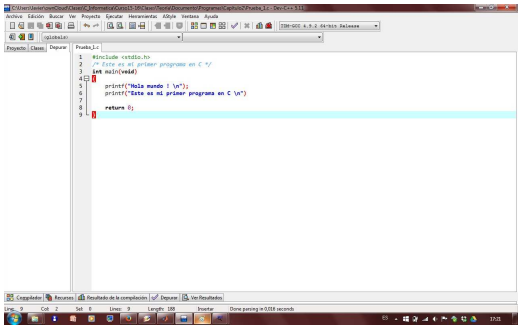


(a) Guarda el código fuente

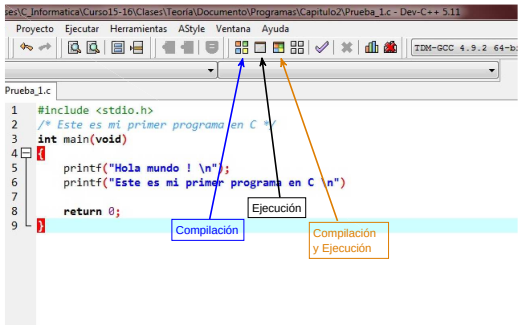


(b) Fichero fuente

Figura 8.3 – *Nombra el código fuente*



(a) código fuente



(b) Iconos para compilar y ejecutar

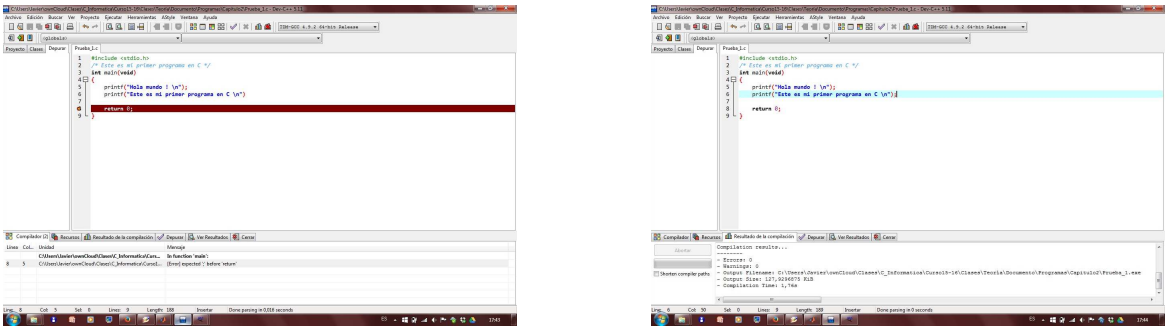
Figura 8.4 – *Compilación y ejecución*

Una vez seleccionado el icono de ***compilación***, se compila el fichero fuente y se obtiene la pantalla que muestra la figura 8.5(a). Esto indica que hay un error. En este caso, no se ha puesto el `;` al final de la sentencia. Una vez corregido, se debe volver a compilar y se obtiene ya el proceso sin errores, como muestra la figura 8.5(b).

8.2.4. Ejecución

Una vez que la compilación ha sido exitosa, se procede a la ejecución, seleccionando el icono de ***ejecución***, visto en la figura 8.4(a). Una vez seleccionado, aparece la pantalla que muestra la figura 8.6.

Es importante señalar, que para que una modificación en el fichero fuente tenga su reflejo en la ejecución del programa es ***indispensable***, la compilación previa del código fuente una vez realizada dicha modificación.



(a) Compilación con errores (b) Compilación correcta

Figura 8.5 – *Proceso de compilación*

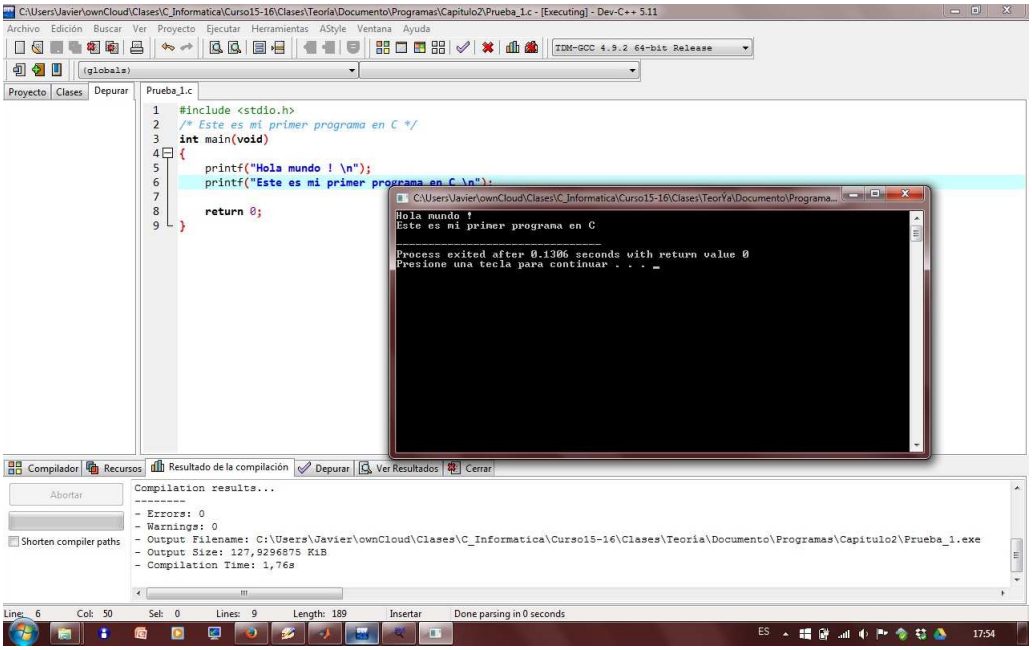


Figura 8.6 – *Ejecución del programa*

CAPÍTULO 9

Un poco de Geometría Computacional

9.1. Polígonos

9.1.1. Introducción

Un ***polígono*** puede definirse como una región *conexa* del plano, acotada por una sucesión de segmentos rectos (véase figura 9.1). Los segmentos que limitan la región plana son llamados ***lados*** y los puntos en los

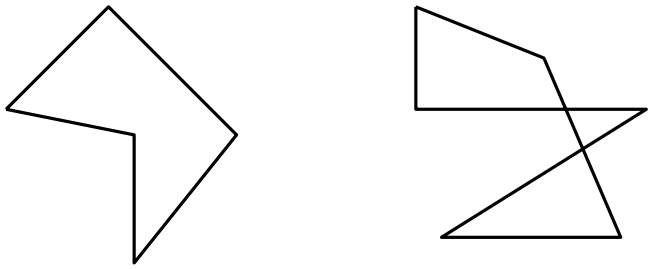


Figura 9.1 – *Ejemplos de polígonos*

que se unen los lados, ***vértices***. Los lados consecutivos forman una curva ***cerrada*** *suave a trozos* que se llama ***frontera, borde o contorno*** del polígono[1].

Si el contorno del polígono no se interseca a sí mismo, se dice que es un *polígono simple*. Además, se dice que el polígono está ***positivamente orientado*** si al recorrer su contorno, la región que encierra queda a la izquierda (es la llamada *regla del sacacorchos*, véase figura 9.2).

9.1.2. Área de un polígono

El área de un *polígono simple* se puede deducir a partir del ***Teorema de Green*** (véase por ejemplo, Marsden–Tromba [13] o Larson *et al.* [12])

Teorema 9.1.1 (Teorema de Green) *Sea D una región con frontera ∂D cerrada simple, suave a trozos y orientada positivamente. Si $M(x,y), N(x,y), \partial M/\partial y(x,y), \partial N/\partial x$ (x,y) son continuas en un conjunto abier-*

[1] A veces, el polígono hace referencia exclusivamente al contorno, no al interior. En este documento no se seguirá este criterio. Se llamará polígono al contorno, junto a su interior.

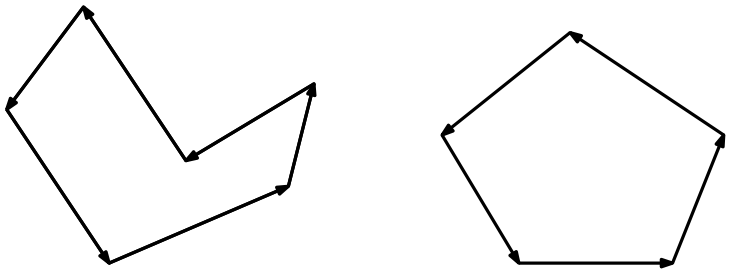


Figura 9.2 – *Ejemplos de polígonos* simples orientados *positivamente*

to que contiene a D, entonces

$$\int_{\partial D} M\,dx + N\,dy = \iint_D \left(\frac{\partial N}{\partial x} - \frac{\partial M}{\partial y}\right) dxdy$$

Este teorema, se puede aplicar para calcular el área de la región D. En efecto, si se define

$$M(x,y) = -\frac{y}{2} \qquad \text{y} \qquad N(x,y) = \frac{x}{2}$$

entonces por el *teorema de Green*,

$$\iint_D dxdy = \text{Área(D)} = \frac{1}{2}\int_{\partial D} x\,dy - y\,dx$$

La idea es aplicar este resultado para determinar el área del polígono P. Supongamos que se tiene un polígono P cuyo contorno viene determinado por los vértices: $[p_1, \ldots, p_n]$ (se ha de entender que p_1 es el vértice siguiente a p_n)

$$\partial P = \overline{p_1p_2} \cup \overline{p_2p_3} \cup \ldots \cup \overline{p_{n-1}p_n} \cup \overline{p_np_1}$$

positivamente orientado. Entonces,

$$\text{Área(P)} = \frac{1}{2}\int_{\partial P} x\,dy - y\,dx = \frac{1}{2}\sum_{i=1}^{n}\int_{\overline{p_ip_{i+1}}} x\,dy - y\,dx \qquad \text{con } p_{n+1} = p_1$$

por tanto se trata de resolver la *integral de línea*

$$\int_{\overline{p_ip_{i+1}}} x\,dy - y\,dx \qquad \text{con} \quad i = 1, \ldots, n$$

Para ello, se parametriza el segmento $\overline{p_ip_{i+1}}$, de manera que el contorno del polígono P se mantenga positivamente orientado. Suponiendo que $p_i = (x_i, y_i)$ y $p_{i+1} = (x_{i+1}, y_{i+1})$ entonces

$$\boldsymbol{\sigma}(t) = \begin{cases} x(t) = x_i + t(x_{i+1} - x_i) \\ y(t) = y_i + t(y_{i+1} - y_i) \end{cases} \qquad \text{con} \quad t \in [0,1]$$

por tanto

$$\boldsymbol{\sigma}'(t) = (x_{i+1} - x_i, y_{i+1} - y_i)$$

así pues, la integral de línea queda:

$$\int_{\overline{p_ip_{i+1}}} x\,dy - y\,dx = \int_0^1 x(t)\cdot y'(t) - y(t)\cdot x'(t)\,dt$$
$$= \int_0^1 -y_i(x_{i+1}-x_i) + x_i(y_{i+1}-y_i)\,dt = x_iy_{i+1} - y_ix_{i+1} = \begin{vmatrix} x_i & y_i \\ x_{i+1} & y_{i+1} \end{vmatrix}$$

De esta forma:

$$\text{Área(P)} = \frac{1}{2}\sum_{i=1}^{n}\int_{\overline{p_ip_{i+1}}} x\,dy - y\,dx = \frac{1}{2}\sum_{i=1}^{n}\begin{vmatrix} x_i & y_i \\ x_{i+1} & y_{i+1} \end{vmatrix} \qquad \text{con } p_{n+1} = p_1$$

9.1.3. Polígonos convexos

En ciertas ocasiones resulta importante determinar si un polígono simple es convexo o no. Un polígono simple es convexo si la región que encierra es convexa (esto es, para cualesquiera par de puntos contenidos en esta región, el segmento que los une, también está contenido. En la figura 9.3 se muestran un polígono convexo y otro no.

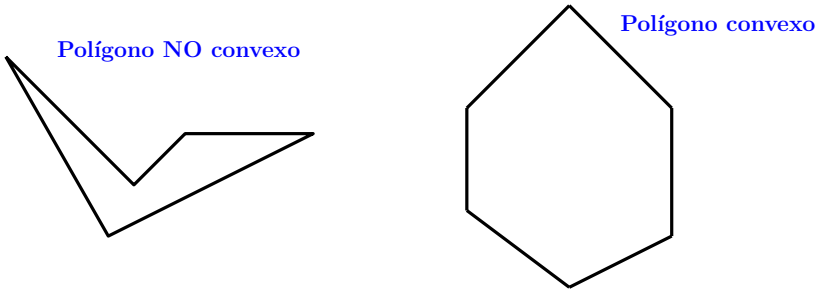


Figura 9.3 – *Ejemplos de un polígono no convexo y otro convexo*

Una forma de saber si un polígono es convexo o no, consiste en ir comprobando si los tres vértices consecutivos están positivamente orientados. Esto es, si los vértices p_i, p_{i+1}, p_{i+2} cumplen $\Delta(p_i, p_{i+1}, p_{i+2}) > 0$, siendo

$$\Delta(p_i, p_{i+1}, p_{i+2}) = \begin{vmatrix} x_i & y_i & 1 \\ x_{i+1} & y_{i+1} & 1 \\ x_{i+2} & y_{i+2} & 1 \end{vmatrix}$$
$$= (x_{i+1}-x_i)\cdot(y_{i+2}-y_i) - (x_{i+2}-x_i)\cdot(y_{i+1}-y_i)$$
$$= \begin{vmatrix} (x_{i+1}-x_i) & (y_{i+1}-y_i) \\ (x_{i+2}-x_i) & (y_{i+2}-y_i) \end{vmatrix}$$

Si $\Delta(p_i, p_{i+1}, p_{i+2}) = 0$ es que están sobre una recta.

9.1.4. Determinación si un punto es interior a un polígono

En ciertas ocasiones resulta necesario determinar si un punto en interior o no, a un polígono. La idea es *lanzar un rayo* (realmente una semirecta) desde el punto en cuestión y contar el número de intersecciones con el borde del polígono. Si el número de intersecciones es *impar* el punto está *fuera* y si es *par* está *dentro*.

Sea $p_0 = (x_0, y_0)$ el punto en cuestión y se construye la recta que pasa por dos vértices consecutivos: $p_i = (x_i, y_i)$ y $p_{i+1} = (x_{i+1}, y_{i+1})$

$$y = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i) + y_i$$

entonces se evalúa

1. $x_i \leqslant x_0 \leqslant x_{i+1}$ o $x_{i+1} \leqslant x_0 \leqslant x_i$
2. se calcula la ordenada de la recta:

$$r = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x_0 - x_i) + y_i$$

y se comprueba si $y_0 < r$.

Si se cumple alguna de las dos opciones del primer punto y el segundo, entonces se produce una intersección de la semirecta con el polígono (véase figura 9.4).

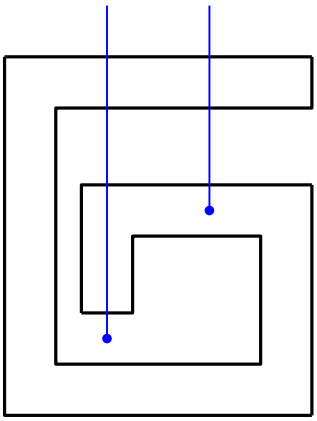


Figura 9.4 – *Se comprueba si un punto es interior o no a un polígono*

Un posible algoritmo del proceso es el siguiente

Código 9.1 – *Algoritmo de determinación si un punto es exterior o interior*

```
1   Dado p0 y P = [p1,....,pN]
2   Determina si p0 está dentro o no de P
3
4   int cortes = 0;
5   Mientras i < N
6       P = ( p[i+1].y - p[i].y )/( p[i+1].x - p[i].x );
7       cond1 = p[i].x <= p0.x  &&  p0.x < p[i+1].x
8       cond2 = p[i+1].x <= p0.x  &&  p0.x < p[i].x
9       encima = p0.y < P * (p0.x - p[i].x) + p[i].y
10      Si se cumple ( (cond1 || cond2) && encima ) entonces
11            cortes = cortes  + 1;
12      Fin Si
13  Fin Mientras
14  return (cortes)
```

Bibliografía

[1] Bruce, E. *Thinking in C++, Volume 1 (2nd. Edition)*. Prentice Hall (2000).

[2] Bryant, R. E. y O'Hallaron, D. R. *Computer systems: a programmer's perspective (Second Edition)*, tomo 2. Prentice Hall Upper Saddle River (2011).

[3] Cantone, D. *Introducción a la Informática*. Starbook (2010).

[4] Deitel, P. y Deitel, H. *C for programmers (with a introduction to C11)*. Deitel/Prentice Hall (2013).

[5] Floyd, T. *Fundamentos de sistemas digitales (novena edición)*. Pearson/Prentice Hall (2006).

[6] Gil, F. A. M. y Quetglás, G. M. *Introducción a la programación estructurada en C*, tomo 64. Universitat de València (2003).

[7] Horton, I. *Beginning C. From novice to professional. (Fourth Edition)*. Apress (2013).

[8] Infante del Río, J.-A. y Rey Cabezas, J. M. *Métodos Numéricos: Teoría, problemas y prácticas con Matlab*. Pirámide (2007).

[9] Kernighan, B. W. y Ritchie, D. M. *El Lenguaje de Programación C*. Pearson Educación (1991).

[10] Knuth, D. *The art of computer programming 1: Fundamental algorithms 2: Seminumerical algorithms 3: Sorting and searching* (1968).

[11] Kochan, S. G. *Programming in C*. Pearson Education (2014).

[12] Larson, R. E.; Hostetler, R. P.; Edwards, B. H.; Rapún, L. A. y López, J. L. P. *Cálculo y geometría analítica*. McGraw-Hill (1989).

[13] Marsden, J. E. y Tromba, A. J. *Cálculo Vectorial*. Pearson (1998).

[14] Mathews, J. H. y Fink, K. D. *Métodos numéricos con MATLAB*. Prentice Hall (2000).

[15] Prata, S. *C Primer Plus, sixth edition*. Addison–Wesley (2014).

[16] Press, W. H.; Teukolsky, S. A.; Vetterling, W. T. y Flannery, B. P. *Numerical recipes in C*. Cambridge university press Cambridge (1996).

[17] Reese, R. *Understanding and Using C pointers*. .º'Reilly Media, Inc."(2013).

[18] Rodríguez-Losada, D.; Muñoz, J. y García Cena, C. *Introducción a la programación en C*. Sección de Publicaciones de la E. T. S. I. Industriales, Universidad Politécnica de Madrid (2008).

[19] Wirth, N. *Algoritmos + estructuras de datos = programas*. Ediciones del Castillo (1980).